2004

Pattern-based model transformation: a metamodel-based approach to model evolution

Sheena Judson Miller
Louisiana State University and Agricultural and Mechanical College, sjudson@lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations
Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_dissertations/3447

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
PATTERN-BASED MODEL TRANSFORMATION: A METAMODEL-BASED APPROACH TO MODEL EVOLUTION

A Dissertation

Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in

The Department of Computer Science

by
Sheena Judson Miller
B.S., Southern University, 1987
M.S., Southern University, 1995
December 2004
To
My Father, Albert Judson
My Brother, A. Lenard Judson
and
My Sister, Rosa Denise Edwards
Your memory will always be with me

To
Blaine and Joei
Without your love and support
this would not have been possible
Acknowledgements

The completion of this dissertation was only possible with the support and contributions of many people. I would like to begin by thanking my husband, Blaine, for standing behind me and for believing that I would finish this dissertation. Without his love, support and understanding, this dissertation would not be. I would like to thank my daughter Joei, for not feeling neglected when I did not have time to do those “girly” things with her. I would also like to thank my mother, Jessie H. Judson, for instilling in me the importance of a good education and making me believe that the world was mine to conquer. Behind the scenes were my friends and family who were always there for me during my doctoral studies. I am beholden to you all for your fellowship, reassurance, and support.

I would like to thank the family who gave the gift of life on February 7, 1998. Without their sacrifice and thoughtfulness, I would not have been able to enjoy the things that life has to bring nor complete this research. I thank you from the bottom of my heart.

Special thanks to Dr. Doris L. Carver, my advisor, for her generous support, guidance, and patience and allowing me to explore an area of great interest to me and for her confidence in my work. Thanks to Dr. Gestur Olafsson (my minor professor from Mathematics) for his mentoring and his encouragement. I also thank, in no particular order, Dr. Jianhua Chen, Dr. Donald Kraft, and Dr. Xue-Bin Liang (the Dean’s Representative) for serving as members of my doctoral committee.

Finally, I would like to thank Dr. Robert France for his assistance and support in the forming of this research. Without his thorough knowledge of the OMG standards, I would not have been able to complete this research.
Table of Contents

Acknowledgements ................................................................................................................................. iv

List of Tables .................................................................................................................................................. vii

List of Figures ................................................................................................................................................ ix

Abstract ....................................................................................................................................................... xii

Chapter 1 Introduction ................................................................................................................................ 1
  1.1 Software Evolution via Modeling ........................................................................................................ 1
  1.2 Research Objective .............................................................................................................................. 2
    1.2.1 Pattern-Based Model Transformation ......................................................................................... 5
  1.3 Structure of Dissertation ..................................................................................................................... 6

Chapter 2 Background and Related Research .............................................................................................. 7
  2.1 Unified Modeling Language (UML) .................................................................................................... 7
    2.1.1 UML Class Diagram ................................................................................................................... 10
    2.1.2 UML Architecture ..................................................................................................................... 10
    2.1.3 The UML Metamodel ............................................................................................................... 12
  2.2 Object-Constraint Language (OCL) .................................................................................................... 14
  2.3 Role-Based Metamodeling Language (RBML) ................................................................................ 14
    2.3.1 Static Role Model ...................................................................................................................... 15
    2.3.2 RBML Conformance ................................................................................................................ 17
  2.4 Design Patterns .................................................................................................................................... 18
  2.5 Model Evolution through the Transformation of Models .................................................................... 20
  2.6 Related Work on Design Pattern Transformation ............................................................................. 21
  2.7 Summary ............................................................................................................................................... 23

Chapter 3 Action Language for Pattern-Based Model Transformations .................................................. 25
  3.1 UML Action Semantics ....................................................................................................................... 25
  3.2 Pattern-Based Action Language (PBAL) ............................................................................................. 26
  3.3 PBAL Constructs .................................................................................................................................. 27
    3.3.1 Create Instances of Objects ...................................................................................................... 27
    3.3.2 Destroy Instances of Objects ................................................................................................... 27
    3.3.3 Create Link ................................................................................................................................ 28
    3.3.4 Destroy Link ............................................................................................................................. 28
    3.3.5 Retrieve Instances of Classifier Objects ................................................................................... 28
    3.3.6 Retrieve Instances of Operations ............................................................................................ 29
  3.4 Summary ............................................................................................................................................... 29

Chapter 4 Controlled Model Evolution ...................................................................................................... 30
  4.1 Pattern-Based Model Transformation ............................................................................................... 30
  4.2 A Metamodel Approach to Specifying Transformations ................................................................... 31
    4.2.1 Metamodel-Level Transformations .......................................................................................... 31
    4.2.2 Model-Level Transformations ................................................................................................ 33
  4.3 Model-Level Transformation Specification ....................................................................................... 34
  4.4 Validation of Model-Level Transformation ....................................................................................... 35
    4.4.1 Composition ............................................................................................................................. 35
  4.5 Summary ............................................................................................................................................... 36

Chapter 5 Characterizing Creational Pattern Transformations ..................................................................... 37
5.1 The Abstract Factory Pattern ........................................................................................................ 38
5.2 AF Pattern Model Transformation Example ................................................................................. 39
5.3 AF Transformation Patterns .......................................................................................................... 42
  5.3.1 Source Schema ..................................................................................................................... 42
  5.3.2 Transformation Schema ...................................................................................................... 43
  5.3.3 Transformation Constraint ................................................................................................. 45
  5.3.4 Metamodel-Level Pre- and Postconditions ........................................................................ 46
5.4 AF Pattern Transformation Specification ...................................................................................... 46
  5.4.1 Model-Level Pre- and Postconditions ................................................................................ 49
  5.4.2 Composition of AF Model-Level Pre- and Postconditions ................................................. 52
5.5 Pre- and Postconditions Conformance ......................................................................................... 61
5.6 Structural Conformance .................................................................................................................. 65
5.7 Summary ....................................................................................................................................... 71

Chapter 6 Characterizing Behavioral Pattern Transformations ........................................................... 72
  6.1 The Visitor Pattern .................................................................................................................... 72
  6.2 Visitor Transformation Pattern ............................................................................................... 73
    6.2.1 Source Schema ................................................................................................................ 73
    6.2.2 Transformation Schema .................................................................................................. 75
    6.2.3 Transformation Constraint ............................................................................................. 77
    6.2.4 Metamodel-Level Pre- and Postconditions .................................................................... 77
  6.3 Visitor Pattern Transformation Specification ............................................................................. 78
    6.3.1 Model-Level Pre- and Postconditions ............................................................................ 80
    6.3.2 Composition of Visitor Model-Level Pre- and Postconditions ........................................ 83
  6.4 Visitor Transformation Pre- and Postcondition Conformance .................................................... 94
  6.5 Structural Conformance ............................................................................................................. 96
  6.6 Summary .................................................................................................................................. 103

Chapter 7 Characterizing Structural Pattern Transformations .......................................................... 104
  7.1 The Bridge Pattern .................................................................................................................... 104
  7.2 Bridge Transformation Pattern ................................................................................................ 104
    7.2.1 Source Schema ................................................................................................................ 105
    7.2.2 Transformation Schema .................................................................................................. 106
    7.2.3 Metamodel-Level Pre- and Postconditions .................................................................... 108
  7.3 Bridge Pattern Transformation Specification ............................................................................. 108
    7.3.1 Model-Level Pre- and Postconditions ............................................................................ 111
    7.3.2 Composition of Bridge Model-Level Pre- and Postconditions ......................................... 115
  7.4 Bridge Transformation Pre- and Postcondition Conformance .................................................... 127
  7.5 Structural Conformance ............................................................................................................. 129
  7.6 Summary .................................................................................................................................. 133

Chapter 8 Summary and Conclusion ................................................................................................ 134
  8.1 Contributions ............................................................................................................................ 135
  8.2 Future Work .............................................................................................................................. 135

References ........................................................................................................................................... 137

Appendix: Reusable Mini-Transformations ......................................................................................... 140

Vita ...................................................................................................................................................... 144
List of Tables

Table 3.1. Pattern-Based Action Language (PBAL) Syntax ................................................................. 29
Table 4.1. Form for Expressing Transformation Specification .......................................................... 34
Table 5.1. AF Transformation Specification ..................................................................................... 47
Table 5.2. Action 1 - Create Factory Class ....................................................................................... 49
Table 5.3. Action 2 - Create Factory Operations .............................................................................. 50
Table 5.4. Action 3 - Create Specialized Factory Classes ................................................................. 50
Table 5.5. Action 4 - Specialized Factory Operations ....................................................................... 51
Table 5.6. Action 5 - Composite Product Operation ....................................................................... 52
Table 5.7. Action 6 - Remove CreateOp from Client Class ............................................................... 52
Table 5.8. Composed Action 1 (pre1^post1) With Action 2 (pre2^post2) ........................................... 53
Table 5.9. Composed pre^post2' with Action 3 (pre3^post3)' ............................................................ 56
Table 5.10. Composed pre^post3' with Action 4 (pre4^post4)' ......................................................... 58
Table 5.11. Composed pre^post4' with Action 5 (pre5^post5)' ........................................................... 59
Table 5.12. Composed pre^post5' with Action 6 (pre6^post6)' .......................................................... 60
Table 5.13. Model-Level Pre- and Postcondition (preModel^postModel) ....................................... 60
Table 6.1. Visitor Transformation Program ......................................................................................... 79
Table 6.2. Action 1 - Create a Visitor Class ......................................................................................... 81
Table 6.3. Action 2 - Create Concrete Visitor Classes ................................................................. 81
Table 6.4. Action 3 - Create Visiit Operations .................................................................................. 82
Table 6.5. Action 4 - Create a Accept Operations .............................................................................. 83
Table 6.6. Action 5 - Remove ElemOp from Element and Concrete Element .................................. 83
Table 6.7. Action 1 (pre1^post1) Composed With Action 2 (pre2^post2) ........................................ 84
Table 6.8. Compose pre^post2' with Action 3 (pre3^post3)' ........................................................... 88
Table 6.9. Compose pre^post3' with Action 4 (pre4^post4)' ............................................................ 92
Table 6.10. Compose pre^post4' with Action 5 (pre5^post5) ............................................................ 92
List of Figures

Figure 1.1. Generic View of Model Transformation ....................................................................................................2
Figure 1.2. Overview of Research ................................................................................................................................3
Figure 1.3. Overview of Pattern-Based Model Transformation Approach...........................................................5
Figure 2.1. Inputs into the Development of UML [Quatrani 2002]..............................................................8
Figure 2.2. The Diagrams of the UML [OMG 2003] ...............................................................................................9
Figure 2.3. UML Class Diagram ......................................................................................................................10
Figure 2.4. Four-layer Metamodel Hierarchy [OMG 2003b] ..........................................................11
Figure 2.5. Fragment of the UML Metamodel [OMG 2003a]...........................................................................12
Figure 2.6. SRM Structure [Kim 2004] ........................................................................................................15
Figure 2.7. Structurally Conforming Class Diagram [Kim 2004] ..............................................................17
Figure 4.1. Fragment of the Extended UML Package Structure .......................................................................31
Figure 4.2. Generic View of a Model Transformation Approach........................................................................32
Figure 5.1. Abstract Factory Design Pattern [Gamma et al., 1994]..........................................................38
Figure 5.2. MazeGame Class Diagram Source Model [Gamma et al, 1994] ....................................................39
Figure 5.3. MazeGame Sequence Diagram Source Model [adapted from Gamma et al., 1994] .................40
Figure 5.4. Transformed MazeGame Class Diagram [Gamma et al., 1994] .................................................41
Figure 5.5. Transformed MazeGame Sequence Diagram [adapted from Gamma et al., 1994] ......................41
Figure 5.6. Abstract Factory Transformation Pattern ......................................................................................42
Figure 5.7. Model to Metamodel Level Validation of Pre- and Postconditions ..................................................62
Figure 5.8. M1 to M2 Postcondition Mapping for Add Factory Class ..........................................................63
Figure 5.9. M1 to M2 Mapping for Add Factory Operations ...........................................................................63
Figure 5.10. M1 to M2 Postcondition Mapping for Add Specialized Factory Class .....................................64
Figure 5.11. M1 to M2 Postcondition Mapping for Add SpecializedFactory Operations ..............................64
Figure 5.12. M1 to M2 Postcondition Mapping for Client Operations ..........................................................65
Figure 5.13. Structurally Conforming MazeGame Source Model ..............................................................65
Figure 5.14. MazeGame Source Object Model

Figure 5.15. Add Factory Class

Figure 5.16. Add Specialized Factory Classes

Figure 5.17. Add CreatedProducts to Specialized Factory Classes

Figure 5.18. Add CreatedProducts to Factory Classes

Figure 5.19. Add CreateCompProd

Figure 5.20. Remove instances of CreateOp

Figure 5.21. MazeGame Target Model

Figure 5.22. Structurally Conforming MazeGame Target Model

Figure 6.1. Visitor Design Pattern [Gamma et al., 1994]

Figure 6.2. Visitor Transformation Pattern

Figure 6.3. M1 to M2 Postcondition Mapping Add Concrete Visitor

Figure 6.4. M1 to M2 Postcondition Mapping Add Visitor Class

Figure 6.5. M1 to M2 Postcondition Mapping Add Visit Operations

Figure 6.6. M1 to M2 Mapping Add & Delete Element and ConcreteElement Operations

Figure 6.7. UML Diagram of Equipment Inventory System

Figure 6.8. Structurally Conforming Visitor Source Model

Figure 6.9. Equipment Inventory Source Object Model

Figure 6.10. Add Visitor Class

Figure 6.11. Create ConcreteVisitor Classes

Figure 6.12. Create Visit Operations

Figure 6.13. Create Accept Operations

Figure 6.14. Remove instances of ElemOp

Figure 6.15. Equipment Inventory Target Model

Figure 6.16 Conforming Target Model

Figure 7.1. Bridge Design Pattern Structure [Gamma et al., 1994]

Figure 7.2. Bridge Transformation Pattern

Figure 7.3. M1 to M2 Mapping Product Abstraction
Abstract

Software systems continue to grow in complexity at a rapid pace, creating systems that are complex to build and evolve. The problems that accompany changes in requirements, system upgrades, and error correction produce a desire for software evolution methods that increase the efficiency and effectiveness of adapting complex software to changes. As software systems evolve, design models must be modified to accommodate the required changes. Techniques that control the changes to models in a systematic manner are a key to model evolution. A process that improves the ability to effectively modify a design, thereby enhancing design qualities, supports the need for improved model evolution techniques.

Design patterns are common forms of reusable design experiences. They offer solutions to common design problems, reduce complexity by naming and defining abstractions, and provide a foundation for building reusable software. Well-known pattern solutions are expressed in a natural language as fragments of code which are sometimes difficult to understand and implement by software modelers. With increased focus on development of model-driven approaches, rigorous descriptions of design patterns that capture solutions during design instead of implementation are needed.

This research defines an approach for the transformation of models that supports controlled model evolution. More precisely, a process for capturing design patterns in UML class diagrams is defined. This process involves defining a metamodel-level representation which specifies how a software developer can introduce design patterns into existing design models.

We defined transformation patterns as an extension of the UML metamodel to characterize source and target model elements. The transformation pattern consists of specialized metamodel elements that specify the structure of source and target metamodels. Transformation patterns were specified for the Abstract Factory, Bridge and Visitor design patterns to show how the model-level transformations can be perform on patterns that represent different functionalities.

We developed an action language to specify constructs which add, delete, retrieve and connect model elements. We used the constructs of the action language to define transformation specifications that implement model-level transformations on class diagrams. To determine the potential of this approach we manually implemented the transformation specification on a UML design.
Chapter 1
Introduction

Software systems continue to grow in complexity at a rapid pace, resulting in systems that are increasingly complex to build and evolve to meet changing requirements. Software evolution is a major software development issue. The problems that accompany changes in requirements, system upgrades, and error correction are well documented and commonly known. These problems produce a need for software evolution methods that will increase the efficiency and effectiveness of adapting complex software to changes. Software modeling, an important component of the software development and evolution processes, provides support for complexity using abstraction to address relevant details at various stages during the development process. The focus, in recent years, has been to use model-driven development approaches that treat models as the primary artifact of the development process, thus transferring the capture of solutions from implementation to design.

1.1 Software Evolution via Modeling

As software systems evolve, design models must be modified to accommodate the required changes. In the context of software modeling, model evolution is defined as the process in which modifications are successively applied to a design that has the capabilities to (1) produce new software systems from conception, (2) produce software artifacts from legacy systems, and (3) provide corrective, adaptive, and perfective maintenance after release [France & Bieman, 2001]. Model evolution can be classified as corrective, adaptive, and perfective. Corrective evolution corrects errors in the design. Adaptive evolution modifies a design model to accommodate changes in requirements and design constraints. Perfective evolution modifies a design to enhance design quality.

Design patterns and model transformations have been used successfully for model evolution [Eden et al, 1997; Cinneide, 2000; Gamma et al., 1994; Akehurst and Kent, 2002; France et al., 2004; Khriss & Keller, 1999]. A design pattern names, abstracts, and identifies key aspects of a common design structure deeming it useful for creating a reusable design [Beck et al, 1996]. Model evolution through transformation, shown in Figure 1.1, restructures software design models based on well-defined steps that preserve the intended meaning of the source design in the target design. Uncontrolled, ad-hoc approaches to model evolution can produce designs that reflect poor design characteristics. Techniques that control the changes in the model in a systematic manner are a key to
One technique for providing a systematic approach to model evolution is to apply well-defined transformations to a source model to produce a target model that reflects the required changes.

![Figure 1.1. Generic View of Model Transformation](image)

Controlled model evolution is accomplished by specifying metamodels of well-defined transformations. The specified metamodels are used to constrain how transformations are carried out at the model level. They act as checkpoints against which model-level transformations are checked for conformance.

## 1.2 Research Objective

To address the need for improved model evolution techniques, the research objective is to define a process that will improve the software engineer’s ability to perform perfective evolution efficiently. The research hypothesis is well-defined transformations defined at the metamodel level constrain how models are modified and result in controlled model evolution, thereby producing a new design model that has improved qualities.

The Unified Modeling Language (UML), developed by OMG (Object Management Group), provides a modeling platform where modeling elements are expressed in terms of a metamodel. We use UML as the model representation method in this research for three reasons:

- The abstract syntax for the UML is defined by a metamodel. Therefore, the metamodel can be used to control the impact of modifications on model elements. By controlling modifications, model transformations can be restricted such that the intended outcome of the design model does not change [Sunyé et al, 2002].
- The pre-defined transformations, expressed as UML metamodels, provide a mechanism to check conformance of UML models against the metamodel [France et al, 2004].
- UML is the de-facto standard for modeling languages [OMG 2004b; Rumbaugh et al., 2004].

To address the research objective, we use pattern-based model transformation. Pattern-based model transformation is the restructuring of a source model into a target model by instantiating the source model with a design pattern. This research specifies a controlled model evolution process that (1) constrains how transformations...
are carried out, and (2) checks the model-level transformation for conformance. Controlled model transformation results in UML metamodel conformance when the UML metamodel is specialized to include newly introduced metamodel elements (i.e., meta-classes and meta-associations) whose instances can be handled as if they were instances of their UML metaclass. Specifically, object-oriented design models expressed in UML are transformed into pattern-based design models by generating transformations on the metamodel.

There are several components developed for this research, as shown in Figure 1.2: (1) Specialize the UML metamodel; (2) Define an action language for pattern-based transformations; (3) Develop model-level transformation specifications; (4) Develop model-level pre- and postconditions; (5) Validate pre- and postconditions; and (6) verify structural conformance.

**Figure 1.2. Overview of Research**

**Specialize the UML Metamodel:** The UML metamodel is specialized by extending it to describe families of transformations on UML diagrams. Metamodels provide the foundation for the establishment of rigorous
transformation specifications to support controlled evolution of models. In the approach developed during this research, transformations are defined as an extension of the M₁ (model) and M₂ (metamodel) levels of the UML architecture. The source schema, transformation schema, and transformation constraints are specialized metamodel elements. The source schema specifies preconditions that must be satisfied to ensure conformance to the source model. The model elements within the source schema must exist in the UML design model (i.e., the source model) before the model can be transformed into a target model that includes a design pattern. The transformation schema specifies the new classes of model elements that are introduced and the existing classes of model elements that are removed by the transformation. The transformation constraint further constrains the basic structure defined by the transformation schema. It specifies (1) constraints on source and target model elements that cannot be expressed elsewhere in the specialized metamodel and (2) relationships that must hold between source and target model elements for a valid transformation.

**Action Language for Pattern-Based Transformations:** An action language specifies the operations that can manipulate UML models and defines the concrete syntax for the UML action semantics. The Pattern-Based Action Language (PBAL), a Java-like action language that specifies the constructs for the pattern-based transformation of UML models, is defined for this research.

**Model-Level Transformation Specification:** Model-level transformation specifications (i.e., programs) specify the operations required to restructure a source model into a target model. These operations add, delete and connect model elements as specified by the transformation pattern.

**Model-Level Preconditions and Postconditions:** The model-level transformation specifications can be expressed as individual actions. Pre- and postconditions expressed in the Object Constraint Language (OCL) are attached to each action such that the precondition specifies what elements must exist in the model before the action can be carried out and the postcondition specifies what elements have been added, deleted, or connected by the action. These pre- and post conditions allow the model-level transformations to be checked for conformance against the metamodel.

**Validation: Conformance checking** of model-level transformations against the metamodel provides one method of validation for this research. Composing the model-level pre- and postconditions into a single pre- and postcondition pair allows the model-level transformation specification to be validated formally. The graphical
specifications (i.e., transformation patterns) provide an informal technique to structurally verify model-level transformation against metamodel-level transformations.

Figure 1.3. Overview of Pattern-Based Model Transformation Approach

1.2.1 Pattern-Based Model Transformation

As previously stated, this research defines a specialization of the UML metamodel by describing families of model transformations. The model transformation family consists of transformation patterns which describe graphically how to introduce a design pattern into a UML class diagram at the metamodel level. The manual
A pattern-based transformation is defined at two different levels of the UML hierarchy: the metamodel-level (M2) and the model-level (M1). At the metamodel level, a transformation pattern developer specifies the transformation pattern. The transformation pattern developer then generates the transformation specification, which conforms to (i.e., is an instance of) the transformation schema and transformation constraint. The target pattern can be generated explicitly by adding and deleting the model elements specified by the transformation schema and applying the restrictions specified by the transformation constraints.

The transformation pattern and transformation specification are stored in a transformation repository for later use by an application developer.

At the model-level, an application developer desiring to introduce a design pattern into an existing UML design model, referred to as a source model, first needs to determine if the model is a valid source model. This determination is performed by manually inspecting the source model against the source schema with respect to the bindings defined by the source schema on the model elements. When a valid source model exists, the application developer generates a new UML design model by applying the transformation specification to the source model. In Figure 1.3, the inputs of a pattern-based transformation are the transformation specification and the source model, while the output is an improved UML design.

The transformation can be validated by verifying the model-level transformations against the metamodel, that is, comparing the model-level postconditions (PostModel) with the metamodel-level postconditions (PostMeta). The application developer validates the improved UML design by determining if it structurally conforms to the target pattern.

1.3 Structure of Dissertation

Chapter 2 presents the background concepts and related research. Chapter 3 discusses the concrete syntax for a pattern-based action language. Chapter 4 describes the approach defined in this research to pattern-based model transformation. In Chapters 5 thru 7, the metamodel-level transformation definitions are described for creational, behavioral, and structural design patterns, respectively. Chapter 8 summarizes the work presented in this dissertation, discusses the major contributions, and describes future work.
Chapter 2
Background and Related Research

The foundational concepts of this research are the UML, OCL, Role-Based Modeling, and design patterns. Sections 2.1 – 2.4 provide an overview of these concepts. Section 2.5 discusses the evolution of software models. Section 2.6 presents other work that relates to this research. Section 2.7 summarizes the chapter.

2.1 Unified Modeling Language (UML)

The Object Management Group (OMG), formed in 1989, is a consortium that sets standards for object oriented computing and management across different platforms and environments. The mission of the OMG is to establish industry guidelines and detailed object management specifications providing a common framework for application development. Two well-known OMG standards are Common Object Request Broker Architecture (CORBA) [see http://www.omg.org/corba; Bolton & Walshe, 2001] and UML [see http://www.omg.org/uml; Rumbaugh et al, 2004].

The UML [Rumbaugh et al, 2004] is a general-purpose visual modeling notation used to model software-intensive object-oriented systems. During the early 1990’s, there were numerous methodologies providing notations for the design of object-oriented systems, each having strengths and weaknesses. The three most popular were the Booch methodology developed by Grady Booch, Object Modeling Technique (OMT), and Object-Oriented Software Engineering (OOSE). The Booch methodology was strong in design and weak in analysis, OMT was stronger in analysis but weaker in design, and OOSE presented a strong approach for behavior analysis but was weak in the other areas [Quatrani 2002].

The desire to provide a uniform method of software modeling yielded the UML resulting from combining the strengths of Booch, OOSE and OMT with the best ideas of several other methods as illustrated in Figure 2.1. The UML has become the generally accepted way to model and design software systems. UML consist of a collection of diagrams (the notation) which are described using graphical and textual features, along with an informal description of the semantics that defines the meaning of the diagrams and their features.

Figure 2.2 shows the logical organization of the diagram types, each expressing a different UML property. UML diagrams are classified as Structural or Behavioral [OMG 2003a]. As described in [OMG 2003a]:
Figure 2.1. Inputs into the Development of UML [Quatrani 2002]

Structure diagrams show the static structure of the objects in a system. That is, they depict those elements in a specification that are irrespective of time. The elements in a structure diagram represent the meaningful concepts of an application, and may include abstract, real-world and implementation concepts. For example, a structure diagram for an airline reservation system might include classifiers that represent seat assignment algorithms, tickets, and a credit authorization service. Structure diagrams do not show the details of dynamic behavior, which are illustrated by behavioral diagrams. However, they may show relationships to the behaviors of the classifiers exhibited in the structure diagrams.

Behavior diagrams show the dynamic behavior of the objects in a system, including their methods, collaborations, activities and state histories. The dynamic behavior of a system can be described as a series of changes to the system over time.

The following diagrams are supported by the UML [Rumbaugh et al., 2004, Ambler 2004]:

- A **Class Diagram** is a graphical presentation that shows a collection of static model elements such as classes and types, their contents, and their relationships.

- A **Composite Structure Diagram** depicts the internal structure of a classifier (such as a class, component, or use case) by showing elements that work together. The composite diagram is similar to a class diagram, but it shows parts and connectors.
- A **Component Diagram** depicts the internal structure (i.e., the software units), interfaces and relationships of an application that will be used to build the system.
- An **Object Diagram** shows real-life instances of a class diagram and their relationships at a point in time.
- A **Package Diagram** shows how model elements and diagrams are organized into packages to handle the complexity of large models.
- A **Deployment Diagram** describes how components of an application are deployed across the implementation architecture of a system.
- An **Activity Diagram** is used to describe behavior represented as a sequential flow of activities that describe concepts.
- A **Use Case** describes required behavior of a system as it appears to a user.
- A **State Machine** (i.e., state chart) captures the lifecycle of an object by indicating the possible states of an object and the transitions between states.
- A **Sequence Diagram** describes how instances of objects interact within a system to accomplish a task. A sequence diagram focuses on the message interchange between lifelines.
- A **Communication Diagram** shows the relationship among objects in an interaction through an architecture view of the internal structure, focusing on how objects interact with each other. It shows the exchange of messages and the relationship between objects.
• An Interaction Overview Diagram is a variant of an activity diagram that gives an overview of the activity flow of control within a system.

• A Timing Diagram depicts a change in the state over time of an instance of a classifier and the interactions between classifiers in response to the occurrence of external events.

Not all diagrams are needed to model an application. Any mixture of diagrams can be used, depending on the aspects of the final system. Since this research focuses on the transformation of models represented as class diagrams, the UML class diagram will be described further in Section 2.1.1.

![UML Class Diagram](image)

Figure 2.3. UML ClassDiagram

2.1.1 UML Class Diagram

The UML class diagram shown in Figure 2.3 describes classifiers (e.g., classes and interfaces) and relationships between classifiers (e.g., association, generalization, and dependency). A class describes a family of objects that share common attributes and operations. Associations between classes specify links between instances of classifiers. The class diagram in Figure 2.3 shows five classes (Person, Address, Faculty, Staff, and Student) and their relationship with Address. The link between Person and Address is an association.

2.1.2 UML Architecture

UML is structured as a four-layer hierarchical infrastructure (e.g., see Figure 2.4 [OMG 2003b]). This type of structure specifies a separation of concerns across different layers of abstraction where each layer represents a different functionality. The four-layer hierarchy defines the logical architecture of any UML-based system. The structure consists of the following levels, referred to as $M_n$, where “M” stands for model and “n” represents the layer within the hierarchy:
- M₀, the first level, is at the bottom of the hierarchy. This level holds the executable entities (i.e., runtime instances of model elements) that occur during the execution of the code generated from the model [PJMS 2001].

- M₁ is the model level representing the model as the designer conceives it. The M₁ level allows users to model a wide variety of problem domains.

- The M₂ layer is the metamodel level. It defines the characteristics of a syntactically correct model. The primary responsibility of the M₂ layer is to define a language for specifying models.

- The meta-metamodel level, M₃, is the foundation for the metamodelling architecture. It provides the definition of the metamodel syntax, which defines the language for specifying a metamodel.

The UML meta-metamodel level conforms to the OMG’s Meta-Object Facility (MOF) [OMG 2003c] standard. The MOF, a technology developed to assist and standardize the handling of abstract data, defines an abstract language and framework for specifying, constructing and managing metamodels by “providing a set of generic domain-independent concepts and relations” [Breton & Bézivin, 2001]. A metamodel is an instance of a meta-metamodel, meaning that every element of the metamodel is an instance of an element in the meta-metamodel. The objects of the model are instances of the classes of the metamodel. The responsibility of the layer Mₙ defines the language that describes the objects of the layer Mₙ₋₁.
2.1.3 The UML Metamodel

The UML specification is defined using a metamodeling approach, where a metamodel is used to specify the model that defines UML. UML has a uniform and precise description of its syntax in the form of a metamodel. The objects of the model are instances of the classes of the metamodel. Thus, a metamodel is a model of a model. The UML metamodel specifies valid forms of syntactically well-formed UML models. The metamodel consist of a class diagram and a set of well-formedness rules defining the abstract syntax and informal descriptions of semantics.

The metamodel for the class diagram consists of classes whose instances are UML model elements as shown in Figure 2.5 [OMG 2003a]. The diagram shows (1) the relationship between UML classifiers and their properties (e.g., attributes) and operations, (2) the generalization relationship between classifiers, (3) the relationship between classifiers and associations, and (4) the relationship between operations and the actions they define. There are many types of actions. One type is CreateObjectAction, whose instances are actions that create instances of the Classifier with which they are associated. The following describes the UML metamodel elements shown in Figure 2.5 [OMG 2003a; Rumbaugh et al., 2004].

- A Classifier describes a set of instances that have behavioral and structural features in common.
• A Class is a kind of classifier describing a set of objects that share the same specification of features, constraints, and semantics. The purpose of a class is to specify a classification of objects and the features that characterize the structure and behavior of those objects.

• Generalization specifies the relationship between a general classifiers (superclasses) and more specific classifiers (subclasses). Each instance of the general classifier is an indirect instance of the general classifier. Thus, the subclass inherits the features of the superclass.

• A Feature declares the structural or behavioral characteristic of classifiers. A feature can be either a property (structural feature) or an operation (behavioral feature). A StructuralFeature declares the structural aspects (e.g., attribute or association) of the classifier instances. A BehavioralFeature describes dynamic behavior of one or more classifiers.

• A Property is a structural feature of a classifier. When a property is owned by a class, it represents an attribute which describes the values that instances of classifier can hold. A property relates an instance of the class to a value (or set of values) of the type of attribute. When a property is owned by an association, the property represents a non-navigable end of the association, and the type of the property is the type of the end of the association. In Figure 2.5, the multiplicity (2..*) at the member end of Property specifies that an association must have at least two ends (properties).

• An Association specifies the link between typed instances.

• An Operation is a behavioral feature specifying the name, type, parameters, and constraints for invoking an associated behavior. Instances of Operation represent operations of a class. An operation can consist of activities (instance of Activity).

• An Activity is a specification of behavior expressed as the flow of execution via a sequence of subordinate units whose primitive elements are individual actions. An activity consist of actions (an instance of Action), where an action is a fundamental unit of a behavioral specification that represents some transformation or processing in the modeled system. The execution of an action corresponds to the execution of a particular action within an activity. Actions are contained in activities, which provide control and data sequencing constraints among actions. An instance of CreateObjectAction is an action that creates an object that is an instance of a classifier. The created object conforms statically to the specified classifier.
2.2 Object-Constraint Language (OCL)

UML provides a standard notation for all aspects of software modeling; however, it can only express details that can be represented graphically. OCL [Warmer & Kleppe, 2003] is a textual, declarative specification language used to express properties (i.e., constraints) on UML models that cannot be represented graphically in the diagrams. OCL supports UML by providing the ability to navigate through models and to express constraints on model elements using invariants, preconditions and postconditions [Ritchers & Gogolla, 2002; Warmer & Kleppe, 2003]. An invariant is a static structure constraint specifying conditions that must always evaluate to true at any moment in time. Preconditions specify the conditions that must evaluate to true when the operation begins execution. A postcondition specifies conditions that must evaluate to true at the exact moment execution ends. The OCL expresses UML well formedness rules that assist in the validation of the UML metamodel abstract syntax and the identification of errors on the UML metamodel.

2.3 Role-Based Metamodeling Language (RBML)

One of the main objectives of this research is to specify how to rigorously introduce design patterns into existing UML models. This objective requires validation and verification of conformance between the model at the M1-level and the metamodel representation of that model. By specifying syntactic and semantic constraints of the metamodel, conformance of the model instance can be ensured. In order to achieve this conformance, precise metamodeling techniques are needed. The Role-Based Modeling Language (RBML) [France et al., 2002b; France et al., 2004; Kim et al., 2004] is a modeling notation for rigorously specifying families of UML models to characterize valid UML diagrams. RBML extends the metamodel (i.e., specializing the metamodel) to define a family of UML diagrams for modeling the structural and behavioral properties of an application.

An RMBL specification is a structure of role models (hereafter referred to as roles). RBML roles, the core of RBML, specify structural and behavioral properties a UML model element must have if it is to be part of a solution model. The notion of roles is defined at the metamodel level. Each role is associated with a UML metamodel class (e.g., Class, Generalization) called its base. Roles define a constrained form of the UML metamodel that specifies families of models at the M1 level by constraining the metamodel defined at the M2 level specifying solutions. Adding constraints to the UML metamodel produces a specialized metamodel that defines a subset of valid forms of UML models. The properties defined in a role determine a subset of the role’s base instances, elements at level M1. For example, a role with the Class base determines a subset of class constructs [Song et al., 2002].
2.3.1 Static Role Model

A Static Role Model (SRM) is a characterization of a family of UML static structural models that characterizes a conforming class diagram. A UML model is said to conform if the class diagram conforms to an SRM. This implies that a model conforms to a specification if it satisfies the constraints defined in the specialized metamodel. An SRM defines a specialization of the UML class diagram metamodel; thus, it is expressed as a variant of the class diagram.

An SRM characterizes a family of UML static structural models. It consists of classifier and relationship roles, where a classifier role is connected to other classifier roles via relationship roles. The base of a role is a metamodel class whose instances are elements of UML static models. A classifier role has a base that is a subtype of Classifier (e.g., Class, Interface) and a relationship role has a base that is a subtype of Relationship (e.g., Association, Dependency, and Generalization). The relationship between a classifier and an association is illustrated in the example SRM structure given in Figure 2.6.

A classifier role defines properties that classifier constructs must have if they are to play the role, and a relationship role defines properties that UML relationship constructs must have if they are to play the role. A
classifier role can be associated with feature roles that specify behavioral and structural features (e.g., attributes, operations) of the conforming classifier.

The structure of a classifier role is illustrated in Figure 2.6. The classifier role is divided into two compartments [France et al., 2004, Kim 2004]. The top compartment consists of three parts. The first part is a label specified in the form of **Base Role**, where **Base** indicates the name of a metamodel class. The second part is a role name declaration of the form |**RoleName**, where the symbol “|” indicates that the string (e.g., RoleName) that follows represents the name of the role. The third part of the top compartment is a realization multiplicity specifying the number of conforming classifiers that can exist for the role. The bottom compartment consists of feature roles that specify features (e.g., attributes and operations) that are associated with the conforming classifier. The RMBL specification defines two types of feature roles: (1) StructuralFeature roles specify a family of classifier structural features (either an attribute or a query) and (2) BehavioralFeature roles specify a family of classifier operations.

The association (i.e., relationship) role shown in Figure 2.6 indicates how associations are expressed between two class roles. A role can be associated with another role, indicating that realizations of the roles are associated in a manner consistent with how the bases of the roles are associated in the UML metamodel.

A class diagram that conforms to the SRM must have at least one class that conforms to the Class role. When a structural or behavioral feature role is specified for a class, the class must have a structural feature that plays the StructuralFeature role or an operation that plays the BehavioralFeature role. For each feature role, a realization multiplicity specifies the number of features that can play the feature role in a conforming classifier role. A feature role realization multiplicity with a lower bound of 0 (e.g., *) indicates that the feature may or may not be present in a conforming classifier. A relationship role is represented by a syntactic variant of the UML association symbol. The ends of an association role symbol represent association-end roles. The example SRM structure illustrated in Figure 2.6(b) shows Subject and Observer class roles that conform to the Classifier role shown in Figure 2.6(a). The |**Subject** and |**Observer** roles are connected to each other using the association role Observes.

A role can also be constrained by metamodel-level constraints and constraint templates. These constraints are defined separately from the SRM to avoid cluttering the diagrams. Metamodel-level constraints expressed in OCL specify well-formedness rules for the elements characterized by the role. Metamodel level constraints, like UML well-formedness rules, determine the form of model construct that can play a role. Constraint templates are used to restrict the form of constraints placed on the conforming UML models by specifying semantic properties associated
with features that conform to feature roles [Kim et al., 2004]. Two types of constraint templates can be specified with RBML specifications. An operation template restricts the form of pre- and post-conditions associated with operations that conform to behavioral feature roles. An invariant template specifies invariant properties in a UML model. Constraint templates are also associated with structural feature roles to obtain constraints associated with conforming structural features.

### 2.3.2 RBML Conformance

A UML model that consists of a class diagram is said to conform to a role model if the class conforms to the specialized UML metamodel determined by the SRM [Kim et al., 2004]. A specialized UML metamodel consist of (1) the UML metamodel class diagram with specialized classes defined by roles, and (2) a set of well-formedness rules and constraint templates defined by the syntax for conforming class diagrams [France et al., 2004]. The metamodel and the well-formedness rules define the syntax for a conforming class diagrams. The constraint templates are used to obtain operations that must evaluate to true to establish that a model conforms to a role model.

Checking conformance of a model against a role model involves checking that the static structural diagram conforms to the SRM. A class diagram is said to conform to a role model if [France et al., 2004]:

![Structurally Conforming Class Diagram](image)
1. Model elements are bound to the roles they intended to play.

2. The number of classifiers bound to a classifier role satisfies the realization multiplicities associated with the role.

3. Structural conformance of classifiers to their bound roles is achieved when the classifier satisfies the metamodel-level constraints associated with the classifier role, the features bound to feature roles in the classifier role satisfy the realization multiplicities of the feature roles, and the mandatory feature roles have features bound to them.

4. Relationship conformance of relationships is achieved when the relationships bound to relationship roles satisfy metamodel-level constraints associated with the roles, and the relationships have ends attached to classifiers that conform to the roles at the ends of the relationship roles.

An example of conformance with respect to bindings is shown in Figure 2.7 [Kim 2004]. The bindings are indicated by the dashed lines between the class diagram and the SRM. The SRM specification show a class diagrams in which |Subject classes can have exactly one structural feature that can be monitored, and the |Subject can be a part of only one association connected to the |Observer [Kim 2004]. The dashed lines indicate the class Kiln is bound to the |Subject role, the class TempOps is bound to the |Observer role, and the association obsTemp is bound to the |Observer association role. The diagram shows a class conforming to the |Subject role must have one structural feature that plays the role of |SubjectState and one behavioral feature that plays the role of |Attach. A class conforming to the |Observer role must have one structural feature that plays the role of |ObserverState and one behavioral feature that plays the role of |Update. The conforming diagram, Figure 2.7(a), indicates that in the class Kiln, the structural feature temp is bound to the |SubjectState role and the behavioral feature AttachTempObs is bound to the |Attach role. In the class TempOps, the structural feature curTemp is bound to the |ObserverState role and the behavioral feature UpdateTemp is bound to the |UpdateAttach role. The class Kiln describes kiln objects whose temperatures are monitored by TempOps objects.

2.4 Design Patterns

The concept of design patterns was influenced by the work of an architect and urban planner named Christopher Alexander who introduced the word “pattern” to refer to recurring designs in building architecture [Alexander, 1977]. He defined a pattern as a proven “solution to a problem in context” [Alexander, 1977].
Each pattern describes a problem which occurs over and over again in our environment and then describes
the core of the solution to that problem, in such a way that you can use this solution a million times over,
without ever doing it the same way twice. [Alexander 1977]

The ideas Alexander presented in his work influenced researchers to develop patterns for software. In the early
1990’s, Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson collaborated on the book Design Patterns:
Elements of Reusable Object-Oriented Software [Gamma et al., 1994] which is the most influential design patterns book today.

Design patterns [Gamma et al., 1994] describe a family of proven solutions to common recurring design
problem. These solutions are based upon the experience software designers have gained when faced with recurring
problems. Introducing a design pattern into an existing design model is a form of perfective evolution. Since design
patterns provide reusable solutions, the design quality is enhanced.

Software design patterns [Gamma et al., 1994; Buschmann et al., 1996] capture design experiences in the form
of reusable solutions that address recurring problems detected during software development. Each design pattern
focuses on a particular design problem or issue. Generally, design patterns “can be used to reduce the effort and
time taken to develop good design models” [Kim et al., 2004]. There are several benefits of design patterns to the
software community [Shalloway & Trott, 2004]:

- Design patterns offer reusable solutions the common recurring problems.
- Design patterns make communication between designers more efficient by using common terminology.
- Patterns give a high-level of perspective on the problem and on the process of design and object-
  orientation.

In [Gamma et al., 1994], twenty-three patterns were given and categorized as creational, structural, and
behavioral. Creational patterns provide guidance on how to create objects when their creation requires deciding
which class to instantiate or to which objects an object will delegate responsibility. Structural patterns deal with the
composing different types of classes or objects with each other to form larger structures that realize new
functionality. Behavioral patterns are used to organize, manage and combine behavior through the assignment of
responsibilities between objects. Behavioral patterns characterize the way in which classes or objects interact to
distribute responsibility [Gamma et al., 1994; Shalloway & Trott, 2004].
2.5 Model Evolution through the Transformation of Models

Evolution is the “process of continuous change from a lower, simpler, or worse [state] to a higher, more complex or better state” [Merriam-Webster 2004]. This definition is applicable to anything that changes state over time. During the life of a computer system, maintenance and requirement issues sometimes require changes to the software of that system. To properly document those changes, the models that represent the conceptual view of the software must also change. This changing of models is called model evolution. Model evolution is the process in which changes are successively applied to an existing software design to produce a newly modified system which meets various design objectives. Model transformation is a type of model evolution that restructures software design models by acting as an interface between the source and target models.

Model transformations exist in both the vertical or horizontal dimensions [France & Bieman, 2001]. Vertical transformations occur when a source model is transformed into a target model at different levels of abstraction. Examples of vertical transformations include model refinement and the realization of a model into a target programming language. A horizontal transformation occurs when a source model is transformed to a target model that is at the same level of abstraction as the source model. One approach to horizontal transformations is model refactoring. Model refactoring improves specific quality attributes of a model to meet design objectives. Model refactoring occurs when a software model is changed to enhance specific design qualities while preserving some properties of a model. A model refactoring should only “affect a previously chosen subset of the source model [Porres 2003]”. The transformed model represents an improvement in how a desired result is accomplished. Transformations that improve quality attributes result from a desire to meet design goals, address deficiencies uncovered by evaluations, or explore alternative decision paths [France & Bieman, 2001]. This type of horizontal transformation is carried out to support perfective model evolution.

Refactoring techniques that define software transformations to restructure a software system have been proven to provide solution to problems caused by maintenance and evolution at the code-level (e.g., see [Opdyke 1992]; [Roberts 1999]; [Cinnéide, 2000]). Similar solutions are needed at the model level.

The OMG recognized the importance of models to the software development process. In response, they are promoting the Model-Driven Architecture (MDA) initiative as an approach to use models in software development. The vision of MDA is to define an approach to IT (Information Technology) system specification that separates the specification of system functionality from the implementation of that functionality on a specific technology platform.
In other words, MDA aims to allow developers to create systems entirely with models. The vision is for systems composed of many small, manageable models rather than a single model. The focus of MDA is to treat models as the primary artifacts of development, thus providing support for model evolution.

The concept “model-driven” implies that the usage of models will direct the understanding, design, and construction of systems where the software development process is controlled by software modeling. Models represent the software design that captures the properties of the design requirements without specifying details of the intended platform [Karsai & Agrawal, 2003]. The MDA initiative was formulated to define an approach to software development based on the modeling and mapping of models into implementations. This is achieved through well-defined techniques for model transformation.

In response to the need to provide a well-established foundation for defining transformations, the OMG issued the Query / Views / Transformations (QVT) Request for Proposals (RFP) [OMG 2002]. The QVT standardization effort focuses on the technical and conceptual management of model evolution [Guelfi & Perroin, 2004]. QVT addresses the need for a precise definition of model transformations expressed in terms of the relationship between a source metamodel and a target metamodel. In addition, QVT must express a way to query models and create views of models.

To provide support for MDA and QVT, transformations need to be defined such that they can be applied across different aspects of software systems. The transformation describes the relationship between a source metamodel and a target metamodel that generates a target model instance from a source model instance. A simplified description of MDA and QVT is that MDA provides the guidelines needed to structure specifications expressed as models and the mappings between those models while QVT provides the standard means for expressing transformations of models.

2.6 Related Work on Design Pattern Transformation

Mél Ó Cinnéide [Cinnéide 2000] developed a method to automate the transformation of design patterns into existing code using transformation algorithms. In Cinnéide’s work, a “precursor” indicates where a transformation begins (i.e., the starting point) and the design pattern serves as the target of the transformation, such that the transformation algorithm stops executing when the design pattern has been applied to the code. Design pattern transformations were decomposed into a sequence of mini-patterns that represent the recurrent elements within the design pattern catalogue. Each mini-pattern has a corresponding minitransformation that expresses the operations
required to restructure the code. Minitransformations are reusable operations that specified the pre- and postconditions and the algorithmic description of the transformation. Cinnéide’s approach was entirely a source-to-source transformation process; however, the definition of the precursor and minitransformations provide key aspects to any transformation technique.

Maplesden et al [Maplesden et al, 2001] describes a visual model language called Design Pattern Modeling Language (DPML) that provides a notation for specifying design pattern solutions. In DPML, a pattern solution is instantiated to model instances of design patterns as are part of UML object models. The participants when instantiated are linked to objects in the object model. This instantiation occurs during the design of a software system rather than during code implementation. This approach does not describe how to verify conformance of the transformation rule to the object model nor does it describe a mechanism for specifying constraints on pattern participants.

Sunyé et al [Sunyé et al, 2000] developed a metaprogramming approach that uses UML collaborations combined with OCL to allow designers to define and apply variants of known patterns into UML models. Metaprogramming applies a sequence of transformation steps to a starting point in an initial model to produce a final model that has a pattern occurrence as represented by a collaboration occurrence. In this approach, the authors did not specify how to determine a “precursor” that can be uniquely applied for only one specific pattern.

In [Albin-Amiot & Gueheneuc, 2001], a metamodel is used to describe structural and behavioral aspects of design patterns for automatic code generation and design pattern detection. To instantiate a pattern, the metamodel is specialized to add structural and behavioral elements. The specialized metamodel is instantiated to produce an abstract model which is instantiated into a concrete model. The concrete model represents the pattern applied to fit the user’s requirements. We found the metamodel to be complicated and the representation of the instantiation process difficult to interpret. The structure and properties of a pattern are defined at the model level, but the representation of behavior properties are not given at the model level.

In [Sunyé et al, 2002], Action Semantics for UML are used to manipulate model elements (i.e., transform models). They extended previous work on design pattern applications in UML [Sunyé et al, 2000] by illustrating how the UML action semantics, which manipulated model elements, can be combined with OCL pre- and postconditions to specify the transformation on models. They use the OCL to specify their surface language, which in some cases is complex and difficult to understand.
The pattern-based model refactoring approach discussed in [France et al, 2003b] gives an informal description of how role models can be used to incorporate precise specifications of design patterns into class diagrams. The [France et al, 2003b] approach uses the precise pattern specification developed by [Kim et al, 2004] to apply both patterns and transformation rules. This work informally defines how to apply a design pattern to a class diagram. It does not specify how to determine the conditions that allow the pattern to be applied to the source model nor does it specify a formal transformation language. The transformation rules are expressed as generically defined steps to applying a design pattern.

2.7 Summary

Currently, design patterns are described using code implementations in a natural language, making them difficult to understand and interpret. Software modelers are required to think about how to implement design patterns in models during the design process. If design patterns can be represented as well-defined reusable models, then they can easily be introduced into design models by transforming a source model into a refined target model instantiated with a design pattern. However, effective management of the transformation of models requires the models to be restructured such that they can be used as points against which the transformations can be checked for conformance.

This research uses an approach to model-level design pattern transformations that incorporates Cinnéide’s ideas of defining a “precursor” and using mini-transformation to express operations that are reusable throughout the process. Each transformation has pre- and postconditions to specify constraints on pattern participants. These pre- and postconditions ensure the application for the pattern is applied correctly and only affects elements in the model that are part of the transformation. Unlike Maplesden’s work, this research specifies a method to validate conformance between the class diagrams produced and the transformation applied.

Pattern-based model transformation goes further than the theoretical background presented in [France et al., 2003b]. We provide a formal method of defining the application of design patterns into UML class diagrams by specifying the transformation of behavioral and structural properties of a design pattern; provide a transformation language that uses the abstract syntax of the UML action semantics and OCL expression; and provide a mechanism to validate the transformations.

To determine whether a pattern has been applied appropriately and to ensure that the transformations can be validated, the need exists for a rigorous approach for the introduction of design patterns into UML models. The
RBML notation for Static Role Models provides a rigorous specification of UML class diagrams and a method to ensure structural conformance. We use the RBML notation to extend the UML metamodel to define transformations on UML model elements. Specifying UML models as static role models enables the validation of conformance between the M1 level and M2 level. The research documented in this dissertation focuses on instantiating an existing design model with a design pattern.
Chapter 3
Action Language for Pattern-Based Model Transformations

The ability to introduce a design pattern into an existing UML model depends upon the capability to add, delete, and connect model elements within the model. This capability requires a unique way to restructure the design of a model at the metamodel-level. The aim of UML Action Semantics (AS) is to provide the UML with a mechanism for specifying actions in a software-independent manner [OMG 2001]. AS provides the abstract syntax to a minimal set of actions for expressing behavior, but it is not a language for the specification of actions. To address the need to specify transformations on models, an action language called Pattern-Based Action Language (PBAL) was developed.

Section 3.1 discusses the Action Semantics for the UML and the abstract syntax used for this research. Section 3.2 describes the PBAL concrete syntax developed by this research. Section 3.3 summarizes the action language for the transformation of models.

3.1 UML Action Semantics

The UML Action Semantics [OMG 2001, OMG 2003a] provides a rigorous way to specify the behavior of objects by providing a precisely defined abstract syntax for the metamodel and model levels to specify actions on UML models. The use of action semantics enables the analysis, verification, test, and code-generation of models. Action Semantics for the UML are used to specify imperative logic in a form that can be automatically mapped to different programming languages [Czarnecki & Helsen, 2003].

An action is a “fundamental unit of behavior specification that represents the transformation or processing in the modeled system” [OMG 2003b]. As a core package for UML, the Action package defines several kinds of actions, including Read, Write and Link. The UML Superstructure specification [OMG 2003a] describes Read and Write actions as follows:

Objects can be created and destroyed; structural features and variables have values; links can be created and destroyed, and can reference values through their ends; all of which are available to actions. Read actions get values, while write actions modify values and create and destroy objects and links. Read and write actions share the structures for identifying the structural features, links, and variables they access. … Read actions do not modify the values they access, while write actions have only limited effect.

Object actions create and destroy objects. Structural feature actions support the reading and writing of structural features. The abstract metaclass StructuralFeatureAction statically specifies the structural feature being accessed. … Association actions operate on associations and links. … Variable actions support the
reading and writing of variables. The abstract metaclass VariableAction statically specifies the variable being accessed.

The actions used in this research are [Rumbaugh et al, 2004]:

1. **CreateObjectAction.** An action that creates an object that conforms to a specified Role.

2. **DestroyObjectAction.** An action that destroys an instance of the object that conforms to a specified Role. This action also destroys any links associated with the conforming object.

3. **ReadExtentAction.** An action that retrieves all current instances of a specified object. The extent of a specified object “is the set of all instances of a classifier that exist at any one time [OMG 2003a]”.

4. **CreateLinkAction.** An action that creates a link and links two specified objects.

5. **DestroyLinkAction.** An action that destroys a link between specified classes.

### 3.2 Pattern-Based Action Language (PBAL)

This approach to pattern-based model transformation involves the manipulation of models by adding, deleting, and connecting model elements. To accommodate those actions, we developed PBAL. PBAL is an action language which provides the concrete syntax for the manipulation of model elements. The PBAL is a Java-like action language that defines constructs explicitly for the transformation of models. This language is different from JAL (Java-like Action Language) [Dinh-Trong et al., 2004], whose constructs were defined for testing UML models, and J language [Softeam 1999] which has a Java-like syntax that is used to realize all forms of UML model.

The general syntax of PBAL is:

- An operation consists of either a simple statement that has access to model elements contained within a Class Diagram (such as create a link between two classes) or a sequential logic structure (loop, condition).
- Each PBAL statement is terminated by a semi-colon (;) except for loop and condition control constructs.
- PBAL supports both inline and block comments. Inline comments follow the double slash (//), and block comments are inserted between the /* and */ delimiters.

PBAL supports the Integer (**integer**), Boolean (**Boolean**), and String (**String**) primitive data types. Variables are declared with types before they are used with a unique variable identifier. A variable can be either an object handle or a primitive data type. An object handle is the local variable that refers to a single instance (or set of instances) of |Role.
The while and if statements are the two control constructs supported by PBAL. The while construct is used to sequentially execute the actions contained within the while loop as long as the condition (boolean_expression) is evaluated as TRUE. In a while construct, the condition is tested at the start of the loop. The syntax of the while construct is:

```
while (boolean_expression) {
    statement_sequence;  // only executes when boolean_expression evaluates to True
}
```

The condition boolean_expression evaluates to TRUE or FALSE and the statement_sequence may contain one or more PBAL statements.

The if statement is used to control the flow of execution through one of two or more paths depending on the result of a logical test. The else, which is optional, may be used when the condition of the if statement evaluates to False.

```
if (boolean_expression) {
    statement_sequence; // executed if boolean_expression is TRUE
} else
    statement_sequence; // executed if boolean_expression is FALSE
```

### 3.3 PBAL Constructs

PBAL constructs operate on metamodel elements to perform pattern-based model transformations. The descriptions of the concrete syntax required to add, delete, and connect objects are given in Sections 3.3.1 - 3.3.6.

#### 3.3.1 Create Instances of Objects

The concrete syntax for creating instances of objects is:

```
objHandle ::= _create_instance(Role);
```

The handle, objHandle, is the returned reference to the newly create instance of |Role. This command creates instances of |Role and then returns a reference of the instance to objHandle. The syntax _create_instance() specifies the language construct for the UML action CreateObjectAction.

#### 3.3.2 Destroy Instances of Objects

The concrete syntax for deleting instances of objects is:

```
_destroy_instance(objHandle);
```

This construct removes the instance of the object referenced by objHandle. When an instance specified by handle is removed, it is no longer available to the domain in which it was defined. The construct also deletes any links
connected to the objHandle. The syntax _destroy_instance() specifies the language construct for the UML action DestroyObjectAction.

### 3.3.3 Create Link

The create link construct creates a link between two model elements. This construct specifies the language construct for CreateLinkAction. The concrete syntax is:

```plaintext
_create_link(objHandle1, objHandle2);
```

The handles, objHandle1 and objHandle2, are references to instances of the model elements that are connected together. The end of the link connected to objHandle1 is the source of the association and the link end connected to objHandle2 end is the target.

### 3.3.4 Destroy Link

The destroy link construct deletes the link that exist between two model elements. The destroy link construct has two definitions for its concrete syntax. The first definition is:

```plaintext
_destroy_link(objHandle1, objHandle2);
```

The handles, objHandle1 and objHandle2, are references to instances of the model elements which are connected to the link to be deleted.

The other definition deletes all references to the |AssociationRole specified by the association rolename. The concrete syntax is defined as:

```plaintext
_destroy_link(|AssociationRole);
```

The parameter |AssociationRole specifies the role that is played by the link to be deleted. When an instance of a AssociationRole is removed, the association is no longer available to the domain in which it was defined. The syntax _destroy_link() specifies the language construct for the UML action DestroyLinkAction.

### 3.3.5 Retrieve Instances of Classifier Objects

The _retrieve_ClassInstances() operation retrieves all instances of the specified |Role that exist at any one time and assigns a reference to the instances to objHandle. The concrete syntax for retrieving instances of classifier objects is:

```plaintext
objHandle ::= _retrieve_ClassInstances(|Role);
```

The objHandle is the returned reference to the set of instances of the specified |Role. The _retrieve_ClassInstances() operation defines the language construct for ReadExtentAction.
3.3.6 Retrieve Instances of Operations

The `_retrieve_OperationInstances()` operation retrieves all instances of a `|BehavioralFeatureRole` owned by instances referenced by `objHandle` that exist at any time and assigns the reference to `operHandle`. The action syntax for retrieving instances of an operation is:

\[
\text{operHandle ::= } _\text{retrieve}_\text{OperationInstances}(\text{objHandle, } |\text{BehavioralFeatureRole});
\]

The handle, `objHandle`, references instances of the `|Role` that owns the operation. `|BehavioralFeatureRole` refers to the operation owned by the `|Role` reference by `objHandle`. The handle, `operHandle`, is the returned reference to the set of operation instances specified by `|BehavioralFeatureRole`. The construct, `_retrieve_OperationInstances()`, is a PBAL defined syntax that extends ReadExtentAction to perform actions on operations.

3.4 Summary

UML action semantics provide an abstract syntax for the manipulation of models. PBAL defines the concrete syntax for actions specifically tailored for the transformation of model elements. Table 3.1 summarizes the PBAL concrete syntax.

<table>
<thead>
<tr>
<th>Action</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Object</td>
<td><code>objHandle ::= _create_instance(Role);</code></td>
</tr>
<tr>
<td>Destroy Object</td>
<td><code>_destroy_instance(objHandle);</code></td>
</tr>
<tr>
<td>Create Link</td>
<td><code>_create_link(objHandle1, objHandle2);</code></td>
</tr>
<tr>
<td>Destroy Link</td>
<td><code>_destroy_link(objHandle1, objHandle2);</code></td>
</tr>
<tr>
<td>Retrieve Object Instance</td>
<td><code>objHandle ::= _retrieve_ClassInstances(Role);</code></td>
</tr>
</tbody>
</table>
| Retrieve Operation Instance | `operHandle ::= _retrieve_OperationInstances( objHandle,  
                              | `|BehavioralFeatureRole);`             |
Chapter 4
Controlled Model Evolution

Uncontrolled model transformation can produce designs with faulty realizations of patterns or designs with convoluted pattern realizations that are difficult to evolve and analyze. Controlled model transformation is accomplished by specifying metamodels which constrain how the transformations are carried out and act as points against which the model transformations can be checked for conformance. The approach specified in this dissertation describes a metamodeling technique to specify families of transformations on UML class diagrams. Therefore, metamodels are used to describe the structure and the relationships that must exist between elements of the source and target models. Metamodel-based model transformations support the rigorous and systematic application of reusable transformations.

Section 4.1 provides an overview of pattern based model transformation. Section 4.2 discusses the approach defined for pattern-based model transformations. Section 4.3 discusses the pattern-based transformation specification (i.e., program). Section 4.4 discusses how the transformations are validated. Section 4.5 summarizes the pattern-based model transformation process.

4.1 Pattern-Based Model Transformation

It is well known that design patterns describe solutions to recurring design problems when the pattern applied to the problem results in a solution. To encourage the use of design patterns, this research developed a method, referred to as pattern-based model transformation, to improve a quality attribute of a design by introducing design patterns (i.e., reusable experiences) into UML class diagrams.

Pattern-based model transformation improves specific quality attributes of models when the transformation of source models are based upon reusable experiences and is applied to produce target models at the same level of abstraction as the source models. A pattern-based transformation is carried out when it is determined that a pattern can help improve how a design accomplishes its objectives. A design realizes a design pattern if the design possesses the properties specified in the pattern. Thus, this type of transformation occurs when an instantiated pattern, applied to a model, results in a new model reflecting the same solution [France & Bieman, 2001]. When this occurs, the design is a realization of the pattern and a pattern can be viewed as a loose characterization of its
realization [France et al., 2002a]. Developing a rigorous approach for pattern-based model transformation is the focus of this research.

4.2 A Metamodel Approach to Specifying Transformations

This research extends the UML superstructure, as shown in Figure 4.1, by defining a package, referred to as the Transformation package, which contains families of model transformations. Figure 4.1 shows the transformation package and its relationship to other packages in the UML. Defining transformations in the superstructure enhances the ability to expand transformations to support different user domains. The Transformation package is composed of specialized metamodel structures that realize a design pattern and metamodel-level restrictions which constrain the final state of the model after transformation.

The metamodel approach for specifying transformations defines model transformations as an extension of the M1 and M2 levels of the UML metamodel hierarchy. The diagram in Figure 4.2 illustrates a generic view of the model transformation approach utilized by this research. At the M2 level, the metamodel is specialized by extending it to support the metamodeling of transformations. The model level (M1) is extended such that it supports the representation of model transformations.

4.2.1 Metamodel-Level Transformations

The general idea of this approach is to take a specialized model as input then applies the transformation pattern to generate a transformed specialized model as output. Thus, transformations are defined at the M2 level of the UML hierarchy. Descriptions of the components of a transformation at the metamodel-level are given below.
Source Pattern: At the M2 level, the source pattern specifies preconditions that must exist in the source model such that there is conformance between the source pattern and the source model. That is, the metamodel elements in the source pattern represent preconditions that must be satisfied before the transformation can be applied. If the preconditions are not satisfied, the transformations cannot be applied to the source model.

Transformation Pattern: A transformation pattern consists of specialized metamodel elements that specify the structure of source and target metamodels (Source Pattern and Target Pattern, respectively) and the constraints on source and target metamodel elements. The transformation pattern is split into three parts: source schema, transformation schema, and transformation constraint. Each part is specified as a Static Role Model (SRM).

The source schema defines the model elements contained in the source pattern. The source schema is a specialized metamodel structure that specifies the structure of source models targeted by the transformation schema.
The classes shown in the source schema are classifier roles that characterize specializations (subclasses) of classes in the UML metamodel (e.g., see [France, et al., 2002] and [France et al., 2003a]). The source schema determines a specialized UML metamodel that characterizes source models. At least one class identified in the source schema must be a mandatory class, that is, it must have a multiplicity with a lower bound that is greater than 0. This ensures that there is at least one instance of a mandatory class in a model that conforms to the source schema. If there is no mandatory class in a source schema, then the metamodel determined by the source schema is the UML metamodel (i.e., the source schema characterizes all valid UML models). By making at least one source schema class mandatory, it is possible to distinguish the set of UML models that are targeted by transformations.

The transformation schema specifies the model elements that are created, deleted and connected during the transformation. The transformation schema indicates the new classes of model elements that are introduced by the transformation and the existing classes of model elements that are removed by the transformation. The classes shown in the transformation schema are all specializations of UML metamodel classes. Classifier and feature roles that are specified in the transformation schema but are not a part of the source schema represent structures that are created during transformation. Classifier and feature roles that are specified in the source schema but are removed during a transformation are indicated by their absence from the transformation schema.

The transformation constraint further constrains the basic structure defined by the transformation schema. It specifies restrictions on the source and target model elements that cannot be expressed in the source and transformation schemas and the relationships that must hold between target and source model elements. Constraints that cannot be defined graphically are expressed using the OCL.

**Target Pattern:** The target pattern is an explicit representation obtained by adding to (and removing from) the source schema metamodel elements specified in the transformation schema then validating the transformation by verifying the restrictions specified in the transformation constraints.

More than one transformation pattern may be needed to specify transformations based on a single pattern because it may not be convenient to capture all possible variations in the source models or transformation constraints in a single transformation pattern.

### 4.2.2 Model-Level Transformations

A transformation specification (e.g., T) at the M1 level takes a source model and transforms it into a target model. The transformation specification (T) is a member of the family of transformations characterized at the
metamodel level by the transformation pattern. A transformation specification (T) is said to conform to a transformation pattern if: (1) the source model (Source Model) is an instantiation of the source pattern (Source Pattern); (2) the target model (Target Model) is an instantiation of the target pattern (Target Pattern); (3) the relationship between elements of source and target models satisfies the constraints specified by the transformation pattern; and (4) the transformation specification (i.e., program) is defined such that it specifies the model elements added, deleted and connected and the restrictions placed on the model elements as required by the transformation pattern. A transformation specification conforms to a transformation pattern if it is an instance of the Transformation Pattern. A model that conforms to a source pattern is said to be an instance of the Source Pattern. Similarly, a model that conforms to a target pattern is said to be an instance of the Target Pattern.

4.3 Model-Level Transformation Specification

A model-level transformation specification is essentially a transformation program that specifies the operations on a source model required to produce a new, restructured model, referred to as the target model. The operations specified in the transformation specification add, delete, and connect model elements. The transformation specification consists of individual actions executed in a logical sequence or are linked by logical operations. The actions may consist of reusable mini-transformations that perform a specific operation on a model element but are not defined specifically for any one transformation specification. Mini-transformations are necessary to express operations on the model elements having the capability to be used by a wide variety of transformation specifications; therefore they are reusable. The mini-transformations are given in the Appendix.

A transformation specification is defined by combining PBAL constructs with OCL expressed preconditions and postconditions. Table 4.1 shows the general form of a transformation specification. The transformation specification specifies the conditions that restrict the applicability of transforming a model, the actions performed on a source model, and the effect of a transformation on a model. The preconditions are used to verify whether the transformation can be applied. The actions describe how the transformation accomplishes its intent by manipulating the metamodel. The postconditions determine if its application reaches its goals.

<table>
<thead>
<tr>
<th><strong>context</strong> Package PatternTransformation(PatternMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>precondition (pre):</strong> specifies what must exist before an action can be executed</td>
</tr>
<tr>
<td><strong>action:</strong> specifies what changes to perform</td>
</tr>
<tr>
<td><strong>postcondition (post):</strong> specifies the final changes made after the action executes</td>
</tr>
</tbody>
</table>
The introduction of a design pattern into a source model is accomplished by applying a sequence of transformations (actions) to an initial model in order to reach a final model. At the model level (M1), each design pattern has its own transformation specification which must be in conformance with the transformation pattern.

4.4 Validation of Model-Level Transformation

Verification of model-level transformations against metamodel-level transformations provides a method of validation for this research. Informally, the transformations are verified structurally by comparing the bindings on the elements in the target model (at the model level) against the bindings specified in the transformation schema and the restrictions defined by the transformation constraint. The transformation patterns offer an informal technique to verifying model-level transformations against metamodel-level transformations. Pattern-based model transformation can also be validated formally by (1) attaching pre- and postconditions, expressed in the OCL, to each action clause in the transformation specification produce to produce the general form of a model-level transformation program; (2) composing the pre- and postconditions at the model level into a single precondition, postcondition pair, referred to as the preModel and postModel; (3) expressing the metamodel-level transformations, referred to as MetaTransformationSpec, in terms of pre- and postconditions, where the source schema is the metamodel-level precondition, referred to as preMeta, and the transformation schema combined with the transformation constraint determines the metamodel-level postcondition, referred to as postMeta; and then (4) demonstrating that the preModel, postModel pair corresponds to the preMeta, postMeta pair.

4.4.1 Composition

The composition of model-level pre- and postconditions combines the work of Catalysis [D'Souza & Wills, 1998] with Z schema composition [Woodcock & Davis, 1996]. We compose model-level transformation pre- and postconditions to produce a single precondition, postcondition pair. This composition is required since the final postcondition (postN) specified for the last action only describes the effect of that action and not the overall effect of the transformation. For example, if the last action adds only one element, then this is specified in the postN, but all other elements added, deleted or linked by previous actions will not be reflected in postN.

The following is an example of composition:

pre1(x,y)∧ post1(x',y')) composed with (pre2(x,y)∧post2(x',y')
where the primed elements (e.g., $x'$) are the values after execution of the action and the caret character ($^\land$) represents a “logical and”. First, rename $x',y'$ in post1 and $x,y$ in pre2 so that they match (i.e., to indicate that $x',y'$ in post1 are the input states to pre2) which yields the following:

$$\text{pre}_1(x,y)^\land \text{post}_1(p,q)^\land \text{pre}_2(p,q)^\land \text{post}_2(x',y').$$

Next, bind $p,q$ to an existence quantifier.

$$\exists p,q \cdot \text{pre}_1(x,y)^\land \text{post}_1(p,q)^\land \text{pre}_2(p,q)^\land \text{post}_2(x',y')$$

The existence quantifier $\exists p,q \cdot \text{pre}_1(x,y)^\land \text{post}_1(p,q)^\land \text{pre}_2(p,q)^\land \text{post}_2(x',y')$ is true if and only if there is some $p$ and $q$ such that the predicate $\text{pre}_1(x,y)^\land \text{post}_1(p,q)^\land \text{pre}_2(p,q)^\land \text{post}_2(x,y)$ is true.

### 4.5 Summary

The core of the research is to develop a process which introduces design patterns into existing UML models using an extension of the UML metamodel. To achieve this process, we extend the UML package structure by defining a Transformation Package containing transformation pattern packages which define specialized metamodel elements with source and target model structures. Transformation patterns are defined as specialized metamodel structures which represent model elements that must be present in the source model in order to apply the transformation and model elements that are added, deleted and connected by the transformation, as well as, restriction on the transformation.

This approach also defines a specification for the implementation of the transformation on an existing UML model. This specification when applied will produce a new model instantiated with a design pattern.

In chapters 6, 7, and 8, we specify transformation patterns for a creational (Abstract Factory) pattern, behavioral (Visitor) pattern, and structural (Bridge) pattern. These patterns are specialized metamodel structures. We show how the transformation pattern captures the unique characteristics of each design pattern category.
Chapter 5
Characterizing Creational Pattern Transformations

This chapter describes the specification of a metamodel-level transformation for a creational design pattern, the Abstract Factory (AF) pattern. Section 5.1 describes the AF design pattern. Section 5.2 provides an example of incorporating the AF pattern into the MazeGame design [Gamma et al, 1994].

Section 5.3 describes the three parts of the AF transformation pattern - the source schema, the transformation schema, and the transformation constraint. The application of the transformation schema to the source schema results in an explicit definition of the AF target pattern. The target pattern contains an instantiation of the AF pattern when the transformation schema is applied in adherence to the restrictions expressed in the transformation constraint. Transformation patterns are declarative descriptions of model transformations that express relationships between model elements before and after transformations are applied. These declarative descriptions enhance a software designer’s ability to visualize the process of incorporating design patterns into models. For the AF transformation pattern, a metamodel-level transformation specification, referred to as AFMetaTransformationSpec, is obtained by viewing the source schema as metamodel-level preconditions (preMeta) and the transformation schema combined with the restriction expressed in the transformation constraint as metamodel-level postconditions (postMeta). The AFMetaTransformationSpec is used to check for conformance against model-level transformations.

Section 5.4 describes an imperative-like transformation specification for the AF transformation that provides a sequence of actions in a program-like manner which produces a target model that conforms to the target pattern as described in Section 5.3. The transformation specification restructures a class diagram at the model (M1) level. The specification of model-level transformations is required since the metamodel does not have the capability to perform operations. Also included is this section are model-level pre- and postconditions that must hold with respect to each action performed on the model. The pre- and postconditions describes the state of the model before and after the execution of an action. Knowing the state of the model determines whether or not an action can be applied, in the case of the precondition, or if an action was applied correctly, in the case of the postcondition. Next, the pre- and postconditions for each individual action are composed in order to obtain a single precondition-postcondition pair.
that specifies the initial state of the model before the transformation is applied and a final state of the model at the end of the transformation process.

In Section 5.5, we describe how to validate the AF transformation pattern using a formal method. This validation technique ensures conformance between the model-level pre- and postconditions (preModel and postModel) and the metamodel-level pre- and postconditions (preMeta and postMeta) such that the model-level pre- and postconditions express the properties of the transformation pattern. Section 5.6 validates the AF transformation using structural conformance. Ensuring the source and target models conform to the source schema and target pattern provides a graphical method to structurally validate the transformation of the model. Section 5.7 summarizes the characterization of creational transformation patterns.

5.1 The Abstract Factory Pattern

The UML class diagram given in Figure 5.1 (adapted from Gamma et al., 1994) represents the AF pattern structure. The AF pattern is a creational design pattern that “provides an interface for creating families of dependent objects without specifying their concrete classes” [Gamma et al, 1994]. The AF pattern consists of two factory clients, two product classes and a client class. The factory classes are AbstractFactory and ConcreteFactory. AbstractFactory declares an interface for operations that create abstract product objects and ConcreteFactory implements the operations to create concrete product objects. AbstractProduct declares an interface for a type of product object. ConcreteProduct defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface. A client uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Figure 5.1. Abstract Factory Design Pattern [Gamma et al., 1994]
5.2 AF Pattern Model Transformation Example

In this example, we use the class diagram (shown in Figure 5.2) and the sequence diagram (shown in Figure 5.3) for the MazeGame. The class diagram (Figure 5.2) creates two types of mazes: bombed mazes (instance of BombedMaze) and enchanted mazes (instances of EnchantedMaze). A bombed maze consists of rooms with bombs (instances of RoomWithBomb), and an enchanted maze consists of enchanted rooms (instances of EnchantedRoom). A room with a bomb consists of doors (instances of Door) and bombed walls (instances of BombedWall) while an enchanted room consists of doors that need spells (instances of DoorNeedingSpell) and ordinary walls (instances of OrdinaryWall). The sequence diagram (Figure 5.3) shows the sequence of interactions that takes place between the client and products as the client builds a bombed maze consisting of two rooms with walls and a door.

A problem with this design, as pointed out in [Gamma et al., 1994], is that the creation of mazes is hardcoded into the client (MazeGame). Consequently, changing how maze elements are created requires modifying the client.

The AF pattern describes a generic solution that can be used to make a client independent of how the products it manipulates are created. The results of transforming MazeGame class and sequence diagrams with the AF Pattern are shown in Figure 5.4 and Figure 5.5, respectively. Applying the AF pattern to a class diagram involves removing the product creations operations from the client class and creating factory classes that contain operations for creating products. Clients use the factories to create products. This separation of concerns allows changes to be made to how products are assembled without impacting the client.
Applying the AF pattern to a sequence diagram that describes product creation produces in a new sequence diagram that delegates product creation through factories. Transforming the MazeGame sequence diagram shown in Figure 5.3 using the AF pattern involves replacing each create product interaction between a client and a product by two interactions. The first interaction takes place when the client delegates the creation activity to a factory, and the second interaction takes place when the factory creates the product. For example, the interaction $r1 := \text{create}$ between a MazeGame and Rm1 in Figure 5.3 is transformed to the following two interactions in Figure 5.5: the message $\text{Rm1} := \text{makeRoom}$ between a MazeGame and BombedMazeFactory and the message $\text{Rm1} := \text{createRoom}$ between BombedMazeFactory and Rm1.
Figure 5.4. Transformed MazeGame Class Diagram [Gamma et al., 1994]

Figure 5.5. Transformed MazeGame Sequence Diagram [adapted from Gamma et al., 1994]
5.3 AF Transformation Patterns

The specification of the AF transformation pattern is given in Figure 5.6. As stated previously, a transformation pattern consists of three parts: the source schema, the transformation schema, and the transformation constraint. The source schema and transformation schema are static role model (SRM) structures. Each role in the source and transformation schema can be associated with a realization multiplicity that restricts the number of conforming elements that can be bound to the role in a conforming model.

![Figure 5.6. Abstract Factory Transformation Pattern](image)

5.3.1 Source Schema

The source schema consists of two classifier roles, the |Client and the |Product, that are connected to each other using a create dependency role, |ClientProdDependency. The |Client role is a mandatory structure whose instances are classes representing clients in the application domain. The multiplicity on the |Client role specifies that a conforming source model must have exactly one instance of |Client. A class that conforms to the |Client role consists of at least one operation that plays the |CreateOp role. The |Product role hierarchy consists of a |Product and its specializations |CompositeProduct and |SubProduct. A conforming product structure must have at least one
composite product that plays \( \text{CompositeProduct} \) role and at least one subproduct that plays the \( \text{SubProduct} \) role as indicated by the realization multiplicity \((1..*)\). An instance of the \( \text{Product} \) role is a subclass of \( \text{Class} \) whose instances are classes representing products in the application domain. The create dependency role \( \text{ClientProdDependency} \) between \( \text{Client} \) and \( \text{Product} \) specifies that instances of \( \text{Client} \) are connected to one or more classes that play the \( \text{Product} \) role via create dependency relationship.

Each end of an association role has an association-end role. The \( \text{ClientProdAssoc} \) role has two association-end roles: \( \text{ClientEnd} \) and \( \text{ProductEnd} \). The multiplicity \((1..1)\) at the \( \text{ClientEnd} \) association-end role specifies that a conforming \( \text{Client} \) class must be part of only one \( \text{ClientProdDependency} \) dependency relationship. The multiplicity \((1..*)\) at the \( \text{ProductEnd} \) association-end role indicates that one or more association-ends can be associated with a class that conforms to \( \text{Product} \).

The metamodel-level constraints defined on the AF source schema are as follows:

- A client dependency end that conforms to \( \text{ClientEnd} \) must have a multiplicity of \( 1..1 \):

  \[
  \text{context} \ \text{ClientEnd} \\
  \text{inv} \ \text{self.lowerBound()} = 1 \ \text{and} \ \text{self.upperBound()} = 1
  \]

- A supplier dependency end that conforms to \( \text{ProductEnd} \) must have a multiplicity of \( 1..* \):

  \[
  \text{context} \ \text{ProductEnd} \\
  \text{inv} \ \text{self.lowerBound()} = 1 \ \text{and} \ \text{self.upperBound()} = *
  \]

### 5.3.2 Transformation Schema

The transformation schema specifies that an AF pattern transformation introduces factory classes (instances of \( \text{Factory} \) role), specializations of factory classes (instances of \( \text{SpecializedFactory} \) role), create operations associated with the factory and specialized factory classes (instances of \( \text{CreatePartOp} \) role), create composite product operations owned by the client class (instance of \( \text{Client} \) ), and connections between factory and client classes via usage dependency (instance of \( \text{ClientFactDependency} \) ). The transformation schema also indicates that the create operations (instances of \( \text{CreateOp} \) role) owned by the client class have been deleted from the client class. That is, all instances of \( \text{CreateOp} \) are removed from the instance of \( \text{Client} \) during the transformations.

An instance of \( \text{Factory} \) is introduced into the structure and connected to instances of \( \text{Client} \) via a usage dependency (instance of \( \text{ClientFactDependency} \) ). The multiplicity of the \( \text{Factory} \) role indicates a conforming target model must have exactly one class that conforms to \( \text{Factory} \). The transformation also introduces zero or more instances of the \( \text{SpecializedFactory} \) role. Instances of \( \text{SpecializedFactory} \) are connected to the \( \text{Factory} \) role via a
generalization role (instances of \texttt{|FactoryGeneralization}) such that the specialized factory classes are specializations of the factory class. In the transformed model, the factory contains create operations that creates product parts (instances of \texttt{|CreatePartOp}). The client calls these operations and the specialized factory performs the actual creation. The usage dependency role \texttt{|ClientFactDependency} between the \texttt{|Client} and \texttt{|Factory} specifies that the one instance of the \texttt{|Client} is connected to exactly one class that plays the \texttt{|Factory} role.

The metamodel-level constraints for the target model are as follows:

- A client dependency end that conforms to \texttt{|ClientEnd} must have a multiplicity of 1..1:
  ```
  \texttt{context |ClientEnd}
  \texttt{inv: self.lowerBound = 1 and self.upperBound = 1}
  ```

- A supplier dependency end that conforms to \texttt{|FactoryEnd} must have a multiplicity of 1..1:
  ```
  \texttt{context |FactoryEnd}
  \texttt{inv: self.lowerBound = 1 and self.upperBound = 1}
  ```

- A general end that conforms to \texttt{|FactEnd} must have a multiplicity of 1..*:
  ```
  \texttt{context |FactEnd}
  \texttt{inv: self.lowerBound = 1 and self.upperBound = *}
  ```

- A specific end that conforms to \texttt{|SpecFactEnd} must have a multiplicity of 1..1:
  ```
  \texttt{context |SpecFactEnd}
  \texttt{inv: self.lowerBound = 1 and self.upperBound = 1}
  ```

Classes that play the role of \texttt{|Factory} must have at least one operation that conforms to \texttt{|CreatePartOp} which creates a new instance of \texttt{|Product}. The constraint template for \texttt{|CreatePartOp} is given below:

```
\texttt{context |Factory :: |CreatePartOp() : |Product}
\texttt{pre: true}
\texttt{post: result = p and p.oclIsNew() = true}
```

A similar constraint template exists for the \texttt{|SpecializedFactory}.

```
\texttt{context |SpecializedFactory :: |CreatePartOp() : |Product}
\texttt{pre: true}
\texttt{post: result = p and p.oclIsNew() = true}
```

In the transformation schema, classes that play the role of \texttt{|Client} must have exactly one operation that conforms to \texttt{|CreateCompProd} which creates a new instance of composite product. The constraint template for \texttt{|CreateCompProd} is given below:

```
\texttt{context |Client :: |CreateCompProd() : |CompositeProduct}
\texttt{pre: true}
\texttt{post: result = p and p.oclIsNew() = true}
```
5.3.3 Transformation Constraint

A transformation constraint, as shown in Figure 5.6(c), specifies restrictions and relationships that must hold between source and target model elements.

In an AF pattern transformation a unique specialized factory must be created for each create operation (instance of |CreateOp) owned by the client in the source model. This constraint is expressed by the correspondsTo dependency, shown in Figure 5.6(c), between unique instances of |CreateOp and unique instances of |SpecializedFactory. The metamodel-level constraint that must satisfy the correspondsTo dependency states that the number of specialized factory classes (instances of |SpecializedFactory) in the target model must equal the number of create operations (instances of |CreateOp) owned by the client class in the source model. The OCL expression for this constraint is given as follows:

```
context |Client
inv: self.|Factory → collect(|SpecializedFactory) → size() = self.|CreateOp → size()
```

The number of specialized factories in the collection of |SpecializedFactory instances (as determined by the size of the collection) should equal the number of create operations in the client class (as determined by the size of the collection of |CreateOp) instances.

An AF transformation needs information about the type of products created by each create operation owned by a client in order to assign create operations to the appropriate factories. The CreatedProducts association between |CreateOp and |Product shown in the transformation constraint is a derived relationship that provides the information needed by an AF transformation. The constraint template for CreatedProducts is:

```
context |CreateOp :: CreatedProducts() : Set(|Products)
pre: true
post: self.activity.action → select(self.oclIsTypeOf (CreateObjectAction)) → collect(Classifier) →
select(oclIsTypeOf (|Product) ) → asSet()
```

The create actions defined by the instances of |CreateOp become operations in the specialized factories corresponding to the |CreateOp operations in an AF transformation. For each product created by an instance of |CreateOp (i.e., each product in the set determined by the calculated relationship CreatedProducts) there must exist an instance of |CreatePartOp that creates the product in the specialized factory corresponding to the |CreateOp instance. This constraint is represented by the set of links between the |Product set and the |CreatePartOp set. The set of links between the sets of |CreatePartOp in specialized factories and the factory indicates that the factory class (the root generalization) consists of create operations that are inherited by the specialized factories.
The metamodel-level constraints for CreatedProducts specifies that the number of create product operations (instances of \texttt{|CreatePartOp}) owned by instances of \texttt{|SpecializedFactory} is equal to the number of created products.

\begin{verbatim}
context |SpecializedFactory
inv: self.|CreatePartOp \rightarrow size() = CreatedProducts() \rightarrow size()
\end{verbatim}

The metamodel-level constraints for CreatedProducts also specify that the number of create product operations (instances of \texttt{|CreatePartOp}) owned by instances of \texttt{|Factory} is equal to the number of created products.

\begin{verbatim}
context |Factory
inv: self.|CreatePartOp \rightarrow size() = CreatedProducts() \rightarrow size()
\end{verbatim}

5.3.4 Metamodel-Level Pre- and Postconditions

At the metamodel (M2) level, we can view the AF transformation pattern as a metamodel-level transformation specification, referred to as AFMetaTransformationSpec. The source schema graphically depicts the metamodel-level precondition, referred to as preMeta, which must be satisfied before a transformation can execute. The postconditions are depicted graphically in the transformation schema, along with the restrictions specified in the transformation constraint. These postconditions at the metamodel level are referred to as postMeta.

5.4 AF Pattern Transformation Specification

An AF pattern transformation specification (i.e., program) defines the sequence of actions required to introduce the AF pattern into an existing UML model to create a new model instantiated with the AF pattern. The transformation specification is expressed using the PBAL action language.

The transformation program for the AF transformation pattern is given in Table 5.1. The AF transformation specification is defined for a package consisting of a source model (i.e., the context of the AFTransformation is the source model upon which the transformation is applied). The precondition, isValidSource(metamodel), verifies that the source model conforms to the metamodel as specified by the source schema structure in the AF Transformation pattern, shown in Figure 5.6. If the source model conforms to the source schema, the operation isValidSource returns true and the transformation of the model can proceed. An AF Transformation involves the following steps:

1. **Create a Factory Class.** This step creates a factory class and a dependency class. The dependency class specifies the relationship between the factory and client classes. The client end of the Dependency class is connected to the Client class and the supplier end of the Dependency class to the Factory class.

2. **Create Factory Operations for each product in the set of Created Products.** Create product operations (instances of \texttt{|CreatePartOp}) are created from a collection of created products (CreatedProducts) and linked
to the instance of |Factory such that the instances of |CreatePartOp are owned operations of the Factory class.

3. **Create Specialized Factory Classes.** In this step, specialized factory classes (instances of |SpecializedFactory) are created for each instance of |CreateOp owned by the instance of |Client. A factory generalization (instance of |FactoryGeneralization) is created. The general end of the factory generalization is linked to the Factory. For each specialized factory class, link the specific end of the factory generalization to the specialized factory class.

4. **Create Specialized Factory Operations.** This step involves creating create product operations (instances of |CreatePartOp) for each product in the collection of created products (instances of |Product) classes. A set of operations (instance of |CreatePartOp) are created for each instance of |SpecializedFactory and then connected to the instances of |SpecializedFactory such that the instances of |CreatePartOp are owned by the instances of |SpecializedFactory.

5. **Create Composite Product Operation.** A composite product operation (instance of |CreateCompProd) is created and connected to the client class such that the instance of |CreateCompProd is an owned operation of the instance of |Client.

6. **Remove CreateOp operations.** This step deletes all create operations (instances of |CreateOp) owned by the client class such that all references to |CreateOp no longer exist in the application domain.

<table>
<thead>
<tr>
<th>Table 5.1. AF Transformation Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context</strong> Package :: AFTransformation (AFMetamodel)</td>
</tr>
<tr>
<td><strong>pre:</strong> self.isValidSource(AFMetamodel) -- isValidSource(mm) returns true if the model conforms to the metamodel mm</td>
</tr>
</tbody>
</table>
| **action:** /* all variables are local therefore must be declared */
| indx1 : Integer; |
| indx2 : Integer; |
| num_createops : Integer; |
| num_createdprod : Integer; |
| num_specfactory : Integer; |
| /* action1 - Create a factory class & connect factory class to client class via Dependency Relationship */
| Factory a_factory; // Factory variable declaration |
| a_factory ::= _create_instance(Factory); // Create a Factory class |
| Client a_client; // Client variable declaration |
| a_client = _get_instances(Client); // get all instances of Client |
| ClientFactDependency a_clientfactdepend; // Dependency variable declaration |
Table 5.1 continued

```plaintext
a_clientfactdepend ::= _create_instance(ClientFactDependency); // create instance of dependency
_connectClasses_Dependency(a_clientfactdepend, a_factory, a_client); // connect factory class to
client class via a Dependency relationship

/* action2 - In the factory class, include a create operation for each element in the collection of
CreatedProducts */
num_createdprod = CreatedProducts → size(); // get the number of elements in the collection of
CreatedProducts
indx1 = 1;
while (indx1 <= num_createdprod) do {
  a_createpartop[indx1] ::= _create_instance(CreatePartOp); // create an instance of CreatePartOp
  _connect_Op2Class (a_factory, a_createpartop[indx1]); // connect the create parts operation to
  the factory class
  indx1 = indx1 + 1;
}

/* action3 - create specialized factory classes */
CreateOp[ ] a_createop; // CreateOp variable declaration
a_createop ::= _get_operations(a_client, CreateOp); // get all instances of CreateOp owned by the
  Client class
num_createops = a_createop → size(); // get the number of create operations
SpecializedFactory[ ] a_specfactory; // SpecializedFactory set variable declaration
FactoryGeneralization a_factorygen; // FactoryGeneralization variable declaration
a_factorygen ::= _create_instance(FactoryGeneralization); // create an instance of
  FactoryGeneralization
indx1 = 1;
while (indx1 <= num_createops) do {
  a_specfactory[indx1] ::= _create_instance(SpecializedFactory); // create an instance of
    SpecializedFactory
  _connectClasses_FactoryGeneralization(a_factorygen, a_factory, a_specfactory[indx1]); // connect factory class to specialized factory class
  indx1 = indx1 + 1;
}

/* action4 - For each specialized factory, include a create operation for each element in the collection
of CreatedProducts. */
num_createdprod = CreatedProducts → size(); // get the number of elements in the collection of
CreatedProducts
num_specfactory = a_specfactory → size(); // get the number of specialized factory classes
indx1 = 1;
while (indx1 <= num_specfactory) do {
  CreatePartOp[ ] a_createpartop;
  indx2 = 1;
  while (indx2 <= num_createdprod) do {
    a_createpartop[indx2] ::= _create_instance(CreatePartOp); // create an instance of
      CreatePartOp
    _connect_Op2Class (a_specfactory[indx1], a_createpartop[indx2]); // connect the create parts
      operation to the specialized factory
    indx2 = indx2 + 1;
  }
  indx1 = indx1 + 1;
}

/* action5 - Add an operation (instance of CreateCompProd) to the client class */
```


Table 5.1 continued

CreateCompProd a_createCompProd; // CreateCompProd variable declaration
a_createCompProd := _create_instance(CreateCompProd); // create an instance of CreateCompProd
_create_link_compprodLink(a_client, a_createCompProd); // connect create composite product
operation to the client class

/* action6 - Remove instances of CreateOp from the client class. */
CreateOp[] a_createop; // CreateOp variable declaration
a_createop ::= _get_operations(a_client, CreateOp); // get all instances of CreateOp owned by the
Client class
num_createops = a_createop → size(); // get the number of create operations owned by the client
class
indx1 = 1;
while (indx1 <= num_createops) do {
    _destroy_instance( a_createop[indx1] );
    indx1 = indx1 + 1;
}

5.4.1 Model-Level Pre- and Postconditions

The model-level transformation specification, given in Table 5.1, can be expressed as a sequence of action
clauses that convey individual operations on model elements needed to perform AF model-level transformation.
Each action has a precondition and postcondition that must be satisfied before and after the execution of an action.
Tables 5.2 - 5.7 provide the pre- and postconditions for each individual action. The precondition specifies what
must exist in the model before the execution of the action and the postcondition specifies what elements have been
added, deleted, or connected by the action (i.e., the state of the model after the execution of the action).

Table 5.2. Action 1 - Create Factory Class

context Package :: AFTransformation (AFMetamodel)
pre1: true
post1: self.allClasses() = self.allClasses()@pre including(a_factory) -- at least one factory class exist the
collection of classes
and self.allClasses() → one(a_factory) -- only one factory class connect to client
and self.allDependencies() = self.allDepenedences@pre → including(a_clientfactdepend) -- the
collection of dependences has a_clientfactdepend added after execution.
and self.a_client → select(fc | fc.oclIsTypeOf(FactoryEnd)) = self.allClasses() → select(a_factory)--
a_client is connected to client end dependency
and self.allClasses() → collect(a_factory) → select(ce| ce.oclIsTypeOf(ClientEnd)) =
self.allClasses() → select(a_client)-- a_factory is connected to supplier end of dependency

Table 5.2 contains the pre- and postconditions for Action 1 which creates a factory class and connects the
factory to the client class. The precondition of this action is true. The postcondition ensures (1) that only one
factory element exists in the model after the action, (2) a dependency (a_clientfactdepend) has been added to the
collection of dependencies after the action, and (3) the Client class (a_client) is connected to the Factory class
(a_factory) via a dependency relationship (a_clientfactdepend) where a_client is connected the client end of a_clientfactdepend and the a_factory is connected to the supplier end of a_clientfactdepend.

<table>
<thead>
<tr>
<th>Table 5.3. Action 2 - Create Factory Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context</strong> Package :: AFTransformation (AFMetamodel)</td>
</tr>
<tr>
<td><strong>pre2:</strong></td>
</tr>
<tr>
<td>self.allClasses() (\rightarrow) exist(a_factory) -- factory exist in collection of classes</td>
</tr>
<tr>
<td>and self.a_client.a_factory.allOperations() (\rightarrow) excludes(cpo</td>
</tr>
<tr>
<td><strong>post2:</strong></td>
</tr>
<tr>
<td>self.a_client.a_factory.allOperations() = self.a_client.a_factory.allOperations()@pre (\rightarrow)</td>
</tr>
<tr>
<td>including(a_createpartop) -- all instances of CreatePartOp are owned by factory</td>
</tr>
<tr>
<td>and self.a_client.a_factory.allOperations() (\rightarrow) select(a_createpartop) (\rightarrow) size() = CreatedProducts (\rightarrow)</td>
</tr>
<tr>
<td>size() -- number of factory operations equal to the number of products</td>
</tr>
</tbody>
</table>

The pre- and postconditions for Action 2, which is responsible for creating factory operations, are given in Table 5.3. The precondition specifies that a factory class must exist in the collection of classes, and the collection of operations owned by the factory class does not contain elements that are instances of CreatePartOp. The postcondition states for the action to hold (1) the collection of operations owned by the factory class must consist of elements that are instances of CreatePartOp, and (2) the number of factory operations (instances of CreatePartOp) must equal the number of created products derived from the source model.

<table>
<thead>
<tr>
<th>Table 5.4. Action 3 - Create Specialized Factory Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context</strong> Package :: AFTransformation (AFMetamodel)</td>
</tr>
<tr>
<td><strong>pre3:</strong></td>
</tr>
<tr>
<td>self.allClasses() (\rightarrow) exist(a_factory) -- factory class element in collection in collection of classes</td>
</tr>
<tr>
<td>and self.allClasses() (\rightarrow) excludes(sf</td>
</tr>
<tr>
<td>and self.allGeneralizations() (\rightarrow) excludes(fg</td>
</tr>
<tr>
<td><strong>post3:</strong></td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses()@pre (\rightarrow) including(a_specfactory)) -- SpecializedFactory element in collection of classes</td>
</tr>
<tr>
<td>and self.allGeneralizations() = self.allGeneralizations()@pre (\rightarrow) including(a_factorygen) -- FactoryGeneralization element in collection of generalizations</td>
</tr>
<tr>
<td>and self.a_client.a_factory (\rightarrow) collect(a_specfactory) (\rightarrow) size() = self.a_client@pre (\rightarrow)</td>
</tr>
<tr>
<td>select(a_createop) (\rightarrow) size() -- number of factory equivalent to the number of create operations</td>
</tr>
<tr>
<td>and self.a_client.a_factory.allChildren() = self.allClasses() (\rightarrow) select(a_specfactory) -- child classes of factory are instances of SpecializedFactory</td>
</tr>
<tr>
<td>and self.allClasses() (\rightarrow) select(a_specfactory).allParents() = self.allClasses() (\rightarrow) select(a_factory) -- parent class of collection of Specialized Factory is Factory</td>
</tr>
</tbody>
</table>

The pre- and postconditions expressed in Table 5.4 are the constraint conditions that specify the applicability and effect of creating specialized factory classes. The precondition specifies that a factory class must exist in the collection of classes, specialized factory classes are not elements in the collection of classes, and the collection of
generalizations does not elements that are instances of |FactoryGeneralization. The postcondition specifies that after the completion of the action, specialized factory classes must exist for every instance of CreateOp in the Client class, and the specialized factory classes are specializations of the factory class.

The operations children and allChildren are not defined in the OCL documentation or as an additional operation in the UML metamodel. They are defined below as operations for classifiers.

```plaintext
context Classifier :: Children() : Set(Classifier)
inv children = generalization.specific

context Classifier :: allChildren() : Set(Classifier)
inv allChildren = self.children → union(self.children → collect(c | c.allChildren()))
```

Table 5.5 gives the pre- and postconditions for adding create product operations (instances of |CreatePartOp) to each specialized factory classes (Action 4). The precondition specifies that specialized factory classes must exist in the collection of classes and the collection of operations owned by the specialized factory classes does not contain elements that are instances of |CreatePartOp. The postcondition specifies that after the completion of the action, create product operations are added to the specialized factory classes for every product in the collection of CreatedProducts.

```
Table 5.5. Action 4 - Specialized Factory Operations

context Package :: AFTransformation (AFMetamodel)
pre4:
  self.allClasses() → exist(a_specfactory)
  and self.allClasses() → select(a_specfactory).allOperations() → excludes(cpo | cpo.oclIsTypeOf(CreatePartOp))
post4:
  self.allClasses() → select(a_specfactory).allOperations() = self.allClasses() →
  select(a_specfactory).allOperations()@pre → including(a_createpartop)
  and self.allClasses() → select(a_specfactory).allOperations() → select(a_createpartop) → size() =
  CreatedProducts() → size() -- number of product operations equals number of Product instances
```

Action 5 introduces one create composite product operation (instance of |CreateCompProd) to the client class. The pre- and postconditions are given in Table 5.6. The precondition ensures that the collection of operations owned by the client class does not consist of elements that play the role of |CreateCompProd. The postcondition specifies that after the completion of the action, the client class owns only one create product operation (instance of |CreatePartOp).

```
Table 5.6 specifies the postcondition for the action that removes all instances of create operations (instances of |CreateOp) from the client class. The postcondition specifies that the after the completion of the action, all create operations that are instances of |CreateOp are deleted from the collection of operations owned by the client class.
```
Table 5.6. Action 5 - Composite Product Operation

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: AFTransformation (AFMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre5:</td>
<td>self.a_client.allOperations() (\rightarrow) excludes(oclIsTypeOf(CreateCompProd)) -- composite product operation not operation of Client</td>
</tr>
<tr>
<td>post5:</td>
<td>self.a_client.allOperations() = self.a_client.allOperations()@pre (\rightarrow) including(a_createCompProd) -- composite product operation owned by Client and self.a_client.allOperations() (\rightarrow) one(a_createCompProd) -- one composite product operation exist</td>
</tr>
</tbody>
</table>

Table 5.7. Action 6 - Remove CreateOp from Client Class

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: AFTransformation (AFMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre6:</td>
<td>none</td>
</tr>
<tr>
<td>post6:</td>
<td>self.a_client.allOperations() = self.a_client.allOperations()@pre (\rightarrow) excluding(a_createop) -- create operations not a Client operation</td>
</tr>
</tbody>
</table>

5.4.2 Composition of AF Model-Level Pre- and Postconditions

A single precondition postcondition pair, where the precondition is referred to as preModel and the postcondition is referred to as postModel, is obtained by composing each individual pair, given in Tables 5.2 - 5.7, two at a time. The character \(\land\) represents a “logical and”.

1. Compose Action1 pre- and postconditions (pre1 \(\land\) post1) with Action2 pre- and postconditions (pre2 \(\land\) post2).
   - Rename variables that affect the occurrence in post1 and pre2
     - a_factory \(\Rightarrow\) z, where \(\Rightarrow\) stand for “rename”

The OCL expressions for post1 and pre2 are renamed as follows:

post1: \{ self.allClasses() = self.allClasses()@pre \(\rightarrow\) including(z) \(\land\) self.a_client \(\rightarrow\) one(z) \(\land\) self.allDependencies() = self.allDependencies()@pre \(\rightarrow\) including(a_clientfactdepend) \(\land\) self.a_client \(\rightarrow\) select(fc | fc.oclIsTypeOf(FactoryEnd)) = self.allClasses() \(\rightarrow\) select(a_factory)\(\land\) self.a_client.z.supplierDependency = a_clientfactdepend \}

pre2: \{ self.allClasses() \(\rightarrow\) exist(z) \(\land\) self.a_client.z.allOperations() \(\rightarrow\) excludes(cpo | cpo.oclIsTypeOf(CreatePartOp)) \}

- Bind z to an existence quantifier

\(\exists z: \{ self.allClasses() = self.allClasses()@pre \(\rightarrow\) including(z) \(\land\) self.a_client \(\rightarrow\) one(z) \(\land\) self.allDependencies() = self.allDependencies()@pre \(\rightarrow\) including(a_clientfactdepend) \(\land\) self.a_client \(\rightarrow\) select(fc | fc.oclIsTypeOf(FactoryEnd)) = self.allClasses() \(\rightarrow\) select(a_factory)\(\land\) self.a_client.z.supplierDependency = a_clientfactdepend \(\land\) self.allClasses() \(\rightarrow\) exist(z) \(\land\) self.a_client.z.allOperations() \(\rightarrow\) excludes(cpo | cpo.oclIsTypeOf(CreatePartOp)) \} \)
• Simplify the expression
  
  The OCL expressions self.allClasses() = self.allClasses()@pre \rightarrow including(z) and self.allClasses() \rightarrow exist(z) are equivalent expressions since both verify that the z is an element in the collection of classes. The expression self.allClasses() = self.allClasses()@pre \rightarrow including(z) will be used in the composed postcondition since the OCL @pre construct allows the model elements in the source model to be referenced in the postcondition.

  The element a_createpartop is an instance of CreatePartOp. The OCL expressions
  
  self.a_client.a_factory.allOperations() = self.a_client.a_factory.allOperations()@pre \rightarrow including(a_createpartop) and self.a_client.z.allOperations() \rightarrow excludes(cpo | cpo.oclIsTypeOf(CreatePartOp)) references the same model element. The expression
  
  self.a_client.z.allOperations() \rightarrow excludes(cpo | cpo.oclIsTypeOf(CreatePartOp)) refers to the state of the model before execution of the action and the expression
  
  self.a_client.a_factory.allOperations() = self.a_client.a_factory.allOperations()@pre \rightarrow including(a_createpartop) references the source model in the postcondition. The focus of the postcondition is the state of the model after execution of the action. The OCL expression
  
  self.a_client.a_factory.allOperations() = self.a_client.a_factory.allOperations()@pre \rightarrow including(a_createpartop) will be used in the composed postcondition since it shows that the state of the model has changed.

| pre: true |
|post2': |
|{ self.allClasses() = self.allClasses()@pre \rightarrow including(a_factory) ^ self.allClasses() \rightarrow one(a_factory) ^ self.allDependencies() = self.allDependencies()@pre \rightarrow including(a_clientfactdepend) ^ self.a_client \rightarrow select(fc | fc.oclIsTypeOf(FactoryEnd)) = self.allClasses() \rightarrow select(a_factory)^ self.allClasses() \rightarrow collect(a_factory) \rightarrow select(ce ce.oclIsTypeOf(ClientEnd)) = self.allClasses() \rightarrow select(a_client)^ self.a_client.a_factory.allOperations() = self.a_client.a_factory.allOperations()@pre \rightarrow including(a_createpartop) ^ self.a_client.a_factory.allOperations() \rightarrow select(a_createpartop) \rightarrow size() = CreatedProducts \rightarrow size() } |
The pre- and postconditions, with \( z \) changed back to \( \text{a}_\text{factory} \), for the composition of Action 1 with the Action 2 are shown in Table 5.8.

2. Compose pre- and postconditions (\( \text{pre}^2 \text{post}_2^\prime \)) with Action3 pre- and postconditions (\( \text{pre}^3 \text{post}_3 \)).

- Rename variables that affect the occurrence in \( \text{post}_2^\prime \) and \( \text{pre}_3 \)
  - \( \text{a}_\text{factory} \Rightarrow z \)

The OCL expressions for \( \text{post}_2^\prime \) and \( \text{pre}_3 \) are renamed as follows:

\[
\text{post}_2^\prime: \{ \ 	ext{self.allClasses}() = \text{self.allClasses}()@\text{pre} \rightarrow \text{including}(z) \ ^\wedge \text{self.a_client} \rightarrow \text{one}(z) \ ^\wedge \\
\text{self.allDependencies}() = \text{self.allDependencies}()@\text{pre} \rightarrow \text{including}(\text{a_clientfactdepend}) \ ^\wedge \\
\text{self.a_client} \rightarrow \text{select}(fc \ | \ fc.\text{oclIsTypeOf}(	ext{FactoryEnd})) = \text{self.allClasses}() \rightarrow \text{select}(\text{a}_\text{factory}) \ ^\wedge \\
\text{self.a_client}.z.\text{supplierDependency} = \text{a_clientfactdepend} \ ^\wedge \text{self.a_client}.z.\text{allOperations}() = \\
\text{self.a_client}.z.\text{allOperations}()@\text{pre} \rightarrow \text{including}(\text{a_createpartop}) \ ^\wedge \text{self.a_client}.z.\text{allOperations}() \rightarrow \text{select}(\text{a_createpartop}) \rightarrow \text{size()} = \text{CreatedProducts} \rightarrow \text{size}() \}
\]

\[
\text{pre}_3: \{ \ 	ext{self.allClasses}() \rightarrow \text{exist}(z) \ ^\wedge \text{self.allClasses}() \rightarrow \text{excludes}(sf \ | \\
\text{sf.oclIsTypeOf}(	ext{SpecializedFactory})) \ ^\wedge \text{self.allGeneralizations}() \rightarrow \text{excludes}(fg \ | \\
\text{fg.oclIsTypeOf}(	ext{FactoryGeneralization})) \}
\]

- Bind \( z \) to an existence quantifier

\[
\exists z: \{ \ 	ext{self.allClasses}() = \text{self.allClasses}()@\text{pre} \rightarrow \text{including}(z) \ ^\wedge \text{self.a_client} \rightarrow \text{one}(z) \ ^\wedge \\
\text{self.allDependencies}() = \text{self.allDependencies}()@\text{pre} \rightarrow \text{including}(\text{a_clientfactdepend}) \ ^\wedge \\
\text{self.a_client} \rightarrow \text{select}(fc \ | \ fc.\text{oclIsTypeOf}(	ext{FactoryEnd})) = \text{self.allClasses}() \rightarrow \text{select}(\text{a}_\text{factory}) \ ^\wedge \\
\text{self.a_client}.z.\text{supplierDependency} = \text{a_clientfactdepend} \ ^\wedge \text{self.a_client}.z.\text{allOperations}() = \\
\text{self.a_client}.z.\text{allOperations}()@\text{pre} \rightarrow \text{including}(\text{a_createpartop}) \ ^\wedge \text{self.a_client}.z.\text{allOperations}() \rightarrow \text{select}(\text{a_createpartop}) \rightarrow \text{size()} = \text{CreatedProducts} \rightarrow \text{size}() \ ^\wedge \text{self.allClasses}() \rightarrow \text{exist}(z) \ ^\wedge \\
\text{self.allClasses}() \rightarrow \text{excludes}(sf \ | \ sf.\text{oclIsTypeOf}(	ext{SpecializedFactory})) \ ^\wedge \text{self.allGeneralizations}() \rightarrow \text{excludes}(fg \ | \ fg.\text{oclIsTypeOf}(	ext{FactoryGeneralization})) \ ^\wedge \text{self.allClasses}() = \\
\text{self.allClasses}()@\text{pre} \rightarrow \text{including}(\text{a_specfactory}) \ ^\wedge \text{self.allGeneralizations}() = \\
\text{self.allGeneralizations}()@\text{pre} \rightarrow \text{including}(\text{a_factorygen}) \ ^\wedge \text{self.a_client.a_factory} \rightarrow \\
\text{collect}(\text{a_specfactory}) \rightarrow \text{size()} = \text{self.a_client}@\text{pre} \rightarrow \text{select}(\text{a_createop}) \rightarrow \text{size}() \ ^\wedge \\
\}
\]
self.a_client.a_factory.allChildren() = self.allClasses() → select(a_specfactory) ^ self.allClasses() → select(a_specfactory).allParents() = self.allClasses() → select(a_factory) 

- Simplify the expression

  - The OCL expressions self.allClasses() = self.allClasses()@pre → including(z) and self.allClasses() → exist(z) are equivalent expressions since both verify that the z is an element in the collection of classes. The expression self.allClasses() = self.allClasses()@pre → including(z) will be used in the composed postcondition since the OCL @pre construct allows the model elements in the source model to be referenced in the postcondition.

  - The element a_specfactory is an instance of SpecializedFactory. The OCL expressions self.allClasses()→ excludes(sf | sf.oclIsTypeOf( SpecializedFactory))and self.allClasses() = self.allClasses()@pre → including(a_specfactory) references the same model element. The expression self.allClasses()→ excludes(sf | sf.oclIsTypeOf(SpecializedFactory)) refers to the state of the model before execution of the action and the expression self.allClasses() = self.allClasses()@pre → including(a_specfactory)) references the source model in the postcondition. The focus of the postcondition is the state of the model after execution of the action. The OCL expression self.allClasses() = self.allClasses()@pre → including(a_specfactory) will be used in the composed postcondition since it shows that the state of the model has changed.

  - The element a_factorygen is an instance of FactoryGeneralization. The OCL expressions self.allGeneralizations()→ excludes(sf | sf.oclIsTypeOf( FactoryGeneralization))and self.allGeneralizations() = self.allGeneralizations()@pre → including(a_factorygen) references the same model element. The expression self.allGeneralizations()→ excludes(sf | sf.oclIsTypeOf( FactoryGeneralization)) refers to the state of the model before execution of the action and the expression self.allGeneralizations() = self.allGeneralizations()@pre → including(a_factorygen) references the source model in the postcondition. The focus of the postcondition is the state of the model after execution of the action. The OCL expression self.allGeneralizations() = self.allGeneralizations()@pre → including(a_factorygen) will be used in the composed postcondition since it shows that the state of the model has changed.
The pre- and postconditions, with \( z \) changed back to \( a_{\text{factory}} \), for the composition of \( \text{pre}'\text{post}2' \) with the pre- and postconditions for Action3 are shown in Table 5.9.

Table 5.9. Composed pre\textsuperscript{'}post2\textsuperscript{' with Action 3 (pre3\textsuperscript{'post}3\textsuperscript{'})

<table>
<thead>
<tr>
<th>pre: true</th>
</tr>
</thead>
<tbody>
<tr>
<td>post3\textsuperscript{'}:</td>
</tr>
<tr>
<td>{ self.allClasses() = self.allClasses()@pre \rightarrow including(a_{\text{factory}}) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow one(a_{\text{factory}}) ^</td>
</tr>
<tr>
<td>self.allDependencies() = self.allDependencies()@pre \rightarrow including(a_{\text{clientfactdepend}}) ^</td>
</tr>
<tr>
<td>self.a_client \rightarrow select(fc</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow collect(a_{\text{factory}}) \rightarrow select(ce</td>
</tr>
<tr>
<td>select(a_{\text{client}})^</td>
</tr>
<tr>
<td>self.a_client.a_{\text{factory}}.allOperations() = self.a_client.a_{\text{factory}}.allOperations()@pre \rightarrow</td>
</tr>
<tr>
<td>including(a_{\text{createpartop}}) ^</td>
</tr>
<tr>
<td>self.a_client.a_{\text{factory}}.allOperations() \rightarrow select(a_{\text{createpartop}}) \rightarrow size() = CreatedProducts \rightarrow size() ^</td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses()@pre \rightarrow including(a_{\text{specfactory}}) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() = self.allGeneralizations()@pre \rightarrow including(a_{\text{factorygen}})</td>
</tr>
<tr>
<td>self.a_client.a_{\text{factory}} \rightarrow collect(a_{\text{specfactory}}) \rightarrow size() = self.a_client@pre \rightarrow select(a_{\text{createop}})</td>
</tr>
<tr>
<td>\rightarrow size() ^</td>
</tr>
<tr>
<td>self.a_client.a_{\text{factory}}.allChildren() = self.allClasses() \rightarrow select(a_{\text{specfactory}})^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow select(a_{\text{specfactory}}).allParents() = self.allClasses() \rightarrow select(a_{\text{factory}}) }</td>
</tr>
</tbody>
</table>

3. Compose pre- and postconditions (pre\textsuperscript{'}post3\textsuperscript{'}) with Action4 pre- and postconditions (pre4\textsuperscript{'post}4).

- Rename variables that affect the occurrence in post3\textsuperscript{'} and pre4
  - a\text{specfactory} \Rightarrow y

The OCL expressions for post3\textsuperscript{'} and pre4 are renamed as follows:

post3\textsuperscript{'}: \{ self.allClasses() = self.allClasses()@pre \rightarrow including(a_{\text{factory}}) ^ self.allClasses() \rightarrow |
| one(a_{\text{factory}}) ^ self.allDependencies() = self.allDependencies()@pre \rightarrow |
| including(a_{\text{clientfactdepend}}) ^ self.a_client \rightarrow select(fc | fc.oclIsTypeOf(FactoryEnd)) = |
| self.allClasses() \rightarrow select(a_{\text{factory}})^ self.allClasses() \rightarrow collect(a_{\text{factory}}) \rightarrow select(ce | |
| ce.oclIsTypeOf(ClientEnd)) = self.allClasses() \rightarrow select(a_{\text{client}})^ |
| self.a_client.a_{\text{factory}}.allOperations() = self.a_client.a_{\text{factory}}.allOperations()@pre \rightarrow |
| including(a_{\text{createpartop}}) ^ self.a_client.a_{\text{factory}}.allOperations() \rightarrow select(a_{\text{createpartop}}) \rightarrow |
| size() = CreatedProducts \rightarrow size() ^ self.allClasses() = self.allClasses()@pre \rightarrow including(y) ^ |
| self.allGeneralizations() = self.allGeneralizations()@pre \rightarrow including(a_{\text{factorygen}}) ^ |
| self.a_client.a_{\text{factory}} \rightarrow collect(y) \rightarrow size() = self.a_client@pre \rightarrow select(a_{\text{createop}}) \rightarrow size() ^ |
| self.a_client.a_{\text{factory}}.allChildren() = self.allClasses() \rightarrow select(y) ^ self.allClasses() \rightarrow |
| select(y).allParents() = self.allClasses() \rightarrow select(a_{\text{factory}}) \}
\textbf{pre4:} \{ self.allClasses() \rightarrow exist(y) \land self.allClasses() \rightarrow select(y).allOperations() \rightarrow excludes(cpo \mid 
\text{cpooclIsTypeOf(CreatePartOp))} \}

\begin{itemize}
  \item Bind y to an existence quantifier
  \[ \exists z: \{ \text{self.allClasses()} = \text{self.allClasses()@pre \rightarrow including(a\_factory)} \land \text{self.allClasses()} \rightarrow 
  \text{one(a\_factory)} \land \text{self.allDependencies()} = \text{self.allDependencies()@pre \rightarrow} 
  \text{including(a\_clientfactdepend)} \land \text{self.a\_client \rightarrow select(fc | fcoclIsTypeOf(FactoryEnd))} = 
  \text{self.allClasses()} \rightarrow select(a\_factory) \land \text{self.allClasses()} \rightarrow collect(a\_factory) \rightarrow select(ce | ceoclIsTypeOf(ClientEnd)) = 
  \text{self.allClasses()} \rightarrow select(a\_client) \land 
  \text{self.a\_client.a\_factory.allOperations()} = \text{self.a\_client.a\_factory.allOperations()@pre \rightarrow} 
  \text{including(a\_createpartop)} \land \text{self.a\_client.a\_factory.allOperations()} \rightarrow select(a\_createpartop) \rightarrow 
  \text{size()} = \text{CreatedProducts} \rightarrow \text{size()} \land \text{self.allClasses()} = \text{self.allClasses()@pre \rightarrow including(y)} \land 
  \text{self.allGeneralizations()} = \text{self.allGeneralizations()@pre \rightarrow including(a\_factorygen)} \land 
  \text{self.a\_client.a\_factory \rightarrow collect(y)} \rightarrow \text{size()} = \text{self.a\_client@pre \rightarrow select(a\_createop) \rightarrow size()} \land 
  \text{self.a\_client.a\_factory.allChildren()} = \text{self.allClasses()} \rightarrow select(y) \land \text{self.allClasses()} \rightarrow 
  select(y).allParents() = \text{self.allClasses()} \rightarrow select(a\_factory) \land \text{self.allClasses()} \rightarrow exist(y) \land 
  \text{self.allClasses()} \rightarrow select(y).allOperations() \rightarrow excludes(cpo \mid \text{cpooclIsTypeOf(CreatePartOp)} \rangle \land 
  \text{self.allClasses()} \rightarrow select(a\_specfactory).allOperations() = \text{self.allClasses()} \rightarrow 
  select(a\_specfactory).allOperations()@pre \rightarrow including(a\_createpartop) \land \text{self.allClasses()} \rightarrow 
  select(a\_specfactory).allOperations() \rightarrow select(a\_createpartop) \rightarrow size() = \text{CreatedProducts}() \rightarrow 
  size() \}
\end{itemize}

\begin{itemize}
  \item Simplify the expression
  \begin{itemize}
    \item The OCL expressions self.allClasses() = self.allClasses()@pre \rightarrow including(y) and self.allClasses() \rightarrow exist(y) are equivalent statements since both verify that y is an element in the collection of classes. The expression self.allClasses() = self.allClasses()@pre \rightarrow including(y) will be used in the composed postcondition since the OCL @pre construct allows the model elements in the source model to be referenced in the postcondition.
    \item Changing the variable y back to a\_specfactory, the composed pre-and postconditions can be further simplified. The model element a\_specfactory is an instance of SpecializedFactory. The
OCL expressions self.allClasses() → select(a_specfactory).allOperations() → excludes(cpo | cpo.oclIsTypeOf(FactoryOp)) and self.allClasses() → select(a_specfactory).allOperations() = self.allClasses() → select(a_specfactory).allOperations()@pre → including(a_createpartop) references the same model element. The expression self.allClasses() → select(a_specfactory).allOperations() → excludes(cpo | cpo.oclIsTypeOf(FactoryOp)) refers to the state of the model before execution of the action and the expression self.allClasses() → select(a_specfactory).allOperations() = self.allClasses() → select(a_specfactory).allOperations()@pre → including(a_createpartop) references the source model in the postcondition. The focus of the postcondition is the state of the model after execution of the action. The OCL expression self.allClasses() → select(a_specfactory).allOperations() = self.allClasses() → select(a_specfactory).allOperations()@pre → including(a_createpartop) will be used in the composed postcondition since it shows that the state of the model has changed.

The pre- and postconditions, with z changed back to a_factory, for the composition of t (pre^post3') with the pre- and postconditions for Action4 are shown in Table 5.10.

Table 5.10. Composed pre^post3’ with Action 4 (pre4’post4’)

pre: true
post4’:
{ self.allClasses() = self.allClasses()@pre → including(a_factory) ^
  self.allClasses() → one(a_factory) ^
  self.allDependencies() = self.allDependencies()@pre → including(a_clientfactdepend) ^
  self.a_client → select(fc | fc.oclIsTypeOf(FactoryEnd)) = self.allClasses() → select(a_factory)^
  self.allClasses() → collect(a_factory) → select(ce| ce.oclIsTypeOf(ClientEnd)) = self.allClasses() → select(a_client)^
  self.a_client.a_factory.allOperations() = self.a_client.a_factory.allOperations()@pre → including(a_createpartop) ^
  self.a_client.a_factory.allOperations() → select(a_createpartop) → size() = CreatedProducts → size() ^
  self.allClasses() = self.allClasses()@pre → including(a_specfactory) ^
  self.allGeneralizations() = self.allGeneralizations()@pre → including(a_factoregen) ^
  self.a_client.a_factory → collect(a_specfactory) → size() = self.a_client@pre → select(a_createop) → size() ^
  self.a_client.a_factory.allChildren() = self.allClasses() → select(a_specfactory) ^
  self.allClasses() → select(a_specfactory).allParents() = self.allClasses() → select(a_factory)^
  self.allClasses() → select(a_specfactory).allOperations() = self.allClasses() → select(a_specfactory).allOperations()@pre → including(a_createpartop) ^
  self.allClasses() → select(a_specfactory).allOperations() → select(a_createpartop) → size() =
  CreatedProducts() → size() }
4. Compose pre- and postconditions (pre\textsuperscript{4'} post\textsuperscript{4'}) given in Table 5.10 with Action 5 pre- and postconditions (pre\textsuperscript{5} post\textsuperscript{5}) given in Table 5.6. Since there is no condition which restricts the execution of action 5, it is not necessary to rename variables or bind an existence quantifier.

- Simplify the expression
  - The model element \texttt{a\_createCompProd} is an instance of \texttt{CreateCompProd}. The OCL expressions
    \begin{itemize}
      \item self.a\_client.allOperations() \rightarrow excludes(oclIsTypeOf(\texttt{CreateCompProd})) and
      \item self.a\_client.allOperations() = self.a\_client.allOperations()@pre \rightarrow including(a\_createCompProd)
    \end{itemize}
    references the same model element. The expression self.a\_client.allOperations() \rightarrow
    excludes(oclIsTypeOf(\texttt{CreateCompProd})) refers to the state of the model before execution of the action, and the expression self.a\_client.allOperations() = self.a\_client.allOperations()@pre \rightarrow
    including(a\_createCompProd) references the source model in the postcondition. The focus of the postcondition is the state of the model after execution of the action. The OCL expression
    \begin{itemize}
      \item self.a\_client.allOperations() = self.a\_client.allOperations()@pre \rightarrow including(a\_createCompProd)
    \end{itemize}
    will be used in the composed postcondition since it shows that the state of the model has changed.

Table 5.11. Composed pre\textsuperscript{4'} with Action 5 (pre\textsuperscript{5} post\textsuperscript{5})

<table>
<thead>
<tr>
<th>pre: true</th>
</tr>
</thead>
<tbody>
<tr>
<td>post\textsuperscript{5'}:</td>
</tr>
</tbody>
</table>
| \{$self.allClasses() = self.allClasses()@pre \rightarrow including(a\_factory)^\wedge$
| \$self.allClasses() \rightarrow one(a\_factory)^\wedge$
| \$self.allDependencies() = self.allDependencies()@pre \rightarrow including(a\_clientfactdepend)^\wedge$
| \$self.a\_client \rightarrow select(fc \mid fc.oclIsTypeOf(FactoryEnd)) = self.allClasses() \rightarrow select(a\_factory)^\wedge$
| \$self.allClasses() \rightarrow collect(a\_factory) \rightarrow select(ce \mid ce.oclIsTypeOf(ClientEnd)) = self.allClasses() \rightarrow$
| \$select(a\_client)^\wedge$
| \$self.a\_client.a\_factory.allOperations() = self.a\_client.a\_factory.allOperations()@pre \rightarrow$
| \$including(a\_createpartop)^\wedge$
| \$self.a\_client.a\_factory.allOperations() \rightarrow select(a\_createpartop) \rightarrow size() = CreatedProducts \rightarrow size()$
| \$self.allGeneralizations() = self.allGeneralizations()@pre \rightarrow including(a\_factorygen)^\wedge$
| \$self.allClasses() = self.allClasses()@pre \rightarrow including(a\_specfactory)^\wedge$
| \$self.a\_client.a\_factory \rightarrow collect(a\_specfactory) \rightarrow size() = self.a\_client@pre \rightarrow select(a\_createop)$
| size()$\wedge$
| \$self.a\_client.a\_factory.a\_children() = self.allClasses() \rightarrow select(a\_specfactory)^\wedge$
| \$self.allClasses() \rightarrow select(a\_specfactory).allParents() = self.allClasses() \rightarrow select(a\_factory)^\wedge$
| \$self.allClasses() \rightarrow select(a\_specfactory).allOperations() = self.allClasses() \rightarrow$
| \$select(a\_specfactory).allOperations()@pre \rightarrow including(a\_createpartop)^\wedge$
| \$self.allClasses() \rightarrow select(a\_specfactory).allOperations() \rightarrow select(a\_createpartop) \rightarrow size() =$
| \$CreatedProducts() \rightarrow size()^\wedge$
| \$self.a\_client.allOperations() = self.a\_client.allOperations()@pre \rightarrow including(a\_createCompProd)^\wedge$
| and self.a\_client.allOperations() \rightarrow one(a\_createCompProd) \}$
The pre- and postconditions for the composition of (pre‘post4’) with the pre- and postconditions for Action 5 are given in Table 5.11.

5. Compose (pre‘post4’) given in Table 5.10 with Action 6 pre- and postconditions (pre6‘post6) given in Table 5.7. Since no conditions exist which restricts the execution of action 6, it is not necessary to rename a variable or bind an existence quantifier. The pre- and postconditions for the composition of (pre‘post5’) with the pre- and postconditions for Action 6 are shown in Table 5.12.

Table 5.12. Composed pre‘post5’ with Action 6 (pre6‘post6)’

<table>
<thead>
<tr>
<th>pre: true</th>
</tr>
</thead>
<tbody>
<tr>
<td>post6’:</td>
</tr>
</tbody>
</table>
| { self.allClasses() = self.allClasses()@pre → including(a_factory) ^  
  self.allClasses() → one(a_factory) ^  
  self.allDependencies() = self.allDependencies()@pre → including(a_clientfactdepend) ^  
  self.a_client → select(fc | fc.oclIsTypeOf(FactoryEnd)) = self.allClasses() → select(a_factory)^  
  self.allClasses() → collect(a_factory) → select(ce| ce.oclIsTypeOf(ClientEnd)) = self.allClasses() → select(a_client)^  
  self.a_client.a_factory.allOperations() = self.a_client.a_factory.allOperations()@pre →  
  including(a_createpartop) ^  
  self.a_client.a_factory.allOperations() → select(a_createpartop) → size() = CreatedProducts → size() ^  
  self.allGeneralizations() = self.allGeneralizations()@pre → including(a_factorygen) ^  
  self.allClasses() = self.allClasses()@pre → including(a_specfactory)^  
  self.a_client.a_factory → collect(a_specfactory) → size() = self.a_client@pre → select(a_createop) → size() ^  
  self.a_client.a_factory.allChildren() = self.allClasses() → select(a_specfactory) ^  
  self.allClasses() → select(a_specfactory).allParents() = self.allClasses() → select(a_factory) ^  
  self.allClasses() → select(a_specfactory).allOperations() = self.allClasses() → select(a_specfactory).allOperations()@pre → including(a_createpartop) ^  
  self.allClasses() → select(a_specfactory).allOperations() → select(a_createpartop) → size() = CreatedProducts() → size() ^  
  self.a_client.allOperations() = self.a_client.allOperations()@pre → including(a_createCompProd) ^  
  and self.a_client.allOperations() → one(a_createCompProd)^  
  self.a_client.allOperations() = self.a_client.allOperations()@pre → excluding(a_createop) } |

Composing pre‘post6 with the precondition of the transformation (i.e., self.isValidSource(AFMetamodel)) yields the precondition, postcondition pair (preModel‘postModel) shown in Table 5.13. The composition of pre: true with the precondition self.isValidSource(AFMetamodel) produces the final preconditions preModel:

self.isValidSource(AFMetamodel).

Table 5.13. Model-Level Pre- and Postcondition (preModel‘postModel)

<table>
<thead>
<tr>
<th>preModel:</th>
</tr>
</thead>
<tbody>
<tr>
<td>self.isValidSource(AFMetamodel)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>postModel:</th>
</tr>
</thead>
</table>
| { self.allClasses() = self.allClasses()@pre → including(a_factory) ^  
  self.allClasses() → one(a_factory) ^  
  self.allDependencies() = self.allDependencies()@pre → including(a_clientfactdepend) ^ |
To validate the AF transformation, the model-level transformation specification must be verified against the metamodel-level transformation pattern. This verification process requires the model-level preconditions and postconditions, (preModel and postModel) conform to preconditions and postconditions (preMeta and postMeta) specified by the transformation pattern. Figure 5.7 depicts the current stage in the approach to pattern-based model transformation illustrated in Figure 1-3.

The model elements of the source model are instances of the metamodel elements of the AFMetamodel, therefore the model-level pre- and postcondition (preModel and postModel) pairs imply the metamodel-level pre- and postcondition (preMeta and postMeta) pairs.

The diagrams in Figure 5.8 thru Figure 5.12 show the mappings between the model-level postconditions and the postconditions expressed in the transformation pattern. As stated previously, the transformation schema, combined with the transformation constraint, represents the postcondition of the AF transformation which indicates the model elements added, deleted and connected during the transformation. The OCL expressions in the “note” boxes represent the constraints specified by the postconditions at the model-level, and the dashed lines illustrate the mapping between the model-level postcondition to the metamodel-level postcondition. For example, in Figure 5.8,
the postcondition `self.allClasses() \rightarrow \text{one(a}\_\text{factory})` which specifies that the collection of classes contained in the model can have only one a\_factory class as restricted by the realization multiplicity value of 1 on the |Factory role.

Figure 5.7. Model to Metamodel Level Validation of Pre- and Postconditions

The “shaded” constraints (“note” boxes) attached to the elements in the transformation constraint are the constraint templates (i.e., restrictions) that can not be represented graphically in the transformation schema but are required to enforce the relationship between the model elements as specified by the AF pattern. In Figure 5.9, the constraint template for CreatedProducts is attached to the CreatedProducts derived relationship shown in the transformation constraint. The mappings illustrate how the constraint templates relate to the graphical expression given in the transformation constraint.
Figure 5.8. M1 to M2 Postcondition Mapping for Add Factory Class

Figure 5.9. M1 to M2 Mapping for Add Factory Operations
Figure 5.10. M1 to M2 Postcondition Mapping for Add Specialized Factory Class

Figure 5.11. M1 to M2 Postcondition Mapping for Add SpecializedFactory Operations
Figure 5.12. M1 to M2 Postcondition Mapping for Client Operations

5.6 Structural Conformance

The model shown in Figure 5.2 structurally conforms to the source schema given in Figure 5.6(a) with respect to bindings. This structural conformance is shown in Figure 5.13. In Figure 5.13, the dashed lines indicate that the MazeGame plays the role of a |Client; the MazeGame owns two behavioral features (CreateEnchantedMaze and CreateBombMaze) that play the role of |CreateOp; the |CompositeProduct role is played by Maze; and the classes Door, Wall, and Room play the role of SubProduct.

Figure 5.13. Structurally Conforming MazeGame Source Model
The conforming parts of the model given in Figure 5.14 are the metamodel representation of the source model (i.e., the MazeGame object model). In the object model, each metamodel element is depicted as an object. This allows us to manually illustrate the transformation process.

Figure 5.14. MazeGame Source Object Model

The AF transformation pattern characterizes a family of model-level transformations. The following steps represent an example of the transformation to introduce the AF design pattern into the MazeGame source model. The source model conforms to the source schema defined in the AF transformation pattern. Model elements which are added during the transformation are represented by bolded text and shaded boxes.

Figure 5.15. Add Factory Class
The first step in the transformation process is to create a factory class (instances of |Factory), and link the factory class to the client class (instance of |Client). As shown in Figure 5.15, the MazeFactory class is created as an instance of the |Factory role. The diagram also shows that an instance of |ClientFactDependency (ClntFactDep) is created with the client dependency end linked to MazeGame, and the supplier dependency end is linked to MazeFactory.

![Figure 5.16. Add Specialized Factory Classes](image)

We create Specialized Factory classes (instances of |SpecializedFactory) for each instance of |CreateOp owned by the MazeGame class (instance of |Client). The specialized factory classes are connected to the factory class via Generalization (instance of |FactoryGeneralization) relationship. As illustrated in Figure 5.16, the MazeGame owns two operations that are instances of |CreateOp, CreateEnchantedMaze and CreateBombedMaze. The CreateBombedMaze operation corresponds to the creation of the BombedMazeFactory and the CreateEnchantedMaze operation corresponds to the creation of the EnchantedMazeFactory as specified by the transformation constraint shown in Figure 5.6(c). An instance of |FactoryGeneralization, FactGen, is also created. The general end of the FactGen is connected to the parent, MazeFactory, while the specific end is connect to the children (i.e., EnchantedMazeFactory and BombedMazeFactory).

The third next step involves adding create product operations (instances of |CreatePartOp) for each product in the collection of created products (i.e., create operation created for room, door, wall). The create product operations (instances of |CreatePartOp) are connected to each specialized factory (instance of |SpecializedFactory). In the diagram shown in Figure 5.17, create product operations (CreateMaze, CreateRoom, CreateDoor, CreateWall) are
created for each product part associated with the CreateBombedMaze and CreateEnchantedMaze operation owned by the MazeGame class. The create product operations are linked to the BombedMazeFactory and EnchantedMazeFactory classes.

Figure 5.17. Add CreatedProducts to Specialized Factory Classes

The fourth step involves adding CreatedProducts (instances of |CreatePartOp) operations and connecting the operations to the instance of |Factory. In Figure 5.18, a create operations (CreateMaze, CreateRoom, CreateWall, CreateDoor) are connected to MazeFactory class.

Figure 5.18. Add CreatedProducts to Factory Classes
An operation that creates a composite product (instance of `CreateCompProd`) is introduced to the model and connected to the instance of `Client`. In Figure 5.19, the makeMaze operation (instance of `CreateCompProd`) is added and connected to the MazeFactory (instance of `Client`).
The last step in the transformation process deletes the create operations (instances of |CreateOp) that are owned by the instances of the |Client from the model. As shown in Figure 5.20, all create operations (CreateEnchantedMaze and CreateBombedMaze) owned by the MazeGame are deleted from the model such that CreateEnchantedMaze and CreateBombedMaze are not elements linked to the MazeGame.

![Figure 5.21. MazeGame Target Model](image1)

![Figure 5.22. Structurally Conforming MazeGame Target Model](image2)
The diagram in Figure 5.20 is also a representation of the target object model. The target object model resulted from applying the above procedure to the MazeGame source object model given in Figure 5.2. Figure 5.21 shows the transformed target model at the model-level produced by applying the AF transformation specification to the MazeGame source model.

It is a straightforward task to check that the transformation steps outlined above adhered to the transformation schema and the transformation constraint for the AF transformation pattern. This diagram shown in Figure 5.22 illustrates the structural conformance between the target model and the AF transformation pattern.

5.7 Summary

The characterization of creational transformations was illustrated using the AF pattern. By using the AF pattern as an example, we have shown that a precise definition of the transformation pattern can provide clear indicators of design changes needed order to incorporate a pattern.
Chapter 6
Characterizing Behavioral Pattern Transformations

This chapter describes the transformation pattern defined for a design pattern categorized as a Behavioral pattern – the Visitor design pattern. Section 6.1 provides an overview the Visitor design pattern. Section 6.2 describes the parts of the Visitor transformation pattern. Section 6.3 describes the Visitor transformation specification. Validation of the Visitor transformation pattern is provided using two different approaches. Section 6.4 provides a formal method of showing conformance between the model-level transformation and the metamodel-level transformation specification, and Section 6.5 presents an informal technique. Section 6.6 summarizes the Visitor transformation pattern.

6.1 The Visitor Pattern

The Visitor pattern is a behavioral design pattern that allows software developers “to represent an operation that is to be performed on the elements of an object structure and to define new operations without changing the classes of the elements on which they operates” [Gamma et al., 1994]. This activity is useful when distinct and unrelated operations need to be performed on objects in a object structure but defining operations together in one class would pollute the classes. The general idea of the Visitor pattern is to separate the structure of elements (classes) from the operations that can be applied on these elements and to package them into a separate object called a visitor.

![Visitor Design Pattern](image-url)

Figure 6.1. Visitor Design Pattern [Gamma et al., 1994]
Figure 6.1 [Gamma et al., 1994] graphically depicts the Visitor pattern. The Visitor design pattern consists of two types of visitors, ConcreteVisitor1 and ConcreteVisitor2. The visitor visits an instance of ObjectStructure, which consists of two types of elements, ConcreteElementA and ConcreteElementB. The Visitor pattern defines a Visitor as a class that implements the “visit” methods and Element as a class that implements a method called “accept”. When an element “accepts” the visitor (i.e., an element performs the operation defined by the visitor), the element sends an “element specific” message to the visitor, passing itself (i.e., the element) as an argument. The visitor will then execute the operation for that specific element in response to the message.

6.2 Visitor Transformation Pattern

The specification of the Visitor transformation pattern is given in Figure 6.2. The three parts of the Visitor transformation pattern are defined below.

![Figure 6.2. Visitor Transformation Pattern](image)

6.2.1 Source Schema

The source schema, given in Figure 6.2(a), consists of classifier roles, |Client and |ObjectStructure, linked to each other by the |ClientObjStructAssoc association role; a classifier role, |Element, that is connected to the |ObjectStructure role via the |ObjElemAssoc association role, and a |ConcreteElement classifier role which is a
specialization of |Element, as indicated by the generalization role, |ElemGeneralization, that links |Element to |ConcreteElement.

The source schema specifies that a conforming source model must have exactly one class that conforms to |Client, exactly one class that plays the role of |ObjectStructure, exactly one class that conforms to the |Element, and at least one class that plays the role of |ConcreteElement as indicated by the realization multiplicities. The |Client role is mandatory structure whose instances are classes that represent clients and the and |ObjectStructure is a mandatory structure whose instances represent an interface between the client and its elements. The association role |ClientObjStructAssoc specifies the association between |Client and |ObjectStructure. Each conforming |ClientObjStructAssoc association must have one association-end (|Clnt) connected to the |Client class and the other association-end (|Obj2Clnt) connected to the |ObjectStructure class. The realization multiplicity on the |Clnt association-end role specifies that the instance of |Client must be a part of only one |ClientObjStructAssoc association. The realization multiplicity on the |Obj2Clnt association-end role specifies that the instance of |ObjectStructure must be a part of only one |ClientObjStructAssoc association.

Classes that conform to |ObjectStructure must be associated with an |Element role via an |ObjElemAssoc association role. Each conforming |ObjElemAssoc association must have one association-end (|Obj2Elem) connected to the |ObjectStructure class and the other association-end (|Elem2Obj) connected to the |Element class. The realization multiplicity on the |Obj2Elem role specifies that an |ObjectStructure class must be a part of only one |ObjElemAssoc association. The realization multiplicity on the |Elem2Obj role specifies that an |Element class must be a part of only one |ObjElemAssoc association.

Classes that conform to the |Element and |ConcreteElement roles must consist of at least one operation that plays the |ElemOp behavioral feature role. An instance of |Element is a generalization of the classifier role, |ConcreteElement, that is, at least one |ConcreteElement is a specialization of one |Element as specified by the generalization role, |ElemGeneralization. The multiplicity (1..*) of |ElemOp specifies that the |Element and |ConcreteElement must have at least one instance of |ElemOp. In a conforming source model, the general end of |ElemGeneralization that is connected to |Element must conform to the |ParentElem and the specific end that is connected to |ConcreteElement must conform to the |ChildElem. The realization multiplicity on the |ParentElem role specifies that an |Element class must be a part of at least one |ElemGeneralization generalization role. The
realization multiplicity on the |ChildElem role specifies that |ConcreteElement class must be a part of one |ElemGeneralization generalization role.

The metamodel-level constraints for the source schema association role are defined as follows:

- An association-end that conforms to |Clnt must have a multiplicity of 1..1:
  
  ```
  context |Clnt
  inv self.lowerBound() = 1 and self.upperBound() = 1
  ```

- A conforming |Obj2Clnt association-end must have a multiplicity of 1..1:
  
  ```
  context |Obj2Clnt
  inv self.lowerBound() = 1 and self.upperBound() = 1
  ```

- An association-end that conforms to |Obj2Elem must have a multiplicity of 1..1:
  
  ```
  context |Obj2Elem
  inv self.lowerBound() = 1 and self.upperBound() = 1
  ```

- A conforming |Elem2Obj association-end must have a multiplicity of 1..1:
  
  ```
  context |Elem2Obj
  inv self.lowerBound() = 1 and self.upperBound() = 1
  ```

- A conforming |ParentElem general end must have a multiplicity of 1..*:
  
  ```
  context |ParentElem
  inv self.lowerBound() = 1 and self.upperBound() = *
  ```

- A conforming |ChildElem specific end must have a multiplicity of 1..1:
  
  ```
  context |ChildElem
  inv self.lowerBound() = 1 and self.upperBound() = 1
  ```

6.2.2 Transformation Schema

The transformation schema specifies that the Visitor pattern transformation introduces a visitor class (instance of |Visitor), visit operations (instances of |VisitOp) owned by the visitor class, and accept operations (instances of |AcceptOp) owned by the instances of |ConcreteElement.

In the transformation schema, a |Visitor role consisting of at least one visit operation (instances of |VisitOp) is introduced into the structure and connected to the instance of |Client via a |ClientVisitorAssoc association role. The realization multiplicity of the |Visitor role indicates there must be one class that plays the |Visitor role in a conforming target model. The association role |ClientVisitorAssoc specifies the associations between classes that play the role of |Client and the class that plays the role of |Visitor. Each conforming |ClientVisitorAssoc association must have one association-end (|Clnt2Visi)t connected to the |Client role and the other association-end (|Visit2Clnt)
connected to the |Visitor role. The realization multiplicities on the |Clnt2Visit and |Visit2Clnt association-end roles specify that |Client and |Visitor must be a part of only one |ClientVisitorAssoc association.

The transformation schema also introduces a |ConcreteVisitor consisting of at least one visit operation (instances of |VisitOp) into the structure as a specialization of the |Visitor. The |ConcreteVisitor realization multiplicity of (1..*) specifies that a conforming target model must contain at least one class that conforms to the |ConcreteVisitor. In the transformed model, the visitor declares the visit operations (instances of |VisitOp) for each class of concrete elements in the object structure, and the concrete visitors implements each operation (instance of |VisitOp) declared by visitor.

Instances of the |ConcreteVisitor are connected to the |Visitor role by the |VisitorGeneralization generalization role. In a conforming target model, the end of |VisitorGeneralization (general end) connected to an instance of the |Visitor role must conform to |ParentVisit, and the end of |VisitorGeneralization (specific end) connected to the |ConcreteVisitor must conform to |ChildVisit. The 1..* realization multiplicity on the |ParentVisit role specifies that a |Visitor must be part of at least one |VisitorGeneralization. The realization multiplicity (1..1) on the |ChildVisit role specifies that a |ConcreteVisitor can be part of only one |VisitorGeneralization.

The transformation also introduces exactly one behavioral feature role (instance of the |AcceptOp role) into classes that play the |Element and |ConcreteElement roles. Instances of the |ElemOp role are removed from the conforming |Element and |ConcreteElement classes.

The metamodel-level constraints for the transformation schema are defined as follows:

- An association-end that conforms to |Clnt2Visit must have a multiplicity of 1..1:

  ```
  context |Clnt2Visit
  inv self.lowerBound() = 1 and self.upperBound() = 1
  ```

- A conforming |Visit2Clnt association-end must have a multiplicity of 1..1:

  ```
  context |Visit2Clnt
  inv self.lowerBound() = 1 and self.upperBound() = 1
  ```

- A general end that conforms to |ParentVisit must have a multiplicity of 1..*:

  ```
  context |ParentVisit
  inv self.lowerBound() = 1 and self.upperBound() = *
  ```

- A specific end that conforms to |ChildVisit must have a multiplicity of 1..1:

  ```
  context |ChildVisit
  inv self.lowerBound() = 1 and self.upperBound() = *
  ```
The constraint templates as specified in [Kim 2004] for \(\text{VisitOp}\) and \(\text{AcceptOp}\) behavioral feature roles in a conforming target model are given as follows:

- An \(\text{AcceptOp}\) operation lets a “Visitor” object visit a given element by invokes the \(\text{VisitOp}\) operation:
  
  \[
  \text{context } \text{ConcreteElement} :: \text{AcceptOp}(\text{vis} : \text{ConcreteVisitor}) : \text{OclMessage} \\
  \text{pre: true} \\
  \text{post: let elementMessage : OCLMessage = } \text{ConcreteVisitor}^\wedge\text{VisitOp}(\text{elem}) \rightarrow \text{notEmpty()}
  \]

- An \(\text{VisitOp}\) operation invokes an operation call:

  \[
  \text{context } \text{Visitor} :: \text{VisitOp} (\text{elem} : \text{ConcreteElement}) \\
  \text{pre: true} \\
  \text{post: let visitMessage : OclMessage = } \text{ConcreteElement}^\wedge\text{Operation} (\text{id}) \rightarrow \text{notEmpty()}
  \]

### 6.2.3 Transformation Constraint

There are two transformation constraints, as shown in Figure 6.2(c), defined by the Visitor transformation pattern.

The concrete visitor constraint specifies that a concrete visitor class (instance of \(\text{ConcreteVisitor}\)) should be created for every element operation (instance of \(\text{ElemOp}\)) owned by instances of \(\text{Element}\) in the source schema. This constraint implies that the number of classes that play the \(\text{ConcreteVisitor}\) role must equal the number of element operations (instances of \(\text{ElemOp}\)) in the collection of operations owned by the classes that play the \(\text{Element}\) and \(\text{ConcreteElement}\) roles. The OCL expression for this constraint is given as follows:

\[
\text{context } \text{Client} \\
\text{inv: self.} \forall \text{Visitor} \rightarrow \text{select(\text{ConcreteVisitor}) } \rightarrow \text{size()} = \text{self.} \forall \text{ObjectStructure} \rightarrow \text{collect(oclIsTypeOf(\text{Element})) } \\
\rightarrow \text{select(oclIsTypeOf(\text{ElemOp})) } \rightarrow \text{asSet()} \rightarrow \text{size()}
\]

The visitor operation constraint specifies that there exists a visitor operation for each concrete element in the source schema. This constraint implies that the number of visitor operations (instances of \(\text{VisitOp}\)) must equal the number of concrete element classes (instances of \(\text{ConcreteElement}\)). The OCL expression for this constraint is given as follows:

\[
\text{context } \text{Client} \\
\text{inv: self.} \forall \text{Visitor} \rightarrow \text{collect(\text{ConcreteVisitor}) } \rightarrow \text{select(oclIsTypeOf(\text{VisitOp})) } \rightarrow \text{asSet()} \rightarrow \text{size()} = \text{self.} \forall \text{ObjectStructure} \rightarrow \text{select(oclIsTypeOf(\text{ConcreteElement})) } \rightarrow \text{size()}
\]

### 6.2.4 Metamodel-Level Pre- and Postconditions

At the metamodel (M2) level, the transformation specification for the Visitor pattern, referred to as VisitorMetaTransformationSpec, is specified by the pre- and postconditions defined in the Visitor transformation pattern. The Visitor source schema depicts the metamodel-level precondition, referred to as preMeta, which must be
satisfied before a visitor transformation can execute. The postconditions are specified by the visitor transformation schema, along with the restrictions of the visitor transformation constraints. These postcondition at the metamodel level are referred to as postMeta.

6.3 Visitor Pattern Transformation Specification

A Visitor pattern transformation specification (i.e., program) defines the sequence of actions required to introduce the Visitor pattern into an existing UML model to create a new model instantiated with the Visitor pattern. The transformation specification is expressed using the PBAL action language. The transformation program for the Visitor transformation pattern is given in Table 6.1. The Visitor transformation specification is defined for a package consisting of a source model (i.e., the context of the VisitorTransformation is the source model upon which the transformation is applied). The precondition, isValidSource(metamodel), verifies that the source model conforms to the metamodel as specified by the source schema structure in the Visitor Transformation pattern, shown in Figure 6.2. If the source model conforms to the source schema, the operation isValidSource returns true and the transformation of the model can proceed. A Visitor pattern transformation involves the following steps:

1. **Create Visitor Class.** A visitor class (instances of |Visitor) is created and then connected to the client class (instance of the |Client).

2. **Create Concrete Visitor Classes.** Concrete visitor classes (instances of |ConcreteVisitor) are created for each element operation (instance of |ElemOp) owned by the concrete element classes (instances of |ConcreteElement). A visitor generalization (instance of |VisitorGeneralization) is created and the general end is connected to the visitor class. The specific end of the visitor generalization is connected to each concrete visitor class.

3. **Create Visitor Operations.** This step involves creating a set of visit operations (instances of |VisitOp) corresponding to each concrete element class (instance of |ConcreteElement) in the collection of classes for the visitor class and each concrete visitor class. These operations are linked to the visitor class (instance of Visitor) and concrete visitor classes (instances of ConcreteVisitor) such that the operations (instances of |VisitOp) become owned operations of those classes (instances of |Visitor and |ConcreteVisitor).

4. **Create “Accept” Operation.** This step creates an “accept” operation (instances of |AcceptOp) and connects the operations to visitor class and each concrete visitor class.
5. **Delete Element Operations.** This step deletes all instances of |ElemOp from the element class and each concrete element class.

<table>
<thead>
<tr>
<th>Table 6.1. Visitor Transformation Program</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context</strong> Package :: VisitorTransformation (VisitorMetamodel)</td>
</tr>
<tr>
<td><strong>pre:</strong> isValidSource(VisitorMetamodel)</td>
</tr>
<tr>
<td><strong>action:</strong> /* all variables are local therefore must be declared */</td>
</tr>
<tr>
<td>indx1 : Integer;</td>
</tr>
<tr>
<td>indx2 : Integer;</td>
</tr>
<tr>
<td>num_concElem : Integer;</td>
</tr>
<tr>
<td>num_elements : Integer</td>
</tr>
<tr>
<td>num_elemOps : Integer</td>
</tr>
<tr>
<td>num_concElemOps : Integer;</td>
</tr>
<tr>
<td>num_concVisitor : Integer</td>
</tr>
<tr>
<td>/* action1: Create a visitor class &amp; connect the visitor class to client class via Association Relationship */</td>
</tr>
<tr>
<td>Visitor a_visitor; //Visitor variable declaration</td>
</tr>
<tr>
<td>a_visitor ::= _create_instance(Visitor); //create an instance of the Visitor class</td>
</tr>
<tr>
<td>Client a_client; //Client variable declaration</td>
</tr>
<tr>
<td>a_client = _get_instances(Client); // get all instances of Client</td>
</tr>
<tr>
<td>ClientVisitorAssoc a_clientvisitassoc; //ClientVisitorAssoc variable declaration</td>
</tr>
<tr>
<td>a_clientvisitassoc ::= _create_instance(ClientVisitAssoc); //create instance of ClientVisitAssoc</td>
</tr>
<tr>
<td>_connectClasses_Association(a_clientvisitassoc a_client, a_visitor); // connect client class to visitor class via a Association relationship</td>
</tr>
<tr>
<td>/* action2: create concrete visitor classes for each element operation*/</td>
</tr>
<tr>
<td>ElemOp[ ] a_elemOp; // Element operation declaration</td>
</tr>
<tr>
<td>a_elemOp ::= _get_operations(a_concElement, ElemOp); // get all instances of ElemOp</td>
</tr>
<tr>
<td>num_elemOps = a_elemOp → size(); // get the number of “element” operations</td>
</tr>
<tr>
<td>ConcreteVisitor[ ] a_concVisitor; // ConcreteVisitor variable declaration</td>
</tr>
<tr>
<td>VisitorGeneralization a_visitGen; // create visitor generalization</td>
</tr>
<tr>
<td>a_visitGen ::= _create_instance(VisitorGeneralization); // create an instance of VisitorGeneralization</td>
</tr>
<tr>
<td>indx1 = 1;</td>
</tr>
<tr>
<td>while (indx1 &lt;= num_elemOps) do {</td>
</tr>
<tr>
<td>a_concVisitor[indx1] ::= _create_instance(ConcreteVisitor); //create an instance of ConcreteVisitor</td>
</tr>
<tr>
<td>_connectClasses_VisitorGeneralization(a_visitGen, a_visitor, a_concVisitor[indx1]); // connect concrete visitor to visitor via generalization</td>
</tr>
<tr>
<td>indx1 = indx1 + 1;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>/* action3: Create visit operation for each object in the collection of concrete element and add to the visitor and each concrete visitor class */</td>
</tr>
<tr>
<td>VisitOp[ ] a_visitop; // Visitor operation variable declaration</td>
</tr>
<tr>
<td>ConcreteElement[ ] a_concElement; // Element variable declaration</td>
</tr>
<tr>
<td>a_concElement ::= _get_instances(ConcreteElement); // get all instances of ConcreteElement</td>
</tr>
<tr>
<td>num_concElem = a_concElement → size(); // get the number of concrete elements classes</td>
</tr>
<tr>
<td>indx1 = 1;</td>
</tr>
<tr>
<td>while (indx1 &lt;= num_concElem) do {</td>
</tr>
<tr>
<td>a_visitop[indx1] ::= _create_instance(VisitOp); //create an instances of VisitOp</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
Table 6.1 continued

```c
_connect_Op2Class (a_visitor, a_visitop[indx1]); // connect the visit operation to the visitor class
indx1 = indx1 + 1;
}
num_concVisitor := a_concVisitor → size();
indx2 = 1
while (indx2 <= num_concVisitor)
  indx1 = 1;
  while (indx1 <= num_concElem) do {
    a_visitop[indx1] ::= _create_instance(VisitOp); // create an instances of VisitOp
    _connect_Op2Class (a_concVisitor[indx2], a_visitop[indx1]); // connect the visit operation to
    the concrete visitor class
    indx1 = indx1 + 1;
  }
  indx2 = indx2 + 1;
}
```

/* action4: Create instance of AcceptOp and connect to the element and concrete element classes */
Element a_element
AcceptOp[] a_acceptOp;
indx1 = 1
a_acceptOp[indx1] := _create_instance(AcceptOp); // create an instance of AcceptOp
_connect_Op2Class (a_concElement[indx2], a_acceptOp[indx1]); // connect accept operation to the
element class
indx2 = 1
while(indx2 <= num_concElem) {
  indx1 = indx1 + 1
  a_acceptOp[indx1] := _create_instance(AcceptOp); // create an instance of AcceptOp
  _create_link_acceptOpLink(a_element, a_acceptOp[indx1]); // connect accept operation to the
  concrete element class
  indx2 = indx2 + 1;
}

/* action5: Remove all instances of ElemOp*/
ElemOp[] a_elemOp;
a_elemOp ::= _get_operations(a_element, ElemOp); // get all instances of ElemOp
num_elemOps = a_elemOp → size(); // get the number of “element” operations
indx1 = 1;
while (indx1 <= num_elemOps) do { // remove element operations from Element class
  _destroy_instance( a_elemOp[indx1] )
  indx1 = indx1 + 1;
}
a_elemOp ::= _get_operations(a_concElement, ElemOp); // get all instances of ElemOp
num_concElemOps = a_elemOp → size(); // get the number of “element” operations
indx1 = 1;
while (indx1 <= num_concElemOps) do { // remove element operations from Element class
  _destroy_instance( a_elemOp[indx1] )
  indx1 = indx1 + 1;
}
```

6.3.1 Model-Level Pre- and Postconditions

The model-level transformation specification, given in Table 6.1, can be expressed as a sequence of action clauses that convey individual operations on model elements needed to perform AF model-level transformation.
Each action has a preconditions and postconditions that must be satisfied before and after the execution of an action. Tables 6.2 - 6.6 show the individual action clauses expressed with pre- and postconditions.

Table 6.2 specifies the model-level transformation pre- and postconditions for creating a visitor class (Action 1). The precondition for this action is true, since creating a product is the first action in the sequence and it is assumed that the source model is valid. The postcondition ensures that the (1) the collection of classes includes one visitor class, (2) the collection of association includes client visitor association, and (3) the client class (a_client) is connected to the visitor class (a_visitor) via the association relationship (a_clientvisit), where the |Clnt2Visit association-end is connected to a_client and the Visit2Clnt association-end is connected to a_visitor.

<table>
<thead>
<tr>
<th>Table 6.2. Action 1 - Create a Visitor Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context</strong> Package :: VisitorTransformation (VisitorMetamodel)</td>
</tr>
<tr>
<td><strong>pre1:</strong></td>
</tr>
<tr>
<td>true</td>
</tr>
<tr>
<td><strong>post1:</strong></td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses()@pre → including(a_visitor) // visitor class exist in collection of classes</td>
</tr>
<tr>
<td>and self.allClasses() → one(a_visitor) // only one visitor class created</td>
</tr>
<tr>
<td>and self.allAssociations = self.allAssociations@pre → including(a_clientvisit)</td>
</tr>
<tr>
<td>and self.a_client.Visit2Clnt = self.allClasses() → select(a_visitor)</td>
</tr>
<tr>
<td>and self.allClasses() → select(a_visitor).Clnt2Visit = self.a_client</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 6.3. Action 2 - Create Concrete Visitor Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context</strong> Package :: VisitorTransformation (VisitorMetamodel)</td>
</tr>
<tr>
<td><strong>pre2:</strong></td>
</tr>
<tr>
<td>self.allClasses() → exist(a_visitor)</td>
</tr>
<tr>
<td>and self.allClasses() → excludes(oclIsTypeOf(ConcreteVisitor))</td>
</tr>
<tr>
<td><strong>post2:</strong></td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses()@pre → including(a_concVisitor)</td>
</tr>
<tr>
<td>and self.allClasses() → collect(a_concVisitor) → size() = self.allClasses() → collect(a_concElement).allOperations()@pre → select(a_elementOp) → asSet() → size()</td>
</tr>
<tr>
<td>and self.allGeneralizations() = self.allGeneralizations()@pre → including(a_visitorgen)</td>
</tr>
<tr>
<td>and self.allGeneralizations() → collect(a_visitorgen).general = self.allClasses() → select(a_visitor)</td>
</tr>
<tr>
<td>and self.allGeneralizations() → collect(a_visitorgen).specific = self.allClasses() → collect(a_concVisitor).generalization</td>
</tr>
<tr>
<td>and self.a_client.a_visitor.allChildren() = self.allClasses()→ select(a_concVisitor)</td>
</tr>
<tr>
<td>and self.allClasses() → collect(a_concVisitor).allParents() = self.a_client.a_visitor</td>
</tr>
</tbody>
</table>

The pre- and postconditions, given Table 6.3, specifies the constraints on the action responsible for creating concrete visitor classes as specializations of the visitor class. The precondition specifies that the collection of classes must contain a visitor class, but no elements that are instances of |ConcreteVisitor. The postcondition specifies that (1) the collection of classes consists of concrete visitor classes; (2) the collection of generalizations contains visitor generalizations; (3) the general end of visitor generalization is linked to the visitor; (4) the specific
end of visitor generalization is linked to concrete visitor classes; (5) the children of visitor is the concrete visitor classes; and (6) the parents of concrete visitor classes is the visitor.

Table 6.4 provides the specification for the action that creates operations for the visitor class and each concrete visitor class. The precondition specifies that visitor and concrete visitor classes should exist in the collection of classes such that they are elements of the transformed model, but neither the visitor class nor the concrete visitor classes own visit operations that are instances of the |VisitOp. The postcondition states that (1) the visitor and concrete visitor classes must include operations that are instances of |VisitOp, (2) the number of visit operations (instances of |VisitOp) in the visitor classes must equal the number of concrete element classes in the source model; and (3) the number of visit operations (instances of |VisitOp) in the concrete visitor classes must equal the number of concrete element classes in the source model.

Table 6.4. Action 3 - Create Visit Operations

<table>
<thead>
<tr>
<th>context Package :: VisitorTransformation (VisitorMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre3:</td>
</tr>
<tr>
<td>self.allClasses() → exist(a_visitor)</td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses()@pre → including(a_concVisitor)</td>
</tr>
<tr>
<td>self.a_client.a_visitor → excludes(oclIsTypeOf(VisitOp))</td>
</tr>
<tr>
<td>self.allClasses() → collect(a_concVisitor) → excludes(oclIsTypeOf(VisitOp))</td>
</tr>
<tr>
<td>post3:</td>
</tr>
<tr>
<td>self.a_client.a_visitor.allOperations() = self.a_client.a_visitor.allOperations()@pre →</td>
</tr>
<tr>
<td>including(a_visitop))</td>
</tr>
<tr>
<td>and self.allClasses() → collect(a_concVisitor).allOperations() = self.allClasses() →</td>
</tr>
<tr>
<td>collect(a_concVisitor).allOperations()@pre → including(a_visitop))</td>
</tr>
<tr>
<td>and self.a_client.a_visitor → select(a_visitop) → size() = self.allClasses() → select(a_concElement)</td>
</tr>
<tr>
<td>→ size()</td>
</tr>
<tr>
<td>and self.allClasses() → collect(a_concVisitor) → select(a_visitop) → asSet() → size() =</td>
</tr>
<tr>
<td>self.allClasses() → select(a_concElement) → size()</td>
</tr>
</tbody>
</table>

Table 6.5 contains pre- and postconditions for the action responsible for adding an operation of type |AcceptOp to the instances of |Element and |ConcreteElement. The precondition ensures that (1) element and concrete element classes are objects in the collection of classes and (2) Element and ConcreteElement classes do not contain “accept” operations (instances of |AcceptOp). The postcondition verifies that accept operations have been added to element and concrete element classes.

The transformation specification in Table 6.6 removes all instances of operations (instances of |ElemOp) from the element and concrete element classes. The postcondition specifies that all element operations (instances of |ElemOp) are removed from the element and concrete element classes.
Table 6.5. Action 4 - Create a Accept Operations

context Package :: VisitorTransformation (VisitorMetamodel)
pre4:
self.allClasses() \rightarrow \text{exist(oclIsTypeOf(Element))}
and self.allClasses() \rightarrow \text{exist(oclIsTypeOf(ConcreteElement))}
and self.allClasses() \rightarrow \text{collect(a_element).allOperations() \rightarrow excludes(oclIsTypeOf(AcceptOp))}
and self.allClasses() \rightarrow \text{collect(oclIsTypeOf(ConcreteElement)) \rightarrow allOperations() \rightarrow excludes(oclIsTypeOf(AcceptOp))}

post4:
self.allClasses() \rightarrow \text{collect(a_element).allOperations()} = self.allClasses() \rightarrow
collect(a_element).allOperations()@pre \rightarrow including(a_acceptOp)
and self.allClasses() \rightarrow \text{collect(a_concElement).allOperations()} = self.allClasses() \rightarrow
collect(a_concElement).allOperations()@pre \rightarrow including(a_acceptOp)

Table 6.6. Action 5 - Remove ElemOp from Element and Concrete Element

context Package :: VisitorTransformation (VisitorMetamodel)
pre5: -- none
post5:
self.allClasses() \rightarrow \text{collect(a_element).allOperations()} = self.allClasses() \rightarrow
collect(a_element).allOperations()@pre \rightarrow excluding(a_elemOp)
and self.allClasses() \rightarrow \text{select(a_concElement).allOperations() = self.allClasses() \rightarrow
select(a_concElement).allOperations()@pre () \rightarrow excluding(a_elemOp)}

6.3.2 Composition of Visitor Model-Level Pre- and Postconditions

A single precondition postcondition pair, referred to as preModel^postModel, is obtained by composing each individual pair, given in Tables 6.2 - 6.6, two at a time. The character ^ represents a “logical and”.

1. Composing Action 1 pre- and postconditions (pre1^post1) with Action 2 pre- and postconditions (pre2^post2).
   - Rename variables that affect the occurrence in post1 and pre2
     - a_visitor \Rightarrow z

     The OCL statements for pre- and postconditions for actions 1 and 2 are renamed as follows:

     post1: \{ self.allClasses() = self.allClasses()@pre \rightarrow including(z) ^ self.allClasses() \rightarrow one(z) ^
               self.allAssociations = self.allAssociations@pre \rightarrow including(a_clientvisit) ^
               self.a_client.Visit2Clnt = self.allClasses \rightarrow select(z) ^ self.allClasses() \rightarrow select(z).Clnt2Visit =
               self.a_client \}

     pre2: \{ self.allClasses() \rightarrow exist(z) ^ self.allClasses() \rightarrow excludes(oclIsTypeOf(ConcreteVisitor)) \}
   - Bind z to an existence quantifier.
\( \exists z \in \{ \text{self.allClasses() = self.allClasses()@pre} \rightarrow \text{including}(z)^\wedge \text{self.allClasses() \rightarrow one}(z)^\wedge \text{self.allAssociations = self.allAssociations@pre} \rightarrow \text{including(a_clientvisit)}^\wedge \text{self.a_client.Visit2Clnt = self.allClasses() \rightarrow select}(z)^\wedge \text{self.allClasses() \rightarrow exist}(z)^\wedge \text{excludes(oclIsTypeOf(ConcreteVisitor))} \wedge \text{self.allClasses() = self.allClasses()@pre} \rightarrow \text{including(a_concVisitor)}^\wedge \text{self.allGeneralizations() = self.allGeneralizations()@pre} \rightarrow \text{including(a_visitorgen)}^\wedge \text{self.a_client.a_visitor.generalization = self.allGeneralizations() \rightarrow collect(a_visitorgen)}^\wedge \text{self.allClasses() \rightarrow collect(a_concVisitor).generalization = self.allGeneralizations() \rightarrow collect(a_visitorgen)}^\wedge \text{self.a_client.a_visitor.allChildren() = self.allClasses() \rightarrow select(a_concVisitor)}^\wedge \text{self.allClasses() \rightarrow collect(a_concVisitor).allParents() = self.a_client.a_visitor} \}

Table 6.7. Action 1 (pre1^post1) Composed With Action 2 (pre2^post2)

<table>
<thead>
<tr>
<th>pre: true post2^</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{ self.allClasses() = self.allClasses()@pre \rightarrow including(a_visitor)^\wedge self.allClasses() \rightarrow one(a_visitor)^\wedge self.allAssociations = self.allAssociations@pre \rightarrow including(a_clientvisit)^\wedge self.a_client.Visit2Clnt = self.allClasses() \rightarrow select(a_visitor)^\wedge self.allClasses() \rightarrow select(a_visitor).Clnt2Visit = self.a_client^\wedge self.allClasses()@pre \rightarrow including(a_concVisitor)^\wedge self.allClasses() \rightarrow collect(a_concVisitor) \rightarrow size() = self.allClasses() \rightarrow collect(a_concVisitor).allOperations@pre \rightarrow select(a_elementOp) \rightarrow asSet() \rightarrow size()^\wedge self.allGeneralizations() = self.allGeneralizations()@pre \rightarrow including(a_visitorgen)^\wedge self.a_client.a_visitor.generalization = self.allGeneralizations() \rightarrow collect(a_visitorgen)^\wedge self.allClasses() \rightarrow collect(a_concVisitor).generalization = self.allGeneralizations() \rightarrow collect(a_visitorgen)^\wedge self.a_client.a_visitor.allChildren() = self.allClasses() \rightarrow select(a_concVisitor)^\wedge self.allClasses() \rightarrow collect(a_concVisitor).allParents() = self.a_client.a_visitor }</td>
<td></td>
</tr>
</tbody>
</table>

- Simplify the expression
  - The \( \text{self.allClasses() = self.allClasses()@pre} \rightarrow \text{including}(z) \) and \( \text{self.allClasses() \rightarrow exist}(z) \) are equivalent statements since both verify that the \( z \) is an element in the collection of classes.
  - In composing the pre- and postconditions, the concern is with the final state after the execution of the action. Since the OCL statement \( \text{self.allClasses() = self.allClasses()@pre} \rightarrow \text{including(a_concVisitor)} \) in the postcondition of Action 2 evaluates the collection of classes after execution and the OCL statement \( \text{self.allClasses() \rightarrow excludes(a_concVisitor)} \) in the precondition...
of Action 2 evaluates the collection prior to the action being executed. The state of the collection before execution does not need to be included in the result of the composition.

The pre- and postconditions, with \( z \) changed back to a\_visitor, for the composition of pre1\^post1 with pre2\^post2 are shown in Table 6.7.

2. Composing pre\^post2 with Action 3 pre- and postconditions (pre3\^post3).

This step involves composing the (pre\^post2) shown in Table 6.7 with the pre- and postconditions for Action 3 (pre3\^post3) in Table 6.4.

- Rename variables that affect the occurrence in post1 and pre2.
  - a\_visitor \( \Rightarrow z \)
  - a\_concVisitor \( \Rightarrow y \)

The OCL statements for pre- and postconditions for actions 1 and 2 are renamed as follows:

**post2**: \( \{ \text{self.allClasses()} = \text{self.allClasses()}\text{@pre} \rightarrow \text{including}(z) \} \) \( \land \) \( \text{self.allClasses()} \rightarrow \text{one}(z) \) \( \land \) \( \text{self.allAssociations} = \text{self.allAssociations}\text{@pre} \rightarrow \text{including(a\_clientvisi}\text{t}) \) \( \land \) \( \text{self.a\_client.Visit2Clnt} = \text{self.allClasses()} \rightarrow \text{select}(z) \) \( \land \) \( \text{self.allClasses()} = \text{self.allClasses()}\text{@pre} \rightarrow \text{including}(y) \) \( \land \) \( \text{self.allClasses()} \rightarrow \text{collect(a\_concVisitor)} \rightarrow \text{size()} = \text{self.allClasses()} \rightarrow \text{collect(a\_concElement).allOperations()}\text{@pre} \rightarrow \text{select(a\_elementOp)} \rightarrow \text{asSet()} \rightarrow \text{size()} \) \( \land \) \( \text{self.allGeneralizations()} = \text{self.allGeneralizations()}\text{@pre} \rightarrow \text{including(a\_visitorgen)} \) \( \land \) \( \text{self.a\_client.z.generalization} = \text{self.allGeneralizations()} \rightarrow \text{collect(a\_visitorgen)} \) \( \land \) \( \text{self.allClasses()} \rightarrow \text{collect(y).generalization} = \text{self.allGeneralizations()} \rightarrow \text{collect(a\_visitorgen)} \) \( \land \) \( \text{self.a\_client.a\_visitor.allChildren()} = \text{self.allClasses()} \rightarrow \text{select(y)} \) \( \land \) \( \text{self.allClasses()} \rightarrow \text{collect(y).allParents()} = \text{self.a\_client.z} \)

**pre3**: \( \{ \text{self.allClasses()} \rightarrow \text{exist}(x) \} \) \( \land \) \( \text{self.allClasses()} \rightarrow \text{exist}(y) \) \( \land \) \( \text{self.a\_client.z} \rightarrow \text{excludes(oclIsTypeOf(VisitOp))} \) \( \land \) \( \text{self.allClasses()} \rightarrow \text{collect(y)} \rightarrow \text{excludes(oclIsTypeOf(VisitOp))} \) 

- Bind \( y \) and \( z \) to an existence quantifier

\( \exists y, z \bullet \{ \text{self.allClasses()} = \text{self.allClasses()}\text{@pre} \rightarrow \text{including}(z) \} \) \( \land \) \( \text{self.allClasses()} \rightarrow \text{one}(z) \) \( \land \) \( \text{self.allAssociations} = \text{self.allAssociations}\text{@pre} \rightarrow \text{including(a\_clientvisi}\text{t}) \) \( \land \) \( \text{self.a\_client.Visit2Clnt} = \text{self.allClasses()} \rightarrow \text{select}(z) \) \( \land \) \( \text{self.a\_client.a\_visitor.allChildren()} = \text{self.allClasses()} \rightarrow \text{select}(z).\text{Clnt2Visit} = \)
Simplify the expression post2'^pre3^post3.

- The OCL expressions self.allClasses() = self.allClasses()@pre → including(z) and self.allClasses() → exist(z) are referencing the same model element in the collection of classes. The expression self.allClasses() = self.allClasses()@pre → including(z) will be used in the composed postcondition since the OCL @pre construct allows the model elements in the source model to be referenced in the postcondition.

- The OCL expressions self.allClasses() = self.allClasses()@pre → including(y) and self.allClasses() → exist(y) both evaluates the collection of classes to determine if the element is in the collection of classes. The expression self.allClasses() = self.allClasses()@pre → including(y) will be used in the composed postcondition since the OCL @pre construct allows the model elements in the source model to be referenced in the postcondition.

Changing the variables names back to their originally defined names:

- Changing the variable z back to a_visitor, the composed pre-and postconditions can be further simplified. The model element a_visitOp is an instance of VisitOp. The OCL expressions
self.a_client.a_visitor → excludes(oclIsTypeOf(VisitOp)) and

self.a_client.a_visitior.allOperations() = self.a_client.a_visitior.allOperations()@pre →

including(a_visitop) references the same model element. The expression self.a_client.a_visitor →
excludes(oclIsTypeOf(VisitOp)) refers to the state of the model before execution of the action and
the expression self.a_client.a_visitior.allOperations() =

self.a_client.a_visitior.allOperations()@pre → including(a_visitop) references the source model
in the postcondition. The focus of the postcondition is the state of the model after execution of the
action and the action adds the model element a_visitop to the collection of classes. The OCL
expression self.a_client.a_visitior.allOperations() = self.a_client.a_visitior.allOperations()@pre
→ including(a_visitop) will be used in the composed postcondition since it shows that the state of
the model has changed.

- The element a_visitor is an instance of |Visitor. The OCL expressions self.a_client.z →
excludes(oclIsTypeOf(VisitOp)) and self.a_client.a_visitior.allOperations() =

self.a_client.a_visitior.allOperations()@pre → including(a_visitop) references the same model
element. The expression self.a_client.z → excludes(oclIsTypeOf(VisitOp)) refers to the state of
the model before execution of the action and the expression self.a_client.a_visitior.allOperations()
= self.a_client.a_visitior.allOperations()@pre → including(a_visitop) references the source
model in the postcondition. The focus of the postcondition is the state of the model after
execution of the action. The OCL expression self.a_client.a_visitior.allOperations() =

self.a_client.a_visitior.allOperations()@pre → including(a_visitop) will be used in the composed
postcondition since it shows that the state of the model has changed.

- The variables y and a_concVisitor refer to the same element. The OCL expression self.allClasses()
→ collect(z) → excludes(oclIsTypeOf(VisitOp)) and self.allClasses() →

collect(a_concVisitor).allOperations() = self.allClasses() → collect(a_concVisitor)@pre →

including(a_visitop) references the same model element. The expression self.allClasses() →
collect(z) → excludes(oclIsTypeOf(VisitOp)) refers to the state of the model before execution of
the action and the expression self.allClasses() → collect(a_concVisitor).allOperations() =

self.allClasses() → collect(a_concVisitor)@pre → including(a_visitop) references the source
model in the postcondition. The focus of the postcondition is the state of the model after 
execution of the action (the postcondition). The OCL expression self.allClasses() → 
collect(a_concVisitor).allOperations() = self.allClasses() → collect(a_concVisitor)@pre → 
including(a_visitop) will be used in the composed postcondition since it shows that the state of the 
model has changed.

Thus the pre- and postcondition, changing z back to a_visitor and y renamed back to a_concVisitor, for the 
composition pre^post2'^pre3'^post3 is shown in Table 6.8.

Table 6.8. Compose pre^post2' with Action 3 (pre3'^post3)

| pre: true |
| post3': |

{ self.allClasses() = self.allClasses()@pre → including(a_visitor) ^  
  self.allClasses() → one(a_visitor) ^  
  self.allAssociations = self.allAssociations@pre → including(a_clientvisit) ^  
  self.a_client.Visit2Clnt = self.allClasses() → select(a_visitor) ^  
  self.allClasses() → select(a_visitor).Clnt2Visit = self.a_client ^  
  self.allClasses() = self.allClasses()@pre → including(a_concVisitor) ^  
  self.allClasses() → collect(a_concVisitor) → size() = self.allClasses() →  
  collect(a_concElement).allOperations@pre → select(a_elementOp) → asSet() → size() ^  
  self.allGeneralizations() = self.allGeneralizations@pre → including(a_visitorgen) ^  
  self.a_client.a_visitor.generalization = self.allGeneralizations() → collect(a_visitorgen) ^  
  self.allClasses() → collect(a_concVisitor).generalization = self.allGeneralizations() →  
  collect(a_visitorgen) ^  
  self.a_client.a_visitor.allChildren() = self.allClasses() → select(a_concVisitor) ^  
  self.allClasses() → collect(a_concVisitor).allParents() = self.a_client.a_visitor ^  
  self.a_client.a_visitor.allOperations() = self.a_client.a_visitor.allOperations@pre →  
  including(a_visitop)) ^  
  self.a_client.a_visitor.allOperations() = self.allClasses() →  
  collect(a_concVisitor).allOperations@pre → including(a_visitop)) ^  
  self.a_client.a_visitor → select(a_visitop) → size() = self.allClasses() → select(a_concElement) →  
  size() ^  
  self.allClasses() → collect(a_concVisitor) → select(a_visitop) → asSet() → size() = self.allClasses() →  
  select(a_concElement) → size() }  

3. Composing pre^post3' with pre4^post4.

This step involves composing pre^post3' shown in Table 6.8 with the pre- and postconditions for Action 4 
(pre4^post4) in Table 6.5.

- The class _a_concElement is an instance of Concrete Element. The variables a_concElement and 
ConcreteElement can both be renamed to x. The OCL statements for pre- and postconditions for 
actions 1 and 2 are renamed as follows:

  post3': { self.allClasses() = self.allClasses()@pre → including(a_visitor) ^  
    self.allClasses() →  
    one(a_visitor) ^  self.allAssociations = self.allAssociations@pre → including(a_clientvisit) ^  

self.a_client.Visit2Clnt = self.allClasses() → select(a_visitor) ^ self.allClasses() → 
select(a_visitor).Clnt2Visit = self.a_client ^ self.allClasses()@pre →
including(a_concVisitor) ^ self.allClasses() → collect(a_concVisitor) → size() = self.allClasses() 
→ collect(x).allOperations()@pre → select(a_elementOp) → asSet() → size() ^
sel.allGeneralizations() = sel.allGeneralizations()@pre → including(a_visitorgen) ^
self.a_client.a_visitor.generalization = sel.allGeneralizations() → collect(a_visitorgen) ^
self.allClasses() → collect(a_concVisitor).generalization = sel.allGeneralizations() →
collect(a_visitorgen) ^ self.a_client.a_visitor.allChildren() = self.allClasses() →
select(a_concVisitor) ^ self.allClasses() → collect(a_concVisitor).allParents() =
self.a_client.a_visitor ^ self.a_client.a_visitior.allOperations() =
self.a_client.a_visitior.allOperations(@pre → including(a_visitor)) ^ self.allClasses() →
collect(a_concVisitor).allOperations() = sel.allClasses() →
collect(a_concVisitor).allOperations(@pre → including(a_visitor)) ^ sel.a_client.a_visitor →
select(a_visitor) → size() = sel.a_client.oclIsTypeOf(ObjectStructure) → select(x) → size() ^
sel.allClasses() → collect(a_concVisitor) → select(a_visitor) → asSet() → size() =
self.a_client.oclIsTypeOf(ObjectStructure) → select(x) → size() }

pre4: { sel.allClasses() → exist(oclIsTypeOf(Element)) ^ sel.allClasses() → exist(oclIsTypeOf(x)) ^
sel.allClasses() → collect(a_element).allOperations() → excludes(oclIsTypeOf(AcceptOp)) ^
sel.allClasses() → collect(oclIsTypeOf(x)) → allOperations() → excludes(oclIsTypeOf(AcceptOp)) }

- Simplify the expression.
  - The OCL expressions sel.allClasses() = sel.allClasses(@pre → including(x) and sel.allClasses
() → exist(x) are equivalent statement since both verify that the x exist in the collection of classes.
  The expression sel.allClasses() = sel.allClasses(@pre → including(x) will be used in the
composed postcondition since the OCL @pre construct allows the model elements in the source
model to be referenced in the postcondition

- The OCL expressions

  sel.allClasses() → collect(a_element).allOperations() → excludes(oclIsTypeOf(AcceptOp))
and

\[
\text{self.allClasses() → collect(a_element).allOperations() = self.allClasses() → collect(a_element).allOperations()@pre → including(a_acceptOp)}
\]

references the same model elements. The expression \( \text{self.allClasses() → collect(a_element).allOperations() → excludes(oclIsTypeOf (AcceptOp))} \) refers to the state of the model before execution of the action and the expression \( \text{self.allClasses() → collect(a_element).allOperations()@pre → including(a_acceptOp)} \) references the source model in the postcondition. The focus of the postcondition is the state of the model after execution of the action. The OCL expression

\[
\text{self.allClasses() → collect(a_concElement).allOperations() = self.allClasses() → collect(a_concElement).allOperations()@pre → including(a_acceptOp)}
\]

will be used in the composed postcondition since it shows that the state of the model has changed.

- The OCL expressions

\[
\text{self.allClasses() → collect(a_concElement).allOperations() → excludes(oclIsTypeOf (AcceptOp))}
\]

and

\[
\text{self.allClasses() → collect(a_concElement).allOperations() = self.allClasses() → collect(a_concElement).allOperations()@pre → including(a_acceptOp)}
\]

references the same elements. The expression \( \text{self.allClasses() → collect(a_concElement).allOperations() → excludes(oclIsTypeOf (AcceptOp))} \) refers to the state of the model before execution of the action and the expression \( \text{self.allClasses() → collect(a_concElement).allOperations()@pre → including(a_acceptOp)} \) references the source model in the postcondition. The focus of the transformation is the state of the model after execution of the action (the postcondition) and the purpose of transformation a model is to restructure the source model, the OCL expression \( \text{self.allClasses() → collect(a_concElement).allOperations() = self.allClasses() → collect(a_concElement).allOperations()@pre → including(a_acceptOp)} \)
The pre- and postconditions are simplified as:

\[
\{ \text{true} \wedge \text{self.allClasses()} = \text{self.allClasses()} \wedge \text{self.a_client.a_visitor.generalization} = \text{self.allGeneralizations()} \} \wedge \text{self.a_client.a_visitor.visitor Gen.} = \text{self.allGeneralizations()} \]

Thus the pre- and postcondition for the composition \text{pre}^3 \text{post3} \text{pre}^4 \text{post4} is shown in Table 6.9.
Table 6.9. Compose \( \text{pre}^\text{post3}' \) with Action 4 (\( \text{pre4}^\text{post4} \))

<table>
<thead>
<tr>
<th>pre: true</th>
</tr>
</thead>
<tbody>
<tr>
<td>post4':</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{self.allClasses() = self.allClasses()@pre \rightarrow including(a_visitor) } \\
\text{self.allClasses() \rightarrow one(a_visitor) } \\
\text{self.allAssociations = self.allAssociations@pre \rightarrow including(a_clientvisit) } \\
\text{self.a_client.Visit2Clnt = self.allClasses()\rightarrow select(a_visitor) } \\
\text{self.allClasses() \rightarrow select(a_visitor).Clnt2Visit = self.a_client } \\
\text{self.allClasses() = self.allClasses()@pre \rightarrow including(a_concVisitor) } \\
\text{self.allClasses() \rightarrow collect(a_concVisitor)\rightarrow size() = self.allClasses() } \\
\text{collect(a_concElement).allOperations()@pre \rightarrow select(a_elementOp) \rightarrow asSet() \rightarrow size() } \\
\text{self.allGeneralizations() = self.allGeneralizations()@pre \rightarrow including(a_visitorgen) } \\
\text{self.a_client.a_visitor.generalization = self.allGeneralizations() \rightarrow collect(a_visitorgen) } \\
\text{self.allClasses() \rightarrow collect(a_concVisitor).generalization = self.allGeneralizations() \rightarrow } \\
\text{collect(a_visitorgen) } \\
\text{self.a_client.a_visitor.allChildren() = self.allClasses()\rightarrow select(a_concVisitor) } \\
\text{self.allClasses() \rightarrow collect(a_concVisitor).allParents() = self.a_client.a_visitor } \\
\text{self.a_client.a_visitor.allOperations() = self.a_client.a_visitor.allOperations()@pre \rightarrow } \\
\text{including(a_visitop)) } \\
\text{self.allClasses() \rightarrow collect(a_concVisitor).allOperations() = self.allClasses() \rightarrow } \\
\text{collect(a_concVisitor).allOperations()@pre \rightarrow including(a_visitop)) } \\
\text{self.a_client.a_visitor \rightarrow select(a_visitop) \rightarrow size() = self.allClasses() \rightarrow select(a_concElement) \rightarrow } \\
\text{size()} \\
\text{self.allClasses() \rightarrow collect(a_concVisitor)\rightarrow select(a_visitop) \rightarrow asSet() \rightarrow size() = self.allClasses() } \\
\text{self.allClasses() \rightarrow select(a_concElement) \rightarrow size()} \\
\text{self.allClasses() \rightarrow exist(oclIsTypeOf(Element)) } \\
\text{self.allClasses() \rightarrow exist(oclIsTypeOf(ConcreteElement)) } \\
\text{self.allClasses() \rightarrow collect(a_element).allOperations() = self.allClasses() \rightarrow } \\
\text{collect(a_element).allOperations()@pre \rightarrow including(a_acceptOp) } \\
\text{self.allClasses() \rightarrow collect(a_concElement).allOperations() = self.allClasses() \rightarrow } \\
\text{collect(a_concElement).allOperations()@pre \rightarrow including(a_acceptOp) } 
\end{align*}
\]

4. Composing \( \text{pre}^\text{post4}' \) with Action 5 pre- and postconditions (\( \text{pre5}^\text{post5} \)).

This step involves composing the \( \text{pre}^\text{post4}' \) with the pre- and postconditions for Action 5 (\( \text{pre5}^\text{post5} \)).

Since action 5 does not have a precondition, there are no elements in post4' and pre5 which affect the occurrence of variables in the composed postcondition or to which an existence quantifier can bind. Thus, the pre- and postconditions for the composition \( \text{pre}^\text{post3}' \) with \( \text{pre4}^\text{post4} \) are shown in Table 6.10.

Table 6.10. Compose \( \text{pre}^\text{post4}' \) with Action 5 (\( \text{pre5}^\text{post5} \))

<table>
<thead>
<tr>
<th>pre: true</th>
</tr>
</thead>
<tbody>
<tr>
<td>post5':</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{self.allClasses() = self.allClasses()@pre \rightarrow including(a_visitor) } \\
\text{self.allClasses() \rightarrow one(a_visitor) } \\
\text{self.allAssociations = self.allAssociations@pre \rightarrow including(a_clientvisit) } \\
\text{self.a_client.Visit2Clnt = self.allClasses()\rightarrow select(a_visitor) } \\
\text{self.allClasses() \rightarrow select(a_visitor).Clnt2Visit = self.a_client } \\
\text{self.allClasses() = self.allClasses()@pre \rightarrow including(a_concVisitor) } \\
\text{self.allClasses() = self.allClasses()@pre \rightarrow including(a_concVisitor) } \\
\end{align*}
\]
Table 6.10 continued

```plaintext
self.allClasses() \rightarrow collect(a_concVisitor) \rightarrow size() = self.allClasses() \rightarrow 
    collect(a_concElement).allOperations()@pre \rightarrow select(a_elementOp) \rightarrow asSet() \rightarrow size() ^
self.a_client.a_visitor.generalization = self.allGeneralizations() \rightarrow collect(a_visitorgen) ^
self.allClasses() \rightarrow collect(a_concVisitor).generalization = self.allGeneralizations() \rightarrow 
    collect(a_visitorgen) ^
self.allGeneralizations() = self.allGeneralizations()@pre \rightarrow including(a_visitorgen) ^
self.allClasses() \rightarrow collect(a_concVisitor).allOperations() = self.allClasses() \rightarrow 
    collect(a_concVisitor).allOperations()@pre \rightarrow including(a_visitop)) ^
self.a_client.a_visitor \rightarrow select(a_visitop) \rightarrow size() = self.allClasses() \rightarrow select(a_concElement) \rightarrow 
    size() ^
self.allClasses() \rightarrow collect(a_concVisitor) \rightarrow select(a_visitop) \rightarrow asSet() \rightarrow size() = self.allClasses() \rightarrow 
    select(a_concElement) \rightarrow size() ^
self.allClasses() \rightarrow exist(oclIsTypeOf(Element)) ^
self.allClasses() \rightarrow exist(oclIsTypeOf(ConcreteElement)) ^
self.allClasses() \rightarrow collect(a_element).allOperations() = self.allClasses() \rightarrow 
    collect(a_element).allOperations()@pre \rightarrow including(a_acceptOp) ^
self.allClasses() \rightarrow collect(a_concElement).allOperations() = self.allClasses() \rightarrow 
    collect(a_concElement).allOperations()@pre \rightarrow including(a_acceptOp) ^
self.allClasses() \rightarrow collect(a_element).allOperations() = self.allClasses() \rightarrow 
    collect(a_element).allOperations()@pre \rightarrow excluding(a_elemOp) ^
self.allClasses() \rightarrow select(a_concElement).allOperations = self.allClasses() \rightarrow 
    select(a_concElement).allOperations@pre () \rightarrow excluding(a_elemOp) }
```

Composing `pre^post6` with the precondition of the transformation (i.e., `self.isValidSource(VisitorMetamodel)`) yields the precondition, postcondition pair (`preModel^postModel`) shown in Table 6.11. The composition of `true` with `self.isValidSource(VisitorMetamodel)` produces the final preconditions `preModel`:

```
self.isValidSource(VisitorMetamodel).
```

Table 6.11. Model-Level Precondition (preModel) and Postcondition (postModel)

<table>
<thead>
<tr>
<th>preModel:</th>
</tr>
</thead>
<tbody>
<tr>
<td>self.isValidSource(VisitorMetamodel)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>postModel:</th>
</tr>
</thead>
</table>
| `{} self.allClasses() = self.allClasses()@pre \rightarrow including(a_visitor) ^
  self.allClasses() \rightarrow one(a_visitor) ^
  self.allAssociations = self.allAssociations@pre \rightarrow including(a_clientvisit) ^
  self.a_client.Visit2Clnt = self.allClasses() \rightarrow select(a_visitor) ^
  self.allClasses() \rightarrow select(a_visitor).Clnt2Visit = self.a_client ^
  self.allClasses() = self.allClasses()@pre \rightarrow including(a_concVisitor) ^
  self.allClasses() \rightarrow collect(a_concVisitor) \rightarrow size() = self.allClasses() \rightarrow 
    collect(a_concElement).allOperations()@pre \rightarrow select(a_elementOp) \rightarrow asSet() \rightarrow size() ^
  self.allGeneralizations() = self.allGeneralizations()@pre \rightarrow including(a_visitorgen) ^
  self.a_client.a_visitor.generalization = self.allGeneralizations() \rightarrow collect(a_visitorgen) ^
  self.allClasses() \rightarrow collect(a_concVisitor).generalization = self.allGeneralizations() \rightarrow 
    collect(a_visitorgen) ^
  self.a_client.a_visitor.allChildren() = self.allClasses() \rightarrow select(a_concVisitor) ^
  self.allClasses() \rightarrow collect(a_concVisitor).allParents() = self.a_client.a_visitor ^
  self.a_client.a_visitor.allOperations() = self.a_client.a_visitor.allOperations()@pre \rightarrow 
    including(a_visitop)) ^`
### 6.4 Visitor Transformation Pre- and Postcondition Conformance

The Visitor pattern transformation can be validated formally by illustrating the mappings between the model-level pre- and postconditions and the metamodel-level pre- and postconditions. The model elements of the source models are instances of the metamodel elements of the VisitorMetamodel, therefore the model-level pre- and postcondition (preModel and postModel) pairs imply the metamodel-level pre- and postcondition (preMeta and postMeta) pairs.

The diagrams in Figure 6.4 thru Figure 6.6 show the mapping between the model-level postconditions and the transformation patterns. As stated previously, the transformation schema, combined with the restrictions specified in the transformation constraints depicts the postcondition of the Visitor transformation which indicate the model elements added, deleted and connected during the transformation. The OCL expressions in the “note” boxes represent constraint, defined at the model-level and the dashed lines illustrate the mapping of the model-level postcondition to the metamodel-level postcondition. The following is a postcondition specified in Figure 6.3.

```ocl
def self.allClasses() -> collect(a_concVisitor).allOperations() = self.allClasses() -> collect(a_concVisitor).allOperations()@pre -> including(a_visitop) ^
def self.a_client.a_visitor -> select(a_visitop) -> size() = self.allClasses() -> select(a_concElement) -> size() ^
def self.allClasses() -> collect(a_concVisitor) -> select(a_visitop) -> asSet() -> size() = self.allClasses() -> select(a_concElement) -> size() ^
def self.allClasses() -> exist(oclIsTypeOf(Element)) ^
def self.allClasses() -> exist(oclIsTypeOf(ConcreteElement)) ^
def self.allClasses() -> collect(a_element).allOperations() = self.allClasses() -> collect(a_element).allOperations()@pre -> including(a_acceptOp) ^
def self.allClasses() -> collect(a_concElement).allOperations() = self.allClasses() -> collect(a_concElement).allOperations()@pre -> including(a_acceptOp) ^
def self.allClasses() -> collect(a_element).allOperations() = self.allClasses() -> collect(a_element).allOperations()@pre -> excluding(a_elemOp) ^
def self.allClasses() -> select(a_concElement).allOperations = self.allClasses() -> select(a_concElement).allOperations@pre () -> excluding(a_elemOp) }
```

The postcondition states that the number of concrete visitor elements in the collection of classes must equal the number of element operations in the collection of operations owned by concrete element classes in the source model. This restriction is illustrated by the constraint template attached to the “Concrete Visitor Constraint” specified in the transformation constraint.
Figure 6.3. M1 to M2 Postcondition Mapping Add Concrete Visitor

Figure 6.4. M1 to M2 Postcondition Mapping Add Visitor Class
6.5 Structural Conformance

The diagram in Figure 6.7 structurally conforms to the Visitor source schema with respect to the bindings shown in Figure 6.8. The diagram includes an element class structure that composes the system. Figure 6.8 shows
that the Client class plays the role of |Client; the ComputerStructure class is bound to |ObjectStructure; Equipment plays the role of |Element; and the classes – Chassis, Card, FloppyDisk and Bus – plays the role of |ConcreteElement. The conforming parts of the model are given in the specialized metamodel diagram, that is, the metamodel representation of the source model, shown in Figure 6.7.

Figure 6.7. UML Diagram of Equipment Inventory System

Figure 6.8. Structurally Conforming Visitor Source Model.
The following steps represent an example of a transformation to introduce the Visitor pattern into the conforming source model. The model elements added are shown in bold text and darken boxes.

The first step in the transformation process is to create a Visitor class (instance of |Visitor) and link it to the client class (instance of |Client) using an association (instance of |ClientVisitorAssoc). The diagram, shown in Figure 6.10, shows that instances of |Visitor (EquipmentVisitor) and |ClientVisitorAssoc (clntvistassoc) are created. One end of the association (clntvistassoc) is connected to the Client class and the other end is connected to the EquipmentVisitor class.

Next, Concrete Visitor classes (instances of |ConcreteVisitor) are created for each instance of |ElemOp that is owned by the concrete element classes (instances of |ConcreteElement). The concrete visitor classes are connected to the visitor class via a visitor generalization (instance of |VisitorGeneralization) relationship. As illustrated in Figure 6.11 the concrete element owns two operations (Inventory and Pricing) that are instances of |ElemOp. The Inventory operation determines the creation of the InventoryVisitor and the Pricing operation determines the
creation of the PricingVisitor as specified by the transformation constraint given in Figure 6.2(c). An instance of \( \text{VisitorGeneralization} \), VisitGen, is created. The general (parent) end of the VisitGen is connected to the EquipmentVisitor, the parent, while the specific end is connect to the children (i.e., PricingVisitor and InventoryVisitor).

Figure 6.10. Add Visitor Class.

Figure 6.11. Create ConcreteVisitor Classes
The third step involves creating a visit operation (instance of |VisitOp) which corresponds to each concrete element class (instances of |ConcreteElement) for the visitor class (instance of |Visitor) and the concrete visitor classes (instances of |ConcreteVisitor). The diagram shown in Figure 6.12, illustrates that three sets of visit operations – visitChassis, visitCard, visitFloppyDisk, and visitBus – are created and linked to EquipmentVisitor, PricingVisitor and InventoryVisitor.

An operation that accepts the visitor (instance of |AcceptOp) is created and linked to instances of |Visitor and |ConcreteVisitor classes. In the example given in Figure 6.13, the Accept operation (instance of |AcceptOp) is created and linked to Equipment (instance of |Element), Chassis, Card, FloppyDisk, and Bus (instances of |ConcreteElement) classes.

The last step deletes the element operations (instances of |ElemOp) that are owned by the instances of the |Element and |ConcreteElement. All element operations (Inventory and Pricing) are deleted from Equipment, Chassis, Card, FloppyDisk and Bus as shown in Figure 6.14. Figure 6.14 also represents the target object model for the target model given in Figure 6.15. The Target model shows the model-level view of the transformed source model.
The diagram in Figure 6.16 shows that the target model structurally conforms to the specification given in the transformation schema and the restrictions specified by the transformation constraint with respect to the bindings.
Figure 6.15. Equipment Inventory Target Model

Figure 6.16 Conforming Target Model
6.6 Summary

The transformation pattern presented in the chapter formally introduces the Visitor pattern into existing design models that meets the criteria required in the source schema. Defining transformation patterns for Behavioral patterns illustrates how behavior expressed in a class diagrams is represented and restructured without affecting the intended outcome of the original design.
Chapter 7
Characterizing Structural Pattern Transformations

This chapter describes the transformation pattern defined for a Structural design pattern – the Bridge pattern. Section 7.1 provides an overview of the Bridge design pattern. Section 7.2 describes the Bridge transformation pattern. Section 7.3 discusses the Bridge transformation specification. Section 7.4 provides a formal method of validating the transformation pattern using composition of pre- and postconditions and an informal validation technique is presented in section 7.5. Section 7.6 summarizes the Bridge transformation pattern.

7.1  The Bridge Pattern

![Figure 7.1. Bridge Design Pattern Structure [Gamma et al., 1994]](image)

The Bridge pattern, shown in Figure 7.1, “decouples an abstraction from its implementation so that the two can vary independently” [Gamma et al., 1994]. In terms of the basic intent of the Bridge Pattern, abstraction refers to how different things relate to each other, and the implementor is the object that the abstract class and its derivation use to implant themselves with [Shalloway & Trott, 2004]. The abstraction and its implementation have separate hierarchical structures. The abstract class can be specialized to define subclasses of abstractions. The implementation class implements operations defined by abstractions.

7.2  Bridge Transformation Pattern

The Bridge transformation pattern is given in Figure 7.2. For structural design patterns, transformation patterns normally consist of only the source schema and the transformation schema. Since a structural pattern only modifies the structure of a model, all constraints can be expressed on model elements graphically within the transformation schema. The structure of model elements specified by the Bridge transformation pattern is described below.
7.2.1 Source Schema

The source schema, given in Figure 7.2(a), consists of classifier roles, |Client, |ProductImplementor, linked to each other by an association role, |ClntProdImplAssoc.

The source schema specifies that a conforming source model must have exactly one class that conforms to the |Client role and at least one class that plays the role of |ProductImplementor. This is indicated by the realization multiplicities specified on the |Client and |ProductImplementor roles. The |Client is a mandatory structure whose instances are classes that represent clients in the application domain. The |ClntProdImplAssoc association role specifies the relationship between |Client and |ProductImplementor classes. Each conforming |ClntProdImplAssoc association in a conforming source model must have one association-end, |Clnt, connected to the |Client class and the other association-end, |Prod, connected to the |ProductImplementor class. The realization multiplicity on the |Clnt and |Prod association-end roles specifies that a client class (instance of |Client) and product implementation class (instances of |ProductImplementor), respectively, must be a part of only one |ClntProdImplAssoc association.

The metamodel-level constraints for the source schema are defined as follows:

- An association-end that conforms to |Clnt must have a multiplicity of 1..1:

  context |Clnt
  inv self.lowerBound() = 1 and self.upperBound() = 1

- A conforming |Prod association-end must have a multiplicity of 1..1:

  context |Prod
  inv self.lowerBound() = 1 and self.upperBound() = 1
7.2.2 Transformation Schema

The transformation schema change involves:

- Adding new specializations of Class: one representing product abstraction class (|ProductAbstraction), one representing concrete abstractions classes (|ConcreteAbstraction), and one representing product implementation abstraction classes (|Implementor);
- Removing the specialized Association class (|ClntProdImplAssoc) representing associations between client classes and product implementations;
- Adding a specialization of the Association class (|ImplementorAssoc) that represents associations between product abstraction and product implementation abstraction classes;
- Adding a specialization of the Generalization (|AbstractionGeneralization) class representing the generalization relationship between product abstraction and concrete abstraction classes; and
- Adding a specialization of the Generalization (|ImplementorGeneralization) class representing the generalization relationship between product implementation abstraction and product implementation classes.

In the transformation schema, a |ProductAbstraction role is introduced into the structure and connected to an instance of |Client via a |ClntProdAbsAssoc association role. The realization multiplicity of |ProductAbstraction indicates there can be only one class that plays the |ProductAbstraction role in a conforming target model. Each conforming |ClntProdAbsAssoc in a conforming target model must have one association-end, |ClntAbs, connected to |Client and the other association-end, |Abs, connected to |ProductAbstraction. The realization multiplicity on the |ClntAbs association-end role specifies that a client class (instance of |Client) must be a part on only one client product abstraction association (instance of |ClntProdAbsAssoc). The realization multiplicity on the |Abs association-end role specifies that a production abstraction class (instance of |ProductAbstraction) must be a part on only one client product abstraction association (instance of |ClntProdAbsAssoc).

A |ConcreteAbstraction role is introduced into the structure and connected to the instance of |ProductAbstraction via an |AbstractionGeneralization generalization role. The realization multiplicity of |ConcreteAbstraction specifies that the conforming target model may contain zero or more classes that conform to the |ConcreteAbstraction role. In a conforming target model, the general end of |AbstractionGeneralization that is connected to an instance of |ProductAbstraction must conform to |AbsGen and the specific end connected to the
ConcreteAbstraction must conform to |ConcAbsGen. The 0..* realization multiplicity on the |AbsGen role specifies that a |ProductAbstraction class may be part of many |AbstractionGeneralization roles. The realization multiplicity (0..1) on the |ConcAbsGen role specifies that a |ConcreteAbstraction class may be part of at most one |AbstractionGeneralization roles.

The schema introduces an |Implementor classifier role into the structure and connects it to the instance of |ProductAbstraction via an |ImplementorAssoc association role. The realization multiplicity of the |Implementor role indicates there can be only one class that plays the role of |Implementor in a conforming target model. Instances of |Implementor are connected to instances of |ProductImplementor using the |ImplementorGeneralization generalization role. In a conforming target model, the general end of |ImplementorGeneralization connected to an instance of the |Implementor role must conform to the |Impl role, and the specific end of |ImplementorGeneralization connected to the |ProductImplementor must conform to the |ProdImpl role. The 1..* realization multiplicity on the |Impl role, specifies that a |Implementor class must be part of at least one |ImplementorGeneralization generalization role. The realization multiplicity (1..1) on the |ProdImpl role specifies that a |ProductImplementor class may be part only one |ImplementorGeneralization generalization role.

The schema connects |ProductAbstraction to |Implementor using an |ImplementorAssoc association role. Each conforming |ImplementorAssoc in a conforming target model must have the |ProdAbs association-end connected to the |ProductAbstraction role and the |Imp association-end connected to the |Implementor role. The realization multiplicity on |ProdAbs specifies that a production abstraction class (instance of |ProductAbstraction) must be a part on only one implementation association (instance of |ImplementorAssoc). The realization multiplicity on the |Imp association-end role specifies that an implementation abstraction class (instance of |Implementor) must be a part on only one implementation association (instance of |ImplementorAssoc).

The metamodel-level constraints for the transformation schema are defined as follows:

- An association-end that conforms to |ClntAbs must have a multiplicity of 1..1:
  ```
  context |ClntAbs
  inv self.lowerBound() = 1 and self.upperBound() = 1
  ```

- A conforming |Abs association-end must have a multiplicity of 1..1:
  ```
  context |Abs
  inv self.lowerBound() = 1 and self.upperBound() = 1
  ```

- A general end that conforms to |AbsGen must have a multiplicity of 0..*:
• A specific end that conforms to |ConcAbsGen must have a multiplicity of 1..1:
  
  **context** |ConcAbsGen  
  **inv** self.lowerBound() = 1 and self.upperBound() = 1

• An association-end that conforms to |ProdAbs must have a multiplicity of 1..1:

  **context** |ProdAbs  
  **inv** self.lowerBound() = 1 and self.upperBound() = 1

• A conforming |Imp association-end must have a multiplicity of 1..1:

  **context** |Imp  
  **inv** self.lowerBound() = 1 and self.upperBound() = 1

• A general end that conforms to |Implem must have a multiplicity of 1..*:

  **context** |Implem  
  **inv** self.lowerBound() = 1 and self.upperBound() = *

• A specific end that conforms to |ProdImpl must have a multiplicity of 1..1:

  **context** |ProdImpl  
  **inv** self.lowerBound() = 1 and self.upperBound() = 1

### 7.2.3 Metamodel-Level Pre- and Postconditions

At the metamodel (M2) level, a transformation specification for the Bridge pattern, referred to as BridgeMetaTransformationSpec, is specified by the pre- and postconditions defined by the Bridge transformation pattern. The Bridge source schema depicts the metamodel-level precondition, referred to as preMeta, which must be satisfied before a bridge transformation can execute. The postconditions are defined by the Bridge transformation schema and transformation constraint. These postconditions at the metamodel level are referred to as postMeta.

### 7.3 Bridge Pattern Transformation Specification

The Bridge pattern transformation specification (i.e., program) defines the operations on a UML class diagram required to introduce the Bridge design pattern into an existing model. The transformation specification that conforms to Bridge transformation pattern is given in Table 7.1. PBAL action constructs are used to specify to manipulate the model elements. The entity for which the transformation is defined is a source model package upon which the Bridge transformation is applied.

The precondition specifies that the source model must conform to the source schema definition for the Bridge transformation pattern given in Table 7.1. A Bridge pattern transformation involves the following steps:
1. Create Product Abstraction Class. A product abstraction class (instance of |ProductAbstraction) is created and connected to the client class (instance of |Client) using a client abstraction association (instance of |ClntProdAbsAssoc).

2. Create Concrete Abstraction Class. Concrete abstraction classes (instances of |ConcreteAbstraction) are created for each product implementation class (instance of |ProductImplementor).

3. Create Abstraction Generalization between Product Abstraction and Concrete Abstraction Classes. An abstraction generalization class (instance of |AbstractionGeneralization) is created to connect each concrete abstraction (instance of |ConcreteAbstraction) to the product abstraction class (instance of |ProductAbstraction). The general end of |AbstractionGeneralization is connected to the product abstraction class, and the specific end of the generalization is connected to the concrete abstraction classes.

4. Create Implementation Abstraction Class. A product implementation abstraction class (instance of |Implementor) is created and connected to the product abstraction (instance of |ProductAbstraction) using an instance of |ImplementorAssoc.

5. Generalized Product Implementor. A specialization of the Generalization (|ImplementorGeneralization) class is added to represent the generalization relationship between product implementation abstraction class (instance of |Implementor) and product implementation classes (instances of |ProductImplementor).

6. Remove Association between |Client and Product Implementation Classes. Remove the specialized Association class (|ClntProdImplAssoc) representing associations between client class (instance of |Client) and the product implementation class (instance of |ProductImplementor).

Table 7.1. Bridge Transformation Program

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: BridgeTransformation(BridgeMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre</td>
<td>isValidSource(BridgeMetamodel)</td>
</tr>
<tr>
<td>action</td>
<td>/* all variables are local therefore must be declared */</td>
</tr>
<tr>
<td></td>
<td>indx1 : Integer;</td>
</tr>
<tr>
<td></td>
<td>indx2 : Integer;</td>
</tr>
<tr>
<td></td>
<td>num_prodImpl : Integer;</td>
</tr>
<tr>
<td></td>
<td>num_concAbs : Integer;</td>
</tr>
</tbody>
</table>

/* action1 - Create a product abstraction class and Connect to Client via Client Product Abstraction Association */ |
| ProductAbstraction a_prodAbstraction; //ProductAbstraction variable declaration |
| a_prodAbstraction ::= _create_instance(ProductAbstraction); //create an instance of the ProductAbstraction class |

Table 7.1 continued

```c
ClntProdAbsAssoc a_clnProdAbsAssoc;  // ClntProdAbsAssoc variable declaration
a_clnProdAbsAssoc := _create_instance(ClntProdAbsAssoc);  // create instance of ClntProdAbsAssoc
Client a_client;  // Client variable declaration
a_client = _get_instances(Client);  // get all instances of Client
_connectClasses_Association(a_clnProdAbsAssoc, a_client, a_prodAbstraction);  // connect client class to product abstraction class via a Association

/* action2 - Create Implementation Abstraction Class and Connect to Product Abstraction class via Implementation Association */
Implementor a_implAbstraction;  // Implementor variable declaration
a_implAbstraction := _create_instance(Implementor);  // create an instance of the Implementor class
ImplementorAssoc a_implemAssoc;  // Implementor Association variable declaration
a_implemAssoc := _create_instance(ImplementorAssoc);  // create instance of ImplementorAssoc
_connectClasses_Association(a_implemAssoc, a_prodAbstraction, a_implAbstraction);  // connect implementation abstraction class to product abstraction class via a Association

/* action3 - Add concrete abstraction class */
ConcreteAbstraction[ ] a_concAbstraction;  // ConcreteAbstraction variable declaration
a_concAbstraction := _get_instances(ConcreteAbstraction);  // get all instances of ConcreteAbstraction
num_concAbs = a_concAbstraction → size();  // get the number of concrete abstraction classes
indx1 = num_concAbs + 1;
a_concAbstraction[indx1] := _create_instance(ConcreteAbstraction);  // create an instance of ConcreteAbstraction

/* action4 - Add Abstraction Generalization */
AbstractionGeneralization[ ] a_absGeneralization;  // AbstractionGeneralization declaration
a_absGeneralization = _create_instance(AbstractionGeneralization);  // create an instance of AbstractionGeneralization

/* action5 - Connect via Abstraction Generalization product abstraction class and the concrete abstraction class */
_connectClasses_Generalization(a_absGeneralization, a_prodAbstraction, a_concAbstraction[indx1]);  // connect concrete abstraction to product abstraction via abstraction generalization

/* action6 - Remove association between client and product implementation */
ProductImplementor[ ] a_prodImplementor;  // ProductImplementor class declaration
a_prodImplementor := _get_instances(ProductImplementor);  // get all instances of ProductImplementor
num_prodImpl = a_prodImplementor → size();  // get the number of product implementation classes
indx1 = 1;
while (indx1 <= num_prodImpl) do {
    _destroy_link(a_client, a_prodImplementor[indx1])
    indx1 = indx1 + 1;
}

/* action7 - Create Implementation Generalization */
ImplementatorGeneralization[ ] a_implGeneralization;  // ImplementatorGeneralization declaration
a_implGeneralization = _create_instance(ImplementatorGeneralization);  // create an instance of implementation generalization

/* action8 - Connect implementation abstraction class and the product implementation class using the Implementation Generalization */
```
Table 7.1 continued

```plaintext
indx1 = 1;
while(indx1 <= num_prodImpl) do {
    _connectClasses_Generalization(a_implGeneralization, a_implAbstraction,
    a_prodImplementor[indx1]); // connect implementation abstraction to product
    indx1 = indx1 + 1;
}

/*@ action9 - Create Product Implementation Class */
indx1 = num_prodImpl + 1
a_prodImplementor[indx1] = _create_instance(ProductImplementor); // create instance of
                  ProductImplementor
```

7.3.1 Model-Level Pre- and Postconditions

The transformation specification in Table 7.1 can be expressed as a sequence of actions that conveys the
operations required to perform a Bridge model-level transformation. Each action has preconditions and
postconditions that must be satisfied before the execution of an action and after the execution of an action,
respectively. Tables 7.2 - 7.10 provide the pre- and postconditions for each individual action.

Table 7.2 specifies the preconditions and postconditions for creating a product abstraction class. The
precondition for this action is true, since creating a product abstraction is the first action in the sequence and it is
assumed that the source model is valid. The postcondition ensures that the collection of classes includes only one
product abstraction class; the collection of associations includes a client product abstraction association; and the
client class is connected to the product abstraction class via the client product abstraction association, where the
|ClntAbs association-end is connected to the client class and the |Abs association-end is connected to product
abstraction class.

Table 7.2. Action 1 – Create a Product Abstraction

| context Package :: BridgeTransformation (BridgeMetamodel) |
| pre1: true |
| post1: self.allClasses() = self.allClasses@pre -> including(a_prodAbstraction) // product abstraction class
exist in collection of classes
and self.allClasses() -> one(a_prodAbstraction) // only one product abstraction class created
and self.allAssociations() = self.allAssociations()@pre -> including(a_clnProdAbsAssoc)
and self.a_client.Abs = self.allClasses() -> select(a_prodAbstraction)
and self.allClasses() -> select(a_prodAbstraction).ClntAbs = self.a_client |

Table 7.3 gives the pre- and postconditions for creating an implementation abstraction class and connecting it to
the product abstraction class. The precondition specifies that a product abstraction class must be an element in the
collection of classes and the implementation abstraction class is not an element in the collection of classes. The postcondition specifies that the collection of classes after execution consists of the implementation abstraction class; the collection of associations after execution contains an implementation association element; the ProdAbs association-end of the implementation association is connected to the product abstraction class; and the Imp association-end of the implementation association is connected to the implementation abstraction class.

Table 7.3. Action 2 – Create Implementation Abstraction class

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: BridgeTransformation (BridgeMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre2:</td>
<td>self.allClasses() (\rightarrow) exist(a_prodAbstraction) // product abstraction class exist in the collection of classes and self.allClasses() (\rightarrow) excludes(c</td>
</tr>
<tr>
<td>post2:</td>
<td>self.allClasses() = self.allClasses()@pre (\rightarrow) including (a_implAbstraction) // implementation abstraction class exist in collection of classes and self.allAssociations() = self.allAssociations()@pre (\rightarrow) including (a_implemAssoc) // implementation association is an element of the collection of associations and self.allClasses() (\rightarrow) select(a_implAbstraction).ProdAbs = self.allClasses() (\rightarrow) select(a_prodAbstraction) // ProdAbs association-end is connected to product abstraction class and self.allClasses() (\rightarrow) select(a_prodAbstraction).Imp = self.allClasses() (\rightarrow) select(a_implAbstraction) // Imp association-end is connected to the implementation abstraction class</td>
</tr>
</tbody>
</table>

The preconditions and postcondition shown in Table 7.4 are the constraints on the action that creates concrete abstraction classes. The preconditions specify that a product abstraction class must be an element in the collection of classes and that concrete abstraction classes are not elements in the collection of classes. The postcondition specifies that the collection of classes after execution consist of elements that conform to the concrete abstraction class.

Table 7.4. Action 3 – Create Concrete Abstraction Classes

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: BridgeTransformation (BridgeMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre3:</td>
<td>self.allClasses() (\rightarrow) exist(a_prodAbstraction) and self.allClasses() (\rightarrow) excludes(c</td>
</tr>
<tr>
<td>post3:</td>
<td>self.allClasses() (\rightarrow) includes(a_concAbstraction)</td>
</tr>
</tbody>
</table>

Table 7.5 gives the postcondition for the action responsible for creating an abstraction generalization model element. The postcondition specifies that the collection of generalization after execution consist of elements that conform to the abstraction generalization.
Table 7.5. Action 4 – Create Abstraction Generalization

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: BridgeTransformation (BridgeMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre4:</td>
<td>-- none</td>
</tr>
<tr>
<td>post4:</td>
<td>self.allGeneralizations() (\rightarrow) includes(a_absGeneralization)</td>
</tr>
</tbody>
</table>

Table 7.6 specifies the pre- and postconditions for connecting the product abstraction class to concrete abstraction classes using an abstraction generalization. The precondition specifies that the collection of classes must contain a product abstraction and concrete abstraction class, and the collection of generalization relationships must consist of at least one abstraction generalization. The postcondition specifies that the general end of the abstraction generalization is connected to the product abstraction class; the specific end of the abstract generalization is connected to the concrete abstraction class; the children of the product abstraction class are concrete abstraction classes; and the product abstraction class is the parent of all concrete abstraction classes.

Table 7.6. Action 5 – Connect Product and Concrete Abstraction

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: BridgeTransformation (BridgeMetamodel)</th>
</tr>
</thead>
</table>
| pre5:   | self.allClasses() \(\rightarrow\) exist(a_prodAbstraction)  
and self.allClasses() \(\rightarrow\) exist(a_concAbstraction)  
and self.allGeneralizations() \(\rightarrow\) exist(a_absGeneralization) |
| post5:  | self.allGeneralizations() \(\rightarrow\) collect(a_absGeneralization).general = self.allClasses() \(\rightarrow\)  
select(a_prodAbstraction)  
and self.allGeneralizations() \(\rightarrow\) collect(a_absGeneralization).specific = self.allClasses() \(\rightarrow\)  
select(a_concAbstraction)  
and self.allClasses() \(\rightarrow\) select(a_prodAbstraction).allChildren() = self.allClasses() \(\rightarrow\)  
select(a_concAbstraction)  
and self.allClasses() \(\rightarrow\) collect(a_concAbstraction).allParents() = self.a_client.a_prodAbstraction |

The postcondition for removing all instances of the client product implementation association connecting the client class to product implementation classes is given in Table 7.7. The postcondition specifies that the collection of associations after execution is equivalent to the collection of associations before execution of the action excluding association elements that conform to an implementation association.

Table 7.7. Action 6 – Remove Association

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: BridgeTransformation (BridgeMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre6:</td>
<td>--none</td>
</tr>
<tr>
<td>post6:</td>
<td>self.allAssociations() = self.allAssociations()@pre (\rightarrow) excluding(a_clnProdImplAssoc) // client product implementation association is not an element in the collection of all associations</td>
</tr>
</tbody>
</table>
The postcondition given in Table 7.8 is the constraint on the action that creates an implementation generalization. The postcondition specifies that the collection of generalizations after execution consist of elements that conform to implementation generalization.

Table 7.8. Action 7 – Create Implementation Generalization

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: BridgeTransformation (BridgeMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pre7:</strong></td>
<td>-- none</td>
</tr>
<tr>
<td><strong>post7:</strong></td>
<td>self.allGeneralizations() = self.allGeneralizations()@pre → including(a_implGeneralization) // implementation generalization exist in the collection of Generalizations</td>
</tr>
</tbody>
</table>

The pre- and postconditions given in Table 7.9 are the constraints on the action that uses a generalization relationship to connect the implementation abstraction class and the product implementation classes. The precondition specifies that the collection of classes must contain implementation abstraction and product implementation classes. The postcondition specifies that the general end of the implementation generalization is connected to the implementation abstraction class; the specific end of the implementation generalization is connected to the product implementation classes; the children of the implementation abstraction class are product implementation classes; and the implementation abstraction class is the parent of all product implementation classes.

Table 7.9. Action 8 – Connect Classes via Implementation Generalization

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: BridgeTransformation (BridgeMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pre8:</strong></td>
<td>self.allClasses() = exist(a_implAbstraction) // implementation abstraction class exist in collection of classes and self.allClasses() = exist(a_prodImplementor) // product implementation class exist in collection of classes</td>
</tr>
<tr>
<td><strong>post8:</strong></td>
<td>self.allGeneralizations() → select(a_implGeneralization).general = self.allClasses() → select(a_implAbstraction) // general end of implementation generalization is connected to implementation abstraction and self.allGeneralizations() → select(a_implGeneralization).specific = self.allClasses() → select(a_prodImplementor) // specific end of implementation generalization are the product implementation classes and self.allClasses() → select(a_implAbstraction).allChildren() = self.allClasses() → select(a_prodImplementor) // children of implementation abstraction are product implementation classes and self.allClasses() → select(a_prodImplementor).allParents() = self.allClasses() → select(a_implAbstraction) // parent of product implementation classes is implementation abstraction</td>
</tr>
</tbody>
</table>

The postcondition given in Table 7.10 specifies the constraint on the action that adds a product implementation class to the structure. The postcondition specifies that a new instance of production implementation is added to the collection of classes after execution.
Table 7.10. Action 9 – Add Product Implementation Class

<table>
<thead>
<tr>
<th>context</th>
<th>Package :: BridgeTransformation (BridgeMetamodel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre9:</td>
<td>-- none</td>
</tr>
<tr>
<td>post9:</td>
<td>self.allClasses() = self.allClasses()@pre → including (a_prodImplementor) // new instance of product implementation class added to collection of classes</td>
</tr>
</tbody>
</table>

7.3.2 Composition of Bridge Model-Level Pre- and Postconditions

A single precondition, postcondition pair, referred to as preModel and postModel, is obtained from composing the pre- and postconditions pairs given in Tables 7.2 - 7.10, two at a time. The following steps specify how the Bridge transformation pre- and postconditions are composed. The character “\&” represents a “logical and”.

1. Compose Action 1 pre- and postconditions (pre1\&post1) with Action 2 pre- and postconditions (pre2\&post2).
   - Rename variables that affect the occurrence in post1 and pre2
     - a_prodAbstraction ⇒ z

     The OCL statements for post1 and pre2 are renamed as follows:

     **post1:** {
       self.allClasses() = self.allClasses()@pre → including (z) ^ self.allClasses() → one(z) ^
       self.allAssociations() = self.allAssociations()@pre → including(a_clnProdAbsAssoc) ^
       self.a_client.Abs = self.allClasses() → select(z) ^ self.allClasses() → select(z).ClntAbs =
       self.a_client |

     **pre2:** {
       self.allClasses() → exist(z) ^ self.allClasses() → excludes(c | c.oclIsTypeOf(Implementor)) } |

   - Bind z to an existence quantifier

     \[ \exists z \{ \text{self.allClasses()} = \text{self.allClasses()}@pre → including (z) ^ \text{self.allClasses()} → one(z)^ \]

     self.allAssociations() = self.allAssociations()@pre → including(a_clnProdAbsAssoc)^

     self.a_client.Abs = self.allClasses() → select(z) ^ self.allClasses() → exist(z) ^ self.allClasses() → select(z).ClntAbs =

     self.a_client^ self.allClasses() → excludes(c | c.oclIsTypeOf(Implementor))^ self.allClasses() = self.allClasses()@pre → including

     (a_implAbstraction)^ self.allAssociations() = self.allAssociations()@pre → including

     (a_implemAssoc)^ self.allClasses() → select(a_implAbstraction).ProdAbs = self.allClasses() → select(a_prodAbstraction) ^ self.allClasses() → select(a_prodAbstraction).Imp = self.allClasses() → select(a_implAbstraction) } |
• Simplify the expression

- The OCL expressions \( \text{self.allClasses()} = \text{self.allClasses()@pre} \rightarrow \text{including (z)} \) and \( \text{self.allClasses()} \rightarrow \text{exist(z)} \) are equivalent statements since both verify that the \( z \) is an element in the collection of classes. The OCL expression \( \text{self.allClasses()} = \text{self.allClasses()@pre} \rightarrow \text{including (z)} \) will be deleted from the composed pre- and postconditions.

- The element \( \text{a_implAbstraction} \) is an instance of \(|\text{Implementor}|\). The OCL expression \( \text{self.allClasses()} = \text{self.allClasses()@pre} \rightarrow \text{including (a_implAbstraction)} \) in the postcondition of action 2 evaluates the collection of classes after execution, and the OCL expression \( \text{self.allClasses()} \rightarrow \text{excludes(c | c.oclIsTypeOf(Implementor))} \) in the precondition of Action 2 evaluates the collection prior to the action being executed. Since the focus of the transformation is the state of the model after execution of the action, the OCL expression \( \text{self.allClasses()} \rightarrow \text{excludes (c | c.oclIsTypeOf(Implementor))} \) will be deleted from the composed pre- and postconditions.

The simplified expression is as follows:

\[
\exists z \bullet \{ \text{self.allClasses()} \rightarrow \text{exist(z)} \land \text{self.allClasses()} \rightarrow \text{one(z)} \land \text{self.allAssociations()} = \text{self.allAssociations()@pre} \rightarrow \text{including(a_clnProdAbsAssoc) \land self.a_client.Abs = self.allClasses()} \rightarrow \text{select(z)} \land \text{self.allClasses()} \rightarrow \text{select(z).ClntAbs = self.a_client^} \\
\text{self.allClasses()} = \text{self.allClasses()@pre} \rightarrow \text{including (a_implAbstraction) \land self.allAssociations()} = \text{self.allAssociations()@pre} \rightarrow \text{including (a_implemAssoc) \land self.allClasses()} \rightarrow \text{select(a_implAbstraction).ProdAbs = self.allClasses()} \rightarrow \text{select(a_implAbstraction)} \land \text{select(a_implAbstraction).Imp = self.allClasses()} \rightarrow \text{select(a_implAbstraction)} \}
\]

The pre- and postconditions, with \( z \) changed back to \( \text{a_prodAbstraction} \), for the composition of the pre- and postconditions for Action 1 with the pre- and postconditions for Action 2 are shown in Table 7.11.

2. Compose pre^post2' with Action 3 pre- and postconditions (pre3^post3).

This step involves composing the pre- and post conditions (pre^post2') shown in Table 7.11 with the pre- and preconditions for Action 3 (pre3^post3).
Table 7.11. Action 1 (pre1^post1) Composed With Action 2 (pre2^post2)

**pre:**
- true

**post2':**
\[
\{ \text{self.allClasses()} = \text{self.allClasses}@pre \rightarrow \text{including(a_prodAbstraction)} \} \\
\text{self.allClasses()} \rightarrow \text{one(a_prodAbstraction)} \\
\text{self.allAssociations()} = \text{self.allAssociations()}@pre \rightarrow \text{including(a_clnProdAbsAssoc)} \\
\text{self.a_client.Abs} = \text{self.allClasses()} \rightarrow \text{select(a_prodAbstraction)} \\
\text{self.allClasses()} \rightarrow \text{select(a_prodAbstraction)}.\text{ClntAbs} = \text{self.a_client} \\
\text{self.allAssociations()} = \text{self.allAssociations()}@pre \rightarrow \text{including(a_implAbstraction)} \\
\text{self.allAssociations()} = \text{self.allAssociations()}@pre \rightarrow \text{including(a_implemAssoc)} \\
\text{self.allClasses()} \rightarrow \text{select(a_implAbstraction)}.\text{ProdAbs} = \text{self.allClasses()} \rightarrow \\
\text{select(a_implAbstraction)} \\
\text{self.allClasses()} \rightarrow \text{select(a_prodAbstraction)}.\text{Imp} = \text{self.allClasses()} \rightarrow \text{select(a_implAbstraction)} \}
\]

- Rename variables that affect the occurrence in post1 and pre2
  - \text{a_prodAbstraction} \Rightarrow \text{z}

The OCL statements for post1 and pre2 are renamed as follows:

**post2':**
\[
\{ \text{self.allClasses()} \rightarrow \text{exist(z)} \} \cup \text{self.allClasses()} \rightarrow \text{one(z)} \cup \text{self.allAssociations()} = \\
\text{self.allAssociations()}@pre \rightarrow \text{including(a_clnProdAbsAssoc)} \cup \text{self.a_client.Abs} = \\
\text{self.allClasses()} \rightarrow \text{select(z)} \cup \text{self.allClasses()} \rightarrow \text{select(z)}.\text{ClntAbs} = \text{self.a_client} \\
\text{self.allClasses()} = \text{self.allClasses()}@pre \rightarrow \text{including(a_implAbstraction)} \cup \text{self.allClasses()} \rightarrow \\
\text{one(a_implAbstraction)} \cup \text{self.allAssociations()} = \text{self.allAssociations()}@pre \rightarrow \text{including(a_implemAssoc)} \\
\text{self.allClasses()} \rightarrow \text{select(a_implAbstraction)}.\text{ProdAbs} = \text{self.allClasses()} \rightarrow \\
\text{select(a_implAbstraction)} \\
\text{self.allClasses()} \rightarrow \text{select(a_prodAbstraction)}.\text{Imp} = \text{self.allClasses()} \rightarrow \text{select(a_implAbstraction)} \}
\]

**pre3:**
\[
\{ \text{self.allClasses()} \rightarrow \text{exist(z)} \} \cup \text{self.allClasses()} \rightarrow \text{excludes(c |} \\
\text{c.oclIsTypeOf(a_concAbstraction))} \}
\]

- Bind \text{z} to an existence quantifier
\[
\exists \text{z} \bullet \{ \text{self.allClasses()} \rightarrow \text{exist(z)} \} \cup \text{self.allClasses()} \rightarrow \text{one(z)} \cup \text{self.allAssociations()} = \\
\text{self.allAssociations()}@pre \rightarrow \text{including(a_clnProdAbsAssoc)} \cup \text{self.a_client.Abs} = \\
\text{self.allClasses()} \rightarrow \text{select(z)} \cup \text{self.allClasses()} \rightarrow \text{select(z)}.\text{ClntAbs} = \text{self.a_client} \\
\text{self.allClasses()} = \text{self.allClasses()}@pre \rightarrow \text{including(a_implAbstraction)} \cup \text{self.allClasses()} \rightarrow \\
\text{one(a_implAbstraction)} \cup \text{self.allAssociations()} = \text{self.allAssociations()}@pre \rightarrow \text{including(a_implemAssoc)} \\
\text{self.allClasses()} \rightarrow \text{select(a_implAbstraction)}.\text{ProdAbs} = \text{self.allClasses()} \rightarrow \\
\text{select(z)} \cup \text{self.allClasses()} \rightarrow \text{select(z)}.\text{Imp} = \text{self.allClasses()} \rightarrow \text{select(a_implAbstraction)} \cup \\
\]
Simplify the expression post2'^post3'^post3.

- The OCL expressions self.allClasses() → exist(z) ^ self.allClasses() → excludes(c | c.oclIsTypeOf(ConcreteAbstraction)) ^ self.allClasses() = self.allClasses()@pre → including (a_concAbstraction) will be deleted for the composed pre- and postconditions.

- The element a_concAbstraction is an instance of |ConcreteAbstraction, therefore any reference to a_concAbstraction is also a reference to |ConcreteAbstraction. The expression self.allClasses() = self.allClasses()@pre → including (a_concAbstraction) in the postcondition evaluates the collection of classes after execution, and the OCL expression self.allClasses() → excludes(c | c.oclIsTypeOf(ConcreteAbstraction)) in the precondition evaluates the collection prior to the action being executed. Since the focus of the transformation is the state of the model after execution of the action, the OCL expression self.allClasses() → excludes (c | c.oclIsTypeOf(ConcreteAbstraction)) will be deleted from the composed pre- and postconditions.

∃ z • { self.allClasses() → exist(z) ^ self.allClasses() → one(z) ^ self.allAssociations() = self.allAssociations()@pre → including(a_clnProdAbsAssoc) ^ self.a_client.Abs = self.allClasses() → select(z) ^ self.allClasses() → select(z).ClntAbs = self.a_client^ self.allClasses() = self.allClasses()@pre → including (a_implAbstraction) ^ self.allClasses() → one(a_implAbstraction) ^ self.allAssociations() = self.allAssociations()@pre → including (a_implmAssoc) ^ self.allClasses() → select(a_implAbstraction).ProdAbs = self.allClasses() → select(z) ^ self.allClasses() → select(z).Imp = self.allClasses() → select(a_implAbstraction) ^ self.allClasses() = self.allClasses()@pre → including (a_concAbstraction) }

The pre- and postconditions, with z changed back to a_prodAbstraction, for the composition of the pre- and postconditions (pre^post2') with the pre- and postconditions for Action 3 are shown in Table 7.12.

3. Compose pre^post3’ with Action 4 pre- and postconditions (pre4^post4).

This step involves composing the pre- and post conditions (pre^post3’) shown in Table 7.12 with the pre- and preconditions for Action 4 (pre4^post4). There are no variables in post3’ or pre4 that affect the
occurrence of variables in the composed precondition and postcondition or to which an existence quantifier can bind. The pre- and postcondition for the composition of the pre- and postconditions (pre^post3’) with the pre- and postconditions for Action 4 are shown in Table 7.13.

4. Compose pre’post4’ with Action 5 pre- and postconditions (pre5’post5).

This step involves composing the pre- and post conditions (pre’post4’) shown in Table 7.13 with the pre- and postconditions for Action 5 (pre5’post5).

- Rename variables that affect the occurrence in post4’ and pre5
  - a_prodAbstraction ⇒ z
  - a_concAbstraction ⇒ y
The OCL statements for post4' and pre5 are renamed as follows:

**post4':** \{ self.allClasses() \rightarrow \text{exist}(z) \land self.allClasses() \rightarrow \text{one}(z) \land self.allAssociations() = self.associations()@pre \rightarrow \text{including}(a_{clnProdAbsAssoc}) \land self.a_client.Abs = self.allClasses() \rightarrow \text{select}(z) \land self.allClasses() \rightarrow \text{select}(z).ClntAbs = self.a_client^ \\
self.allClasses() = self.allClasses()@pre \rightarrow \text{including}(a_{implAbstraction}) \land self.allClasses() \rightarrow \\
one(a_{implAbstraction}) \land self.allAssociations() = self.associations()@pre \rightarrow \text{including} \\
(a_{implemAssoc}) \land self.allClasses() \rightarrow \text{select}(a_{implemAssoc}).ProdAbs = self.allClasses() \rightarrow \\
select(z) \land self.allClasses() \rightarrow \text{select}(z).Imp = self.allClasses() \rightarrow \text{select}(a_{implemAssoc}) \land \\
self.allClasses() \rightarrow \text{includes}(y) \land self.allGeneralizations() = \text{includes}(x) \} \\

**pre5:** \{self.allClasses() \rightarrow \text{exist}(z) \land self.allClasses() \rightarrow \text{exist}(y) \land self.allGeneralizations() \rightarrow \text{exist}(x)\}

- Bind \(x, y,\) and \(z\) to an existence quantifier for post4'\rightarrow pre5\rightarrow post5.

\(\exists x,y,z \bullet \{ self.allClasses() \rightarrow \text{exist}(z) \land self.allClasses() \rightarrow \text{one}(z) \land self.allAssociations() = self.associations()@pre \rightarrow \text{including}(a_{clnProdAbsAssoc}) \land self.a_client.Abs = self.allClasses() \rightarrow \text{select}(z) \land self.allClasses() \rightarrow \text{select}(z).ClntAbs = self.a_client^ \\
self.allClasses() = self.allClasses()@pre \rightarrow \text{including}(a_{implAbstraction}) \land self.allClasses() \rightarrow \\
one(a_{implAbstraction}) \land self.allAssociations() = self.associations()@pre \rightarrow \text{including} \\
(a_{implemAssoc}) \land self.allClasses() \rightarrow \text{select}(a_{implemAssoc}).ProdAbs = self.allClasses() \rightarrow \\
select(z) \land self.allClasses() \rightarrow \text{select}(z).Imp = self.allClasses() \rightarrow \text{select}(a_{implemAssoc}) \land \\
self.allClasses() \rightarrow \text{includes}(y) \land self.allGeneralizations() = \text{includes}(x) \land self.allClasses() \rightarrow \\
\text{exist}(z) \land self.allClasses() \rightarrow \text{exist}(y) \land self.allGeneralizations() \rightarrow \text{exist}(x) \land \\
self.allGeneralizations() \rightarrow \text{collect}(a_{absGeneralization}).general = self.allClasses() \rightarrow \\
\text{select}(a_{prodAbstraction}) \land self.allGeneralizations() \rightarrow \text{collect}(a_{absGeneralization}).specific = self.allClasses() \rightarrow \\
\text{select}(a_{concAbstraction}) \land self.allClasses() \rightarrow \\
\text{select}(a_{prodAbstraction}).allChildren() = self.allClasses() \rightarrow \text{select}(a_{concAbstraction}) \land \\
self.allClasses() \rightarrow \text{collect}(a_{concAbstraction}).allParents() = self.a_client.a_prodAbstraction} \}

- Simplify the expression post4'\rightarrow pre5\rightarrow post5.
- The OCL expressions `self.allClasses() -> exist(z) ^ self.allClasses() -> exist(z) = self.allClasses() -> exist(z)`. Therefore one of the OCL expressions can be deleted from the composed postconditions.

- The OCL expressions `self.allClasses() = self.allClasses()@pre -> including (y)` and `self.allClasses() -> exist(y)` evaluate the collection of classes to determine if the element y is an element in the collection. Therefore, since the expressions are evaluating the same condition, only one shall be include in the composed postconditions.

- The OCL expressions `self.allGeneralizations() -> includes(x)` and `self.allGeneralizations() -> exist(x)` evaluate the collection of generalizations to determine if the element x is an element in the collection. Therefore, since the expressions are evaluating the same condition, only one shall be include in the composed postconditions.

Composition of the pre- and postconditions, with x changed back to a_absGeneralization, y changed back to a_concAbstraction, and z changed back to a_prodAbstraction, for the composition of the pre- and postconditions (pre^post4') with the pre- and postconditions for Action 5 are shown in Table 7.14.

<table>
<thead>
<tr>
<th>pre: true</th>
<th>post5':</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>self.allClasses() = self.allClasses@pre -&gt; including(a_prodAbstraction) ^</td>
</tr>
<tr>
<td></td>
<td>self.allClasses() -&gt; one(a_prodAbstraction) ^</td>
</tr>
<tr>
<td></td>
<td>self.allAssociations() = self.allAssociations()@pre -&gt; including(a_clnProdAbsAssoc) ^</td>
</tr>
<tr>
<td></td>
<td>self.a_client.Abs = self.allClasses() -&gt; select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td></td>
<td>self.allClasses() -&gt; select(a_prodAbstraction).ClntAbs = self.a_client^</td>
</tr>
<tr>
<td></td>
<td>self.allClasses() = self.allClasses()@pre -&gt; including (a_implAbstraction) ^</td>
</tr>
<tr>
<td></td>
<td>self.allClasses() -&gt; one(a_implAbstraction) ^</td>
</tr>
<tr>
<td></td>
<td>self.allAssociations() = self.allAssociations()@pre -&gt; including (a_implemAssoc) ^</td>
</tr>
<tr>
<td></td>
<td>self.allClasses() -&gt; select(a_implAbstraction).ProdAbs = self.allClasses() -&gt; select(a_prodAbstraction)^</td>
</tr>
<tr>
<td></td>
<td>self.allClasses() -&gt; select(a_prodAbstraction).Imp = self.allClasses() -&gt; select(a_implAbstraction)^</td>
</tr>
<tr>
<td></td>
<td>self.allClasses() -&gt; exist(a_concAbstraction) ^</td>
</tr>
<tr>
<td></td>
<td>self.allGeneralizations() -&gt; exist(a_absGeneralization) ^</td>
</tr>
<tr>
<td></td>
<td>self.allGeneralizations() -&gt; collect(a_absGeneralization).general = self.allClasses() -&gt; select(a_prodAbstraction)^</td>
</tr>
<tr>
<td></td>
<td>self.allGeneralizations() -&gt; collect(a_absGeneralization).specific = self.allClasses() -&gt; select(a_concAbstraction)^</td>
</tr>
<tr>
<td></td>
<td>self.allClasses() -&gt; select(a_prodAbstraction).allChildren() = self.allClasses() -&gt; select(a_concAbstraction)^</td>
</tr>
<tr>
<td></td>
<td>self.allClasses() -&gt; collect(a_concAbstraction).allParents() = self.a_client.a_prodAbstraction</td>
</tr>
</tbody>
</table>

5. Compose pre^post5' with Action 6 pre- and postconditions (pre6^post6).

This step involves composing the pre- and post conditions (pre^post5') shown in Table 7.14 with the pre- and postconditions for Action 6 (pre6^post6). There are no variables in post5’ or pre6 that affect the
occurrence of variables in the composed precondition and postcondition or to which an existence quantifier can bind. Therefore the composition of pre- and postcondition (pre^post5’) with the pre- and postconditions (pre6^post6) of Action 6 are shown in Table 7.15.

Table 7.15. Composed pre’post5’ with Action 6 (pre6’post6)

<table>
<thead>
<tr>
<th>pre: true</th>
</tr>
</thead>
<tbody>
<tr>
<td>post6’:</td>
</tr>
<tr>
<td>{ self.allClasses() = self.allClasses@pre \rightarrow including(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow one(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations()@pre \rightarrow including(a_clnProdAbsAssoc) ^</td>
</tr>
<tr>
<td>self.a_client.Abs = self.allClasses() \rightarrow select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow select(a_prodAbstraction).ClntAbs = self.a_client ^</td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses()@pre \rightarrow including (a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow one(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations()@pre \rightarrow including (a_implmAssoc) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow select(a_implAbstraction).ProdAbs = self.allClasses() \rightarrow</td>
</tr>
<tr>
<td>select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow select(a_prodAbstraction).Imp = self.allClasses() \rightarrow select(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow exist(a_concAbstraction) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() \rightarrow exist(a_absGeneralization) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() \rightarrow collect(a_absGeneralization).general = self.allClasses() \rightarrow</td>
</tr>
<tr>
<td>select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() \rightarrow collect(a_absGeneralization).specific = self.allClasses() \rightarrow</td>
</tr>
<tr>
<td>select(a_concAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow select(a_prodAbstraction).allChildren() = self.allClasses() \rightarrow</td>
</tr>
<tr>
<td>select(a_concAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow collect(a_concAbstraction).allParents() = self.a_client.a_prodAbstraction ^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations()@pre \rightarrow excluding(a_clnProdImplAssoc) }</td>
</tr>
</tbody>
</table>

6. Compose pre’post6 with Action 7 pre- and postconditions (pre7’post7).

This step involves composing the pre- and post conditions (pre’post6’) shown in Table 7.15 with the pre- and preconditions for action 7 (pre7’post7). There are no variables in post6’ or pre7 that affect the occurrence of variables in the composed precondition and postcondition or to which an existence quantifier can bind. The composition of pre- and postconditions (pre’post6’) with the pre- and postconditions (pre7’post7) of action 7 is shown in Table 7.16.

Table 7.16. Composed pre’post6’ with Action 7 (pre7’post7)

<table>
<thead>
<tr>
<th>pre: true</th>
</tr>
</thead>
<tbody>
<tr>
<td>post7’:</td>
</tr>
<tr>
<td>{ self.allClasses() = self.allClasses@pre \rightarrow including(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow one(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations()@pre \rightarrow including(a_clnProdAbsAssoc) ^</td>
</tr>
<tr>
<td>self.a_client.Abs = self.allClasses() \rightarrow select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow select(a_prodAbstraction).ClntAbs = self.a_client ^</td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses()@pre \rightarrow including (a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() \rightarrow one(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations()@pre \rightarrow including (a_implmAssoc) ^</td>
</tr>
</tbody>
</table>
7. Compose pre"post7" with Action 5 pre- and postconditions (pre8"post8).

This step involves composing the pre- and post conditions (pre"post7") shown in Table 7.16 with the pre-
and postconditions for Action 8 (pre8"post8).

- Rename variables that affect the occurrence in post7' and pre8

  - a_implAbstraction ⇒ z

The OCL statements for post4' and pre5 are renamed as follows:

**post7':** { self.allClasses() = self.allClasses@pre → including(a_prodAbstraction) ^ self.allClasses() →
  one(a_prodAbstraction) ^ self.allAssociations() = self.allAssociations@pre →
  including(a_clnProdAbsAssoc) ^ self.a_client.Abs = self.allClasses() → select(a_prodAbstraction)
  ^ self.allClasses() → select(a_prodAbstraction).ClntAbs = self.a_client^ self.allClasses() =
  self.allClasses@pre → including (z) ^ self.allAssociations() = self.allAssociations@pre →
  including (a_implemAssoc) ^ self.allClasses() → select(z).ProdAbs = self.allClasses() →
  select(a_prodAbstraction) ^ self.allClasses() → select(a_prodAbstraction).Imp = self.allClasses()
  → select(z) ^ self.allClasses() → exist(a_concAbstraction) ^ self.allGeneralizations() →
  exist(a_absGeneralization) ^ self.allGeneralizations() → collect(a_absGeneralization).general =
  self.allClasses() → select(a_prodAbstraction) ^ self.allGeneralizations() →
  collect(a_absGeneralization).specific = self.allClasses() → select(a_concAbstraction) ^
  self.allClasses() → select(a_prodAbstraction).allChildren() = self.allClasses() →
  select(a_concAbstraction) ^ self.allClasses() =
  self.allClasses@pre → excluding(a_clnProdImplAssoc) ^
  self.allGeneralizations() → includes(a_implGeneralization) }
self.a_client.a_prodAbstraction^ self.allAssociations() = self.allAssociations(@pre →
excluding(a_clnProdImplAssoc) ^ self.allGeneralizations() → includes(a_implGeneralization) }

pre8: { self.allClasses() = exist(z) ^ self.allClasses() = exist(a_prodImplementor) }

- Bind z to an existence quantifier for post4^pre5^post5.
  \[\exists z \bullet \{ \{ \text{self.allClasses()} = \text{self.allClasses}@pre \rightarrow \text{including(a_prodAbstraction)} ^ \text{self.allClasses()} \rightarrow \text{one(a_prodAbstraction)} ^ \text{self.allAssociations()} = \text{self.allAssociations}@pre \rightarrow \text{including(a_clnProdAbsAssoc)} \} ^ \text{self.a_client.Abs} = \text{self.allClasses()} \rightarrow \text{select(a_prodAbstraction)} ^ \text{self.allClasses()} \rightarrow \text{select(a_prodAbstraction).ClntAbs} = \text{self.a_client^ self.allClasses()} = \text{self.allClasses}@pre \rightarrow \text{including(z)} \} ^ \text{self.allAssociations()} = \text{self.allAssociations}@pre \rightarrow \text{including(a_implemAssoc)} ^ \text{self.allClasses()} → \text{select(z).ProdAbs} = \text{self.allClasses()} \rightarrow \text{select(a_prodAbstraction)} ^ \text{self.allClasses()} → \text{select(a_prodAbstraction).Imp} = \text{self.allClasses()} \rightarrow \text{select(z)} \} ^ \text{self.allClasses()} → \text{exist(a_concAbstraction)} ^ \text{self.allGeneralizations()} → \text{exist(a_absGeneralization)} ^ \text{self.allGeneralizations()} → \text{collect(a_absGeneralization).general} = \text{self.allClasses()} \rightarrow \text{select(a_prodAbstraction)} ^ \text{self.allGeneralizations()} → \text{collect(a_absGeneralization).specific} = \text{self.allClasses()} \rightarrow \text{select(a_concAbstraction)} ^ \text{self.allClasses()} \rightarrow \text{select(a_concAbstraction).allChildren()} = \text{self.allClasses()} \rightarrow \text{select(a_concAbstraction)} ^ \text{self.allClasses()} \rightarrow \text{collect(a_concAbstraction).allParents()} = \text{self.a_client.a_prodAbstraction} ^ \text{self.allAssociations()} = \text{self.allAssociations}@pre \rightarrow \text{excluding(a_clnProdImplAssoc)} ^ \text{self.allGeneralizations()} → \text{includes(a_implGeneralization)} ^ \text{self.allClasses()} = \text{exist(z)} ^ \text{self.allClasses()} = \text{exist(a_prodImplementor)} ^ \text{self.allGeneralizations()} → \text{select(a_implAbstraction)} ^ \text{self.allGeneralizations()} → \text{select(a_implAbstraction).allChildren()} = \text{self.allClasses()} \rightarrow \text{select(a_implAbstraction).allParents()} = \text{self.allClasses()} \rightarrow \text{select(a_implAbstraction)} \} \]

- Simplify the expression post7^pre8^post8.
- The OCL expressions `self.allClasses() = self.allClasses()@pre → including (z)` and `self.allClasses() → exist(z)` evaluate the collection of classes to determine if the element `z` is an element in the collection.

Since the `@pre` construct provides the capability to reference the source model during the transformations, the constraint containing the `@pre` construct is used in the composed postconditions.

The pre- and postcondition, with `z` changed back to `a_implAbstraction`, for the composition of the pre- and postconditions (pre^post7') with the pre- and postconditions for Action 8 is shown in Table 7.17.

Table 7.17. Composed pre^post7' with Action 8 (pre8^post8)

<table>
<thead>
<tr>
<th>pre: true</th>
</tr>
</thead>
<tbody>
<tr>
<td>post8':</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
| | select(a_implAbstraction) }

8. Compose pre^post8' with Action 9 pre- and postconditions (pre9^post9).

This step involves composing the pre- and post conditions (pre^post8') shown in Table 7.17 with the pre- and preconditions for action 9 (pre9^post9). There are no variables in post8' or pre9 that affect the
occurrence of variables in the composed precondition and postcondition or to which an existence quantifier can bind.

- Simplify the expression post<sup>8</sup>′′pre<sup>9</sup>′post<sub>9</sub>.

  - The OCL expressions self.allClasses() = self.allClasses@pre → including (a_prodImplementor) and self.allClasses() → exist(a_prodImplementor) evaluate the collection of classes to determine if the element a_prodImplementor is an element in the collection. Since the expressions are evaluating the same condition, only one shall be include in the composed postconditions.

Therefore the composition of pre- and postcondition (pre<sup>′post8′</sup>) with the pre- and postconditions (pre<sup>9′post9</sup>) of action 9 is shown in Table 7.18.

<table>
<thead>
<tr>
<th>Table 7.18. Composed pre′post8′ with Action 9 (pre9′post9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre: true post9′:</td>
</tr>
<tr>
<td>{ self.allClasses() = self.allClasses@pre → including(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → one(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations@pre → including(a_clntProdAbsAssoc) ^</td>
</tr>
<tr>
<td>self.a_client.Abs = self.allClasses() → select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_prodAbstraction).ClntAbs = self.a_client^</td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses@pre → including (a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → one(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations@pre → including (a_implmAssoc) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_implAbstraction).ProdAbs = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_implAbstraction).Imp = self.allClasses() → select(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses@pre → including(a_concAbstraction) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() = self.allGeneralizations@pre → including(a_absGeneralization) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → collect(a_absGeneralization).general = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → collect(a_absGeneralization).specific = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_concAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_prodAbstraction).allChildren() = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_concAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → collect(a_concAbstraction).allParents() = self.a_client.a_prodAbstraction^</td>
</tr>
<tr>
<td>self.allGeneralizations() = self.allGeneralizations@pre → excluding(a_clntProdImplAssoc) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → includes(a_implGeneralization) ^</td>
</tr>
<tr>
<td>self.allClasses() = exist(a_prodImplementor) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → select(a_implGeneralization).general = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → select(a_implGeneralization).specific = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_prodImplementor) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_implAbstraction).allChildren() = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_prodImplementor) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_prodImplementor).allParents() = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_implAbstraction) ^</td>
</tr>
</tbody>
</table>
Composing pre¬post9 with the precondition of the Bridge transformation yields the precondition¬
postcondition (preModel¬ postModel) as shown in Table 7.19.

Table 7.19. Bridge Pattern Model-Level Pre- and Postcondition (preModel¬ postModel)

<table>
<thead>
<tr>
<th>preModel:</th>
</tr>
</thead>
<tbody>
<tr>
<td>self.isValidSource(BridgeMetamodel)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>postModel:</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ self.allClasses() = self.allClasses@pre → including(a_prodAbstraction)</td>
</tr>
<tr>
<td>self.allClasses() → one(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations()@pre → including(a_clntProdAbsAssoc) ^</td>
</tr>
<tr>
<td>self.a_client.Abs = self.allClasses() → select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_prodAbstraction).ClntAbs = self.a_client ^</td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses()@pre → including (a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → one(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations()@pre → including (a_implemAssoc) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_implemAbstraction).ProdAbs = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_prodAbstraction).Imp = self.allClasses() → select(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() = self.allClasses()@pre → including(a_concAbstraction) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() = self.allGeneralizations()@pre → including(a_absGeneralization) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → collect(a_absGeneralization).general = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_prodAbstraction) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → collect(a_absGeneralization).specific = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_concAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_prodAbstraction).allChildren() = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_concAbstraction) ^</td>
</tr>
<tr>
<td>self.allClasses() → collect(a_concAbstraction).allParents() = self.a_client.a_prodAbstraction^</td>
</tr>
<tr>
<td>self.allAssociations() = self.allAssociations()@pre → excluding(a_clntProdImplAssoc) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → includes(a_implGeneralization) ^</td>
</tr>
<tr>
<td>self.allClasses() = exist(a_prodImplementor) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → select(a_implGeneralization).general = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_implAbstraction) ^</td>
</tr>
<tr>
<td>self.allGeneralizations() → select(a_implGeneralization).specific = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_prodImplementor) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_implAbstraction).allChildren() = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_prodImplementor) ^</td>
</tr>
<tr>
<td>self.allClasses() → select(a_prodImplementor).allParents() = self.allClasses() →</td>
</tr>
<tr>
<td>select(a_implAbstraction) ^</td>
</tr>
</tbody>
</table>

Sections 7.4 and 7.5 provide validation techniques of the Bridge transformation pattern. The Bridge pattern can
be validated formally by composing OCL expressed preconditions and postconditions or informally showing that
conformance exist between a model and the Bridge transformation pattern before and after the pattern has been
applied.

7.4 Bridge Transformation Pre- and Postcondition Conformance

A bridge pattern transformation can be validated formally illustrating the mapping between the model-level pre-
and postconditions and the pre- and postconditions expressed at the metamodel-level. The model elements of the
source models are instances of the metamodel elements of the BridgeMetamodel, therefore the model-level pre- and postcondition (preModel^postModel) pairs implies the metamodel-level pre- and postcondition (preMeta^postMeta) pairs.

Figure 7.3. M1 to M2 Mapping Product Abstraction

The diagrams Figure 7.3 thru Figure 7.6 illustrate the mapping between the model-level postconditions and the transformation patterns. As stated previously, the transformation schema and transformation constraints defined the postcondition of the Bridge transformation. The OCL expressions in the “note” boxes represent the constraints at the model-level, and the dashed lines illustrate the mapping of the model-level postcondition to the metamodel level postcondition. For example, in Figure 7.6, the postcondition self.allAssociations() = self.allAssociations()@pre → excluding(a_clntProdImplAssoc) states that the collection of associations after the transformation is equal to the collection of associations in the source model with the exception of the client product implementation association, i.e. the client product implementation association has been deleted from the collection.

Figure 7.4. M1 to M2 Mapping Implementation Abstraction
7.5 Structural Conformance

In the example given in Figure 7.7, the Bridge pattern transformation is illustrated by refactoring the source model structure containing a Display class associated with a specific implementation class (ImageImpl1) using the Bridge design pattern [Gamma et al., 1994] to a design in which the Display class is associated with a class structure that allows the image implementation to be varied.
The source model shown in Figure 7.7 structurally conforms to the source schema given in the Bridge transformation pattern with respect to the bindings as shown in Figure 7.8. The conforming parts of the model are given in the specialized metamodel diagram, that is, the metamodel representation of the source model as shown in Figure 7.9.

The following steps represent an example of a transformation to introduce the Bridge pattern into the valid source model. The model elements added are shown in bold text and shaded boxes.

The first step in the transformation process is to create a product abstraction class (instance of \|ProductAbstraction) and link it to the client class (instance of \|Client) using an association (instance of
The diagram, shown in Figure 7.10, shows the creation of the class Image (instance of ProductAbstraction) and the association display_image_on (instance of ClntProdAbsAssoc). One end of the association (display_image_on) is connected to the Display class, and the other end is connected to the Image class.

Figure 7.10. Add Product Abstraction Class.

The second step involves creating a product implementation abstraction class and connecting it to the product abstraction class. The diagram in Figure 7.11 illustrates that ImageImpl (instance of Implementor) is created and connected to Image (instance of ProductAbstraction) using the association Implement (instance of ImplementorAssoc).

Figure 7.11. Add Product Implementation Abstraction

Next, the association between the client and product implementation classes must be deleted. As shown in Figure 7.12, the association displayed_on (instance of ClntProdImplAssoc) no longer exists between the Display (instance of Client) and the product implementation class ImageImpl1 (instance of ProductImplementor).

The next step involves adding a generalization relationship between the implementation abstraction class and the product implementation class. Figure 7.13 shows that the implementation generalization ImplGen (instance of
ImplementorGeneralization) is created and connected to ImageImpl (instance of Implementor) and ImageImpl1 (instance of ImageImpl) such that ImageImpl1 is a subclass of ImageImpl.

Figure 7.12. Remove Association between Client and Product Implementation.

Figure 7.13. Add Implementation Generalization

Figure 7.14. Add Product Implementation Class

Figure 7.14 shows the addition of a new product implementation class. In the diagram, a product implementation class (ImageImpl2) is created and connected to the implementation abstraction class (ImageImpl) via the implementation generalization (ImplGen).

The diagram produced given in Figure 7.15 is the target pattern at the metamodel-level.
The diagram in Figure 7.16 shows that the target model structurally conforms to the transformation schema in adherence to the restrictions specified by the transformation constraint as specified by the binding.

7.6 Summary

The transformation pattern presented in the chapter formally introduces the Bridge pattern into existing design models that meet the criteria required in the source schema. This pattern represents a structural design pattern. Since structural patterns modify the structure of a source model, it was determined that transformation constraints are not needed in most cases.
Chapter 8
Summary and Conclusion

The objective of this research was to develop a rigorous approach to introducing design patterns into existing UML class diagrams by defining well-formed transformations that constrain how models are modified in order to produce new design models with improved qualities, thereby enabling controlled model evolution.

To achieve this objective, we specified a transformation process, referred to as pattern-based model transformation, which introduces design patterns into existing UML class diagrams. Transformations were defined at the M2 (metamodel) level of the UML architecture to utilize UML’s capability to extend the modeling language. Extending the metamodel satisfies the desire to apply transformations on model elements. We also extended the UML Superstructure specification to include a new package structure, known as the Transformation package. The transformation package represents properties (i.e., transformation patterns) that define the specialized metamodel structure for a metamodel-level design pattern transformation.

Transformation patterns were defined as an extension of the UML metamodel to characterize source and target model elements. The transformation pattern consists of a source schema, a transformation schema and transformation constraints. The source schema specifies the structure of the model elements that must be in the source pattern in order to introduce the design pattern into the source model. The transformation schema, specifies the model elements that are added, deleted and connected during the transformation. The transformation constraint specifies additional restrictions on source and target model elements that cannot be represented in the transformation schema. Transformation patterns were specified for creational, structural, and behavioral design patterns as described in [Gamma et al., 1994]. More specifically, the Abstract Factory, Bridge and Visitor patterns were chosen to illustrate the representation of metamodel-level transformations because they express the commonality and complexity needed to fully realize the ability of model-level transformations.

An action language, known as PBAL, was developed to provide constructs that add, delete, retrieve and connect model elements. We defined a transformation specification (i.e., program) to implement a model-level transformation on UML class diagrams. The transformation specification uses PBAL constructs for the implementation of actions that transform model elements.
Metamodel-level pre- and postconditions were defined for each transformation pattern to validate the model-level transformation specification against the metamodel-level transformation.

We used two techniques to validate pattern-based transformations on UML class diagrams. One technique verifies structural conformance of the source and target models against the transformation pattern. By verifying the bindings between the source and target models and the source and target patterns, we were able to determine (1) if the design pattern could be applied to the source model, and (2) if the application of the transformation schema and transformation constraints to the source model produced a valid target model. For the cases used in this research, we were able to show that both binding conditions held.

In the second method of validation, we defined an approach for composing pre- and postconditions based upon the combination of Catalysis [D'Souza & Wills, 1998] and Z schema composition [Woodcock & Davis, 1996]. After composing the model-level pre- and postconditions, we were able to verify the model-level transformation specification conforms to the transformation pattern.

By showing conformance between the source model and the source schema and the transformation specification and transformation pattern, we can explicitly infer that the target model conforms to the target pattern.

### 8.1 Contributions

This research makes the following contributions to model transformations:

1. Provides a new pattern-based model transformation method to implement a design pattern as a design model instead of code model. In this method, metamodel-level structures were defining to represent the transformation (i.e., restructuring) of models to include design patterns.
2. Provides an extension to the UML metamodel by specializing the model elements to represent design pattern transformations and an extension the UML package structure by specify a Transformation package.
3. Provides an action language for the specification of transformation on model elements. The significance of this contribution is that we were able to define an action language at the M2 level.
4. Confirms that the RBML concept can be used to express pattern-based model transformations.
5. Supports the definition of transformation patterns for creational, behavioral and structural design patterns.

### 8.2 Future Work

Transformation patterns need to be developed for additional design patterns. The work developed in this research only considered creational, behavioral and structural patterns. Design patterns have also been categorized
as organizational patterns, architectural patterns, process patterns, concurrency patterns, analysis patterns and more. The study and implementation of these patterns require reference to the 23 patterns from [Gamma et al., 1994]. Defining transformation patterns for the additional design patterns defined in [Gamma et al., 1994] provides a foundational definition of transformation patterns for other categories of patterns, which also can be represented as model-level transformations.

Further development is required to expand the applicability of Transformation patterns. The work presented in this dissertation considered only the introduction of design patterns into UML class diagrams. Defining transformation patterns for other UML diagrams, namely the sequence and state diagrams, will allow transformations to be defined between models.

The potential of pattern-based model transformation to support the development of tools has been illustrated by the manual process shown in this dissertation. Developing a tool would further enhance the capability of this approach to support the potential semi-automation of a process to introduce design pattern in UML diagrams.

This research is extendible to perform roundtrip engineering of the transformation process. Currently the transformation of a source model produces a new design model with improved qualities. Roundtrip engineering would provide methods for mapping the target model to the source model, and for synchronizing the models after transformation by keeping them consistent. This would enable the software engineer to freely move between different representations of UML diagrams.
References


Appendix
Reuseable Mini-Transformations

The operations given in Tables A.1 - A.6 represent the reusable mini-transformations defined during this research.

Table A.1 specifies the _connectClasses_AssociationRole operation that connects two class instances to an association model element. |AssociationRole specifies the rolename of the association role which is linked to the classes. The parameter an_association specifies the |AssociationRole model element connected to the parameters a_class1 and a_class2 (instances of |ClassRole). The precondition ensures that the model elements passed as parameters into the operation are elements in the collection of classes and associations. The action clause creates a link that connects the two classes to an association. The postcondition ensures that the association-ends are connected to opposite classes.

Table A.2 specifies the _connectClasses_DependencyRole operation that connects two classes via a dependency relationship. |DependencyRole specifies the rolename of the dependency to link to the classes. The parameter a_dependency specifies the |DependencyRole. The parameters a_client and a_supplier are instances of |ClassRole that are connected to the parameter a_dependency (instance of |DependencyRole). The precondition specifies that model elements passed as parameters into the operation must be an elements in the collection of classes and dependencies. The action clause creates a link that connects the supplier and client model elements to a dependency.
model element. The _create_link_SupplierDep() connects the supplier end of the dependency relationship to the supplier class. The _create_link_ClientDep() connects the client end of the dependency relationship to the client class. The postcondition ensures that the dependency is correctly connected to the client and the supplier.

|Table A.2. Link Classes via Dependency|

```c++
/* connect Client class to Supplier class via Dependency Relationship*/
context Package :: _connectClasses_DependencyRole(a_dependency : DependencyRole, a_supplier : ClassRole, a_client : ClassRole)
pre:
  self.allClasses() → exist(a_supplier) --supplier class exist
  and self.allClasses() → exist(a_client) -- client class exist
  and self.allDependencies() → exist(a_dependency) -- dependency exist
action:
  _create_link_SupplierDep (a_dependency, a_supplier); // connect Dependency to supplier (Factory)
  _create_link_ClientDep (a_client, a_dependency); // connect Dependency to Client
post:
  self.allDependencies() → select(a_dependency).client = client -- client end of a_dependency is client
  and self.allDependencies() → select(a_dependency).supplier = supplier -- supplier end of a_dependency is supplier
```

The operation _connectClasses_GeneralizationRole, given in Table A.3, connects two classes via a generalization relationship. |GeneralizationRole specifies the rolename of the generalization to link between the classes. The parameters a_generalization specifies that the |GeneralizationRole model element creates a specialization of a parent class. The parameters a_parent and a_child are instances of |ClassRole linked to the |GeneralizationRole model element. This operation, similar to the one in Table A.2, specifies a generalization relationship by replacing the dependency model element with generalization model elements and client and supplier classes with general and specific classes, respectively. Note that “general” refers to the parent class while “specific” refers to the child class. The precondition ensures that the collection of classes contains an element that conforms to the parent and child classes, and the collection of generalizations contains an element that plays the role of a_generalization. The action clause creates a link that connects the general end of the generalization to the parent model element and the specific end to the child. The postcondition ensures the generalization is properly connected to the parent and child such that the parent is linked to the general end of the generalization and the child is connected to the specific end.

The operation _connect_OP2Class, given in Table A.4, creates a link between an operation and the class that owns the operation. The parameter “a_class” specifies a class that owns the operation specified by the parameter “op”. The precondition verifies that the operation and class exist in the collection of operations and classes within
the package. The action clause creates a link between the operation and the class. The post-condition verifies that
the class owns the operation specified by the input parameter “op”.

**Table A.3. Link Classes via Generalization**

```plaintext
/* connect Parent class to Child class via Generalization */
context Package :: _connectClasses_GeneralizationRole(a_generalization : GeneralizationRole, a_parent : Class, a_child : Class)
pre:
  self.allClasses() \rightarrow exist(a_parent) -- parent class exist
  and self.allClasses() \rightarrow exist(a_child) -- child class exist
  and self.allGeneralizations() \rightarrow exist(a_generalization) -- generalization exist
action:
  _create_link_ParentGen (a_parent, a_generalization); // connect Generalization to Parent
  _create_link_ChildGen (a_generalization, a_child); // link Generalization to Child
post:
  self.allGeneralizations() \rightarrow select(a_generalization).general = self.allClasses() \rightarrow select(a_parent) -- parent end (general) of a_generalization is a_parent
  and self.allGeneralizations() \rightarrow select(a_generalization).specific = self.allClasses() \rightarrow select(a_child) -- child end (specific) of a_generalization is a_child
```

**Table A.4. Connect Operation to Class**

```plaintext
/* connect a behavioral feature to a class */
context Package :: _connect_Op2Class (op : Operation, a_class : Class)
pre:
  self.allOperations() \rightarrow includes(op) // operation exist in package
  and self.allClasses() \rightarrow includes(a_class) // a_class exist
action:
  _create_link_Op2Class(op, a_class); // link Operation to Class
post:
  self.a_class.ownedoperation = op // ownedoperation end of a_class is op
```

The operations, _get_instances() and _get_operations(), given in Tables A.5 and A.6, obtain a collection of
instances of model elements. The parameter MMClass specifies the metamodel element of which instances are to be
obtained. The _get_instances() operation, shown in Table A.5, returns the set of classes that plays the role of
MMClass. The action clause obtains a set of MMClass instances using the _retrieve_classInstances() action. The
postcondition specifies the result returned by the action is of type MMClass.

The _get_operations() operation, shown in Table A.6, returns the set of instances that play the role of the
behavioral feature, specified by the “MMOperation” parameter. The MMOperation are operations owned by the
metamodel element specified the “MMClass” parameter. The action clause obtains a set of operations conforming to
the MMOperation operation. The postcondition specifies the result returned by the action is of type MMOperation.
Table A.5. Get Instances of a Class

/* Get all instances of a metamodel class */
context Package :: get_instances(MMClass : ClassifierRole) : Set(ClassifierRole)
pre:
  true
action:
  _retrieve_classInstances(MMClass)
post:
  p = result and p.oclIsTypeOf(MMClass) = true

Table A.6. Get Instances of an Operation

/* Get all instances of metamodel operations that are owned by instances of MMClass */
context Package :: get_operations(MMClass : Class, MMOperation : BehavioralFeatureRole) :
  Set(BehavioralFeatureRole)
pre:
  true
action:
  op ::= _retrieve_operationInstances(MMOperation, MMClass);
post:
  p = result and p.oclIsTypeOf(MMOperation) = true
Vita

Sheena Judson Miller was born on March 1, 1965. She is a native of Zachary, Louisiana. She is the fourth and youngest child of Jessie Hickman Judson and the late Albert Judson.

She received her Bachelor of Science in Electrical Engineering from Southern University in 1987 and her Master of Science in Computer Science from Southern University in 1995. She received the degree of Doctor of Philosophy in Computer Science in December, 2004. She is a member of IEEE Computer Society, Association of Computing Machinery (ACM), and Alpha Kappa Alpha Sorority, Incorporated.

Since 1987, Mrs. Judson Miller has worked in the Aerospace and Defense Industries developing software for various defense projects. She is currently employed at Lockheed Martin Aeronautics in Fort Worth, Texas as a Software Engineer.

Mrs. Judson Miller, her husband Blaine, and daughter Joei Lea reside in Fort Worth, Texas.