DISSERTATION

TRANSFORMING UML CLASS MODELS

Submitted by

Devon Michael Simmonds

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2007

COLORADO STATE UNIVERSITY

March 15, 2007

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY DEVON MICHAEL SIMMONDS ENTITLED TRANSFORMING UML CLASS MODELS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

James M. Bieman, Ph.D., Committee Member

Gregory L. Florant, Ph.D., Committee Member

Sudipto Ghosh, Ph.D., Co-Adviser

Robert B. France, Ph.D., Adviser

L. Darrell Whitley, Ph.D., Department Head

ABSTRACT OF DISSERTATION

TRANSFORMING UML CLASS MODELS

In a model driven development (MDD) environment, developers create and evolve applications by creating models and transforming abstract models to more concrete models. To realize the benefits of MDD, model transformation languages are needed. The MOF 2.0 Query View Transformation (QVT) Language is an Object Management Group's (OMG) standard for specifying model transformations. QVT transformations are specified explicitly (in terms of) using instances of metamodel level classes. Using QVT to specify transformations on moderately-sized UML class models results in large object-level specifications that can be tedious to read and understand.

This dissertation presents a language for specifying class model transformations at a higher level of abstraction than the level of instances of metamodel classes. The language leverages the UML class model notation, and is used to create transformation schemas that consist of transformation directives. An interpreter for performing the transformation is also presented. The interpreter performs the transformation by processing the directives found in the transformation schema. The interpretation algorithm is described in this dissertation.

To demonstrate the use of the transformation technique, platform-independent class models describing transaction and distribution features are transformed into

platform-specific class models describing CORBA and Jini realizations of the features.

Devon Michael Simmonds
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Spring 2007

ACKNOWLEDGEMENTS

I thank God who is the source of my life, for the grace, mercy, strength and blessings that He generously provided us during this research. His goodness and His mercy have sustained us.

Thanks to my advisors, Dr. Robert France and Dr. Sudipto Ghosh for their support and guidance during the process. Special thanks to Dr. France who provided support during his sabbatical when he certainly could have used his time for other purposes. Thanks as well to Dr. James Bieman and Dr. Greg Florant my other committee members.

During my years at CSU, I enjoyed excellent relationships and consistent financial support from the Computer Science Department. In that regard, I would like to thank my advisors and all departmental administrators and staff involved. These include: Dr. Dale grit, Dr. L. D. Whitley, Susan Short, Sharon Vangoirder and Carol Calliham. Special thanks as well to Dr. Raghu Reddy, Arnor Solberg and the other members of the CSU software engineering research group who all made a positive contribution to my program. My collaborations with Raghu and Arnor were the best I had with other students during my stay at CSU and I wish to express my thanks to them. Thanks as well to Dr. Eunjee Song and Dr. Trung Thanh Ding Trong.

Words are inadequate to express my sincere gratitude to my wife Marie. This dissertation would not have been completed without her love, care, many devoted

prayers and consistent support. While it is only my name that will appear on the CSU academic transcript, I am very happy that in the courts of heaven, there is another transcript, one more noble and more glorious - with her name on it!

Finally, thanks to my relatives and friends for their constant love, prayers and support. This includes my parents Vivian and Icylin Simmonds; Dr. Devon Lynch and Jennifer Lynch; the pastors and members of the Jamaica Conference of the Church of God (7th day); the pastors and members of the Church of God (7th day) in Denver, Colorado; Faith Gordon; Lennox and Mercedes Deane; Winnifred Taylor; Paul Anderson; Barbara Angella Smith; and Norris George McDermott, the great Man of God.

DEDICATION

I dedicate this dissertation to my lovely, loving, beautiful, darling wife Marie: as a trophy to good wifing, selfless love and warm generosity. Thanks Beautiful!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Models can aid the process of understanding, creating and evolving complex software systems [33]. A model is an abstract representation of a system or entity, that is, a model does not describe all the properties of the entity being represented, but describes only selected properties depending on its purpose. Model Driven Development (MDD) [32, 33] aims to leverage the benefits of models in software engineering. MDD methods take a model-centric approach to software development in which applications are created by transforming abstract models to concrete implementations.

A model transformation is a process that takes as input one or more source models and produces one or more target models [11, 23]. For example, model transformations may be used to integrate multiple source models, divide a single source model into multiple target models, add details to source models or remove details from source models.

1.1 Problem Statement

To realize the benefits of MDD, model transformation languages and mechanisms are needed. A model transformation language provides constructs for specifying transformations. The MOF 2.0 Query View Transformation (QVT) Language [27]

is an Object Management Group's (OMG) standard for specifying model transformations. QVT includes an operational mappings language, a core language and a relations language. The relations language and the core language are designed for describing declarative transformation specifications. In contrast, the operational mappings language is designed for describing imperative implementations of transformations.

The relations language includes equivalent textual and graphical syntax notations. In the graphical notation, a transformation is expressed as a relationship between a source domain and a target domain, where a domain is a pattern that is specified as an object diagram. The source domain pattern describes valid input or source models and the target domain pattern describes valid output or target models.

Each model element in the source domain is an instance of a class in the source metamodel and each model element in the target domain is an instance of a class in the target metamodel. The source and target metamodels are expressed using the OMG's MetaObject Facility (MOF) [28]. The MOF is a standard for creating metamodels. The MOF was used to create the Unified Modeling Language (UML) 2.0. metamodel [46]. The UML is an OMG modeling language standard.

Specifying transformations on UML models using the QVT relations language requires one to work at the level of instances of metamodel classes. Describing models in terms of metamodel class instances can produce large descriptions, and expressing transformations at this level of granularity can be tedious for medium to large-sized models. For example, a QVT source pattern that describes class models consisting of two classes with one attribute each, and one association between the classes, will contain instances for the classes, the attributes, the attribute types, the association and the association ends, that is, at least 9 model

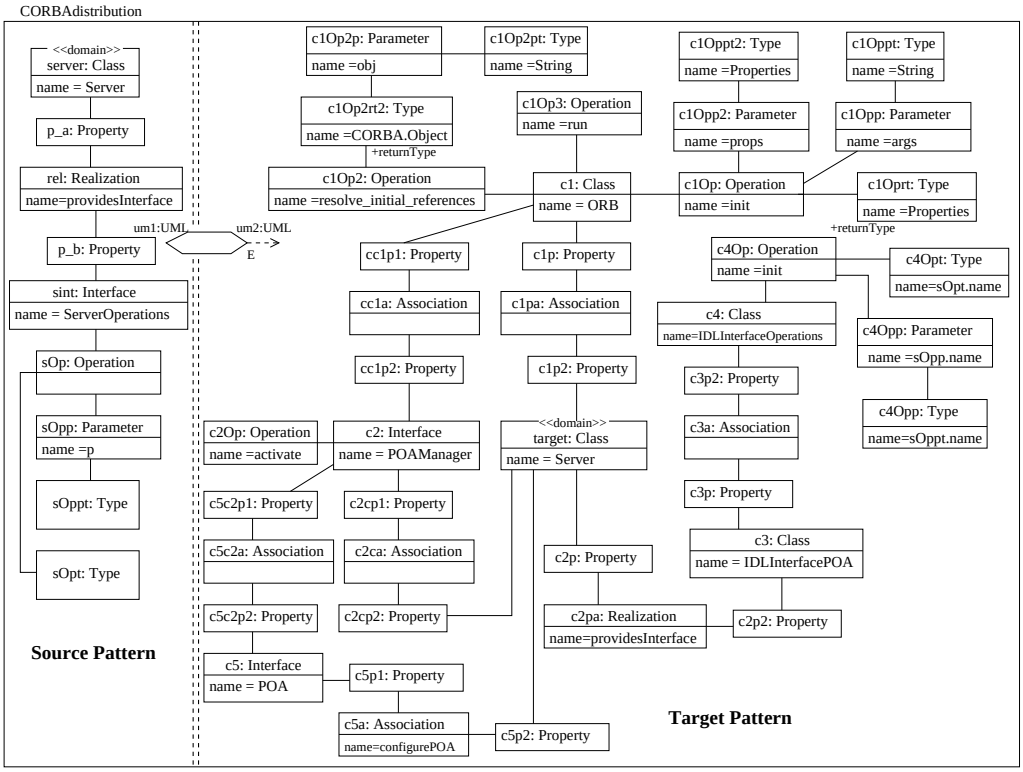

Figure 1.1: QVT Server Distribution Transformation.

elements.

In addition to being large, QVT specifications are difficult to understand because it is difficult to differentiate between types of model elements if they are all represented as instances of metamodel classes. For example, Figure 1.1 shows a QVT transformation specification for transforming a platform-independent server distribution class model into a CORBA distribution class model. The figure includes a source pattern and a target pattern. The dotted lines between the source pattern from the target pattern is added to separate the two patterns and is not a part of the QVT specification.

The source pattern describes valid source or input models and the target pattern describes valid target or output models. The source pattern includes nine

UML object symbols and eight links. A valid source model is shown in Figure 1.2. The source model consists of only a UML class, a UML interface with two operations and a realization dependency between the class and the interface. Figure 1.3 shows a valid target model consisting of nineteen model elements (one interface, five classes, eight operations and five relationships). In comparison, the target pattern consists of eighty three model elements (forty three object symbols and forty links).

The example illustrates that describing families of small models at the level of instances of metamodel classes can be tedious since patterns and relationships among model elements are expressed in terms of explicitly defined object structures. There is a need to raise the level of abstraction at which transformations are specified above the level of instances of metamodel classes. Given two specifications, `Specification-A` and `Specification-B`, `Specification-A` is said to be at a higher level of abstraction than `Specification-B`, if a unit of specification in `Specification-A` describes one or more units of specification in `Specification-B`. This dissertation proposes a transformation language that raises the level of abstraction at which transformations are specified above the level of instances of metamodel classes. The new language defines units of specification, each of which describes one or more instances of metamodel level classes.

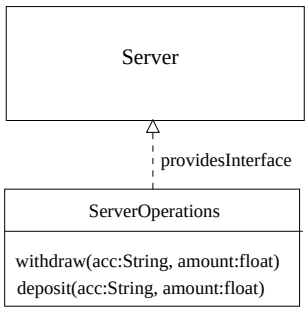

Figure 1.2: A Source Model.

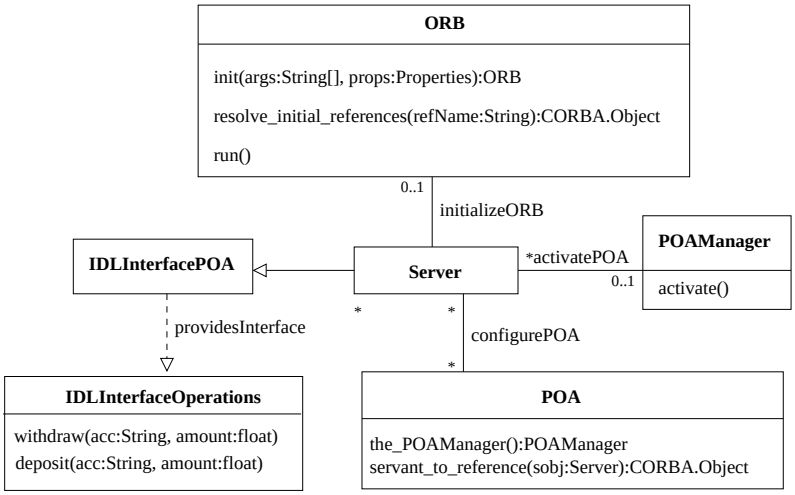

Figure 1.3: The Target Model.

1.2 Solution Overview

This dissertation addresses the problems associated with the use of QVT for specifying class model transformations. The proposed transformation technique consists of a graphical transformation language that raises the level of abstraction at which transformations are specified above the level of instances of metamodel classes, and a mechanism that takes a transformation specification expressed in the language as input, and performs the specified transformation on instances of metamodel classes.

It is advantageous if available modeling tools can be used to specify transformations. We designed the language so that existing UML tools can be used to create the transformation specifications. The language leverages the UML class diagram notation so that the transformation specifications more directly reflect how the target model elements are produced from source model elements.

An overview of the technique is described in the activity diagram shown in Figure 1.4. The activities shown in the diagram are described below:

1. The `Develop Source Pattern` activity produces the `Source Pattern` that describes the set of properties that valid source models must possess. Each source model is an instance of the source pattern.

2. The transformation specification is created in the `Specify Model Transformations` activity. A transformation specification is called a `Transformation Schema` and it includes imperative statements called directives that stipulate how `Source Model` elements are transformed into `Target Model` elements.

3. The directives in the transformation schema are processed in the `Process Transformation Schema` activity. The input to this activity are the `Source`

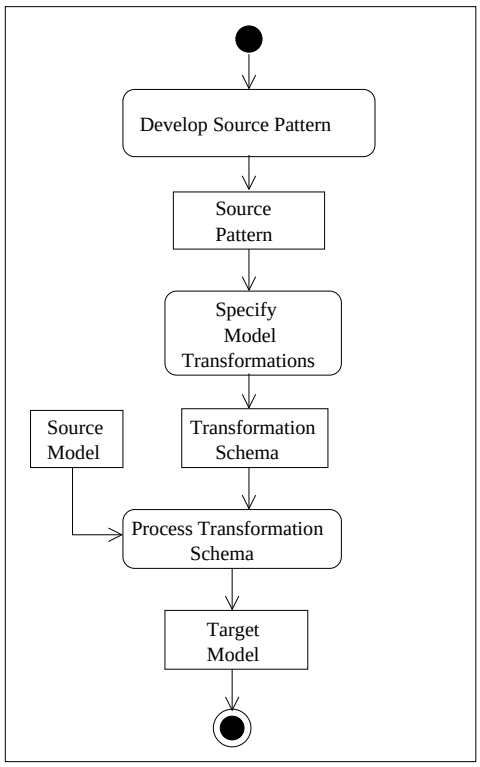

Figure 1.4: Model-to-model Transformation Process.

`Model` and the transformation schema. The activity outputs the `Target Model`.

1.3 Contribution and Scope of Dissertation

The primary contribution of this dissertation is a model transformation technique consisting of a graphical model transformation language that raises the level of abstraction at which transformations are defined and an associated interpreter that performs the specified transformations on metamodel class instances. The technique is demonstrated by using it to transform platform-independent transaction and distribution class models into platform-specific CORBA and Jini class models.

The key feature of the transformation schema language is that the language raises the level of abstraction of transformation specifications above the level of instances of metamodel classes. This is accomplished by leveraging the use of UML class model notation.

The model transformation language was created to transform UML class diagrams because class diagrams are the most widely used models in object-oriented modeling. One can expect that there will be many cases in which transformations on class diagrams will be needed. This dissertation provides a tailored language for specifying class diagram transformations.

1.4 Structure of Dissertation

Chapter 2 provides background information and Chapter 3 discusses related research. Chapter 4 presents the technique for transforming class models using transformation schemas. The technique is applied to the transformation of distribution and transaction class models in Chapters 5 and 6. Chapters 5 presents the transformation of a distribution class model to CORBA and Jini class models and Chapter 6 presents the transformation of a transaction class model into CORBA and Jini class models. The conclusion and future research is presented in Chapter 7.

Chapter 2

Background

This chapter includes an overview of model transformation, an overview to model driven development, a description of how model patterns are expressed and a description of CORBA and Jini middleware features.

2.1 Model Transformation

Model transformation [2, 5, 11, 14, 23, 29, 35, 36] is a process that takes as input one or more source models and produces target models. Model transformations can be classified into various types, based on the way the source models are changed to realize the target models, for example [11]:

1. `Composition` is a transformation in which multiple source models are integrated.

2. `Anti-composition` is a transformation in which a single source model is divided into multiple target models.

3. `Refactoring` is a transformation in which a model is reorganized into different parts at the same level of abstraction.

4. `Refinement` is a transformation in which more detail is added to a model.

5. `Abstraction` is a transformation in which detail is removed from a model.

The transformation language presented in this dissertation may be used to effect abstraction, refinement, refactoring and anti-composition. Composition is beyond the scope of the dissertation. Abstraction may be effected by excluding model elements from a source model using exclude directives. Refinement may be effected by using transformation directives (e.g. new) to add model elements to a source model. Refactoring may be effected by using transformation directives (e.g. new, source, exclude) to reorganize the model elements in a source model. Anti-composition may be effected by specifying a transformation model (i.e. a transformation schema) with two or more sub-schemas each of which results in a separate target model.

Vertical transformations take source models at one level of abstraction and produces target models at another level of abstraction, while *horizontal* transformations have source and target models at the same levels of abstraction [10, 11, 23]. Refinement and abstraction are vertical transformations while composition, anti-composition and refactoring are horizontal transformations.

Model transformations may also be classified based on the way transformations are specified. A `declarative` transformation specification defines a relationship between source model and target model elements. An `imperative` or `operational` transformation specification defines a transformation in terms of actions performed on model elements.

Model transformations are also classified based on the distinction between source and target models [2]. *Model-to-model* transformations produce one or more output models from one or more input models while *model-to-code* transformations produces source code from input models. Model-to-code transformations are a special case of model-to-model transformations where the output model is

code. Model-to-code transformations are a special case of *model-to-text* transformations where the output model is code. In model-to-text transformations the output model may be text, such as source code, XML documents and configuration information.

2.2 Model Driven Development (MDD)

In model-driven development (MDD) [32, 33, 37] design models are the central artifacts of software development. The Model Driven Architecture (MDA) [31, 42, 43, 44, 45] is the most well known MDD initiative. The MDA is the work of the Object Management Group (OMG) [47], a large international trade association, that seeks to help reduce complexity, lower costs, and hasten the introduction of new software applications. In order to achieve these goals the OMG has provided a number of resources and standards, including the Unified Modeling Language (UML), the Meta-Object Facility (MOF), the Common Warehouse Metamodel (CWM), the Object Management Architecture (OMA), and the Model Driven Architecture.

The MDA has two primary principles, (1) models should be the primary artifacts in developing software, and (2) application development should be separated from the integration of a specific technology. The MDA proposes an architectural separation of concerns that articulates three viewpoints of a system:

1. A Computation Independent Model (CIM): A representation of the system requirements from a business perspective without considering software concerns.

2. A Platform Independent Model (PIM): A representation of a system that ignores details related to specific platforms. It captures those elements of a system that remain the same from platform to platform.

3. A Platform Specific Model (PSM): A representation of a system that combines platform independent information with information related to a specific platform.

The MDA proposes the development of transformations to support the transformation of models across different abstraction levels, for example a PIM to PSM transformation or a PIM to PIM transformation.

2.3 Representing Model Patterns Using Templates

In this research, model patterns are used to specify the metamodel of source models. A model pattern is described using a variant of the Role Based Metamodeling Language (RBML) [7, 6, 8, 9, 21, 20, 19]. RBML is a UML [46] based language that supports rigorous specification of pattern solutions, where a pattern solution characterizes a family of solutions for a recurring design problem. In RBML, structural design diagrams are specified using class diagram templates. Class diagram templates have template model elements that are explicitly marked using the "|" symbol.

A class diagram template consists of parameterized class diagram elements, for example, class templates and association templates. A class diagram template defines a family of class diagrams where each class diagram is obtained by binding the parameters to actual values. An example of a class diagram template is shown in Figure 2.1 and the instantiated class diagram is shown in Figure 2.2.

A class template consists of two parts: attribute templates and operation templates. Attribute templates produce attributes when instantiated, and operation templates produce operations when instantiated.

The class diagram template shown in figure 2.1 consists of the following class

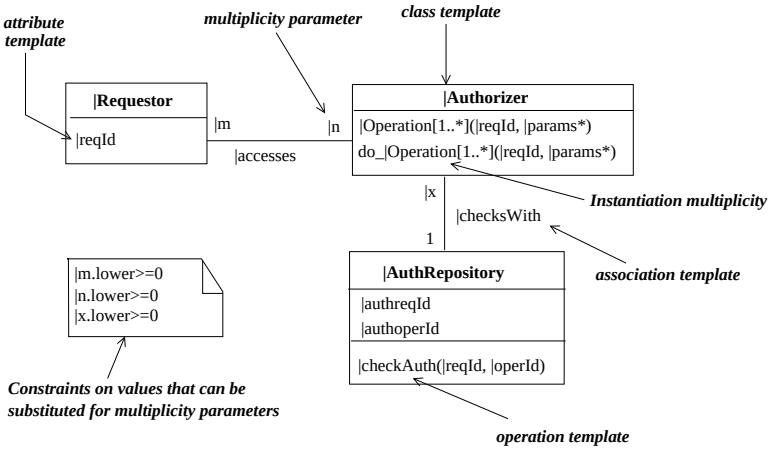

Figure 2.1: Example of A Class Diagram Template

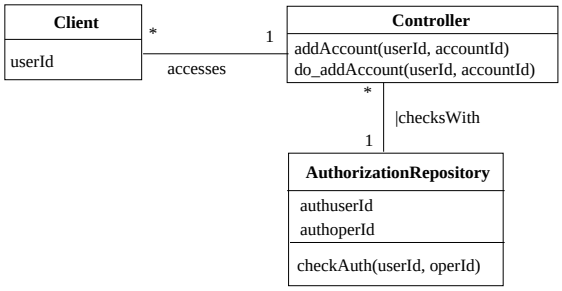

Figure 2.2: Example of An Instantiated Class Diagram

templates: |*Requestor*, |*Authorizer*, and |*AuthRepository*. |*Requestor* has an attribute template named |*reqId*. The instantiation of |*reqId* is the *userId* attribute. |*Authorizer* has a operation templates named |*Operation* and *do_*|*Operation* . Other attribute and operation templates are similarly specified. The instantiation of operation template |*Operation* is the *addAccount* operation.

The template metamodel shown in Figure 2.3, defines types of modeling elements for representing class templates. Each template type is a specialization of a UML metaclass, for example, Classifier Template is a specialization of UML Classifier. Instances of these elements are specialized classes using the same syntax as the parent UML metaclass. `Class Template` instances, for example, are specified

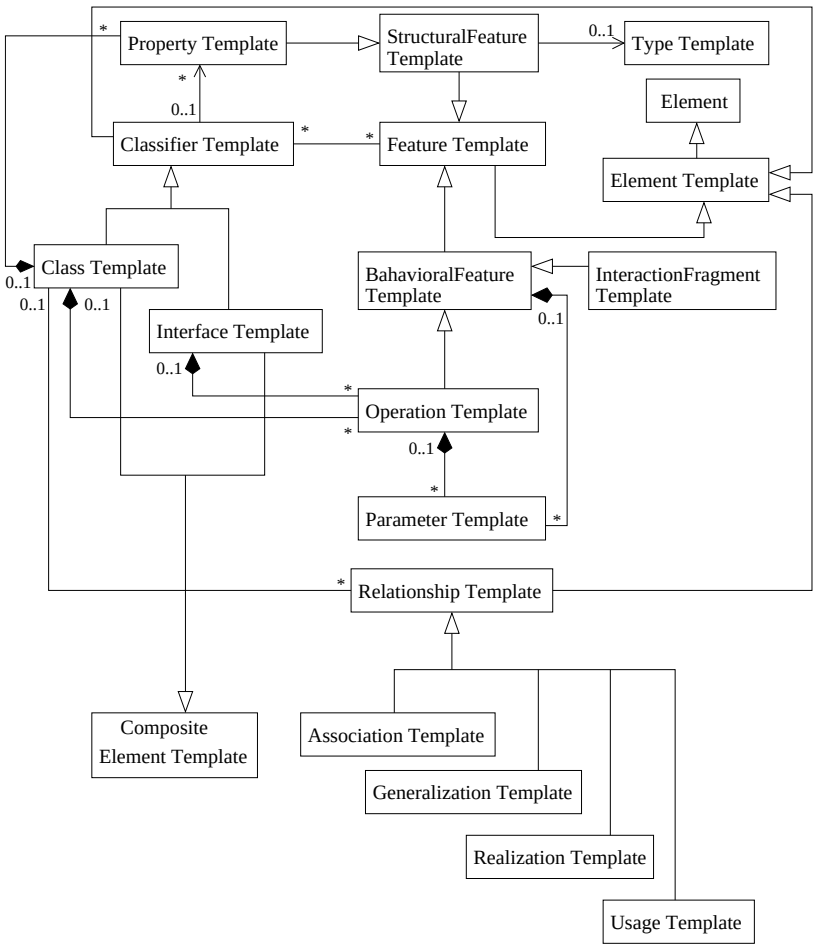

Figure 2.3: Class Template Metamodel.

using UML Class notation. When a model element template is instantiated, the result is an instance of the parent UML metaclass. Class templates, for example, result in UML classes when instantiated and operation templates result in UML operations.

2.4 Middleware Technologies

Middleware platforms are designed to enable the development of complex distributed systems by hiding infrastructural details from the application program.

2.4.1 Jini

Jini is a Java-based middleware technology that enables a federated group of users to share distributed services over a local area network [4, 38, 39, 51]. A federation consists of a collection of users, software, and devices that share a basic understanding on issues of policy, identification, administration, and trust. A service is any resource on the network that can be used by members of the federation. Services may be hardware, software or a combination of both, for example: application programs, operating systems, servers, hardware devices and appliances (e.g., printers) and storage. The main goals of Jini are:

1. To enable the sharing of services and resources over the network
2. To enable all persons and entities in the federation to have the flexibility to add and remove services randomly.
3. To provide location transparency to users (their network locations can change randomly), and easy access to resources on the network.
4. To simplify the construction, modification, and maintenance of a federation.

A federated group becomes a single, dynamic distributed system, and groups of federated entities can be federated into even larger systems. Jini extends the Java application environment from a single virtual machine to a collection of virtual machines. Jini built-in facilities include lookup, event, leasing, and transaction services.

1. The Lookup Service: Jini supports dynamically adding and removing services through the `lookup service`. The lookup service forms a repository where proxies for services are stored. A proxy is an application that knows how and where to access the actual service. The use of proxies shields the

actual service from direct manipulation and facilitates transparent addition and removal of services. When a new entity wishes to join the federation, it first locates a lookup service using the Jini discovery protocol, and then joins the federation (using the Jini join protocol) by registering its proxy with the lookup service. The lookup service will then inform all members of the federation of the availability of the new services.

2. Events - Jini supports a distributed events model that allows one object to receive a notification of events in another object.

3. Leasing - a lease is a grant of guaranteed access to a resource for a given time period. Leases may be either exclusive (only one user per time period), on non-exclusive (multiple users allowed per time period). A lease is negotiated between client and service, and may be renewed, although a renewal request may be denied.

4. Transactions - A transaction is a indivisible collection of operations between servers and clients that remain atomic even if some clients and servers fail. Jini provides a transaction interface that specifies the service protocol used to coordinate a two-phase commit.

2.4.2 Common Object Request Broker Architecture (CORBA)

CORBA [12, 41, 48] is an OMG [47] standard for open distributed object computing. CORBA builds upon the core object model of the OMG's Object Management Architecture (OMA) and provides a framework for interoperability, and a set of mappings from its interface definition language (IDL) to implementation languages. The OMA is the OMG framework within which all technologies

adopted by the OMG fits. CORBA provides a comprehensive set of services and facilities including:

1. The CORBA **naming** Service allows objects to be located by the name of the object. Objects do not have fixed names, instead the name of an object is the name bound to the object within a specific context.

2. The CORBA **event** Service supports communication between objects where one object called a `supplier`, creates event data and and another object called a `consumer` object, receives and processes the event-based data.

3. The CORBA **security** Service provides facilities that address confidentiality, integrity, accountability and availability. The main features of the CORBA security service are: identification and authentication, authorization and access control, security auditing and administration of the security service.

4. The CORBA **transaction** Service provides the facilities through distributed transactions are managed in CORBA. A transaction is a set of discrete operations that occur in time where all the operations are either committed or rolled back. Committing a transaction means the results of the transaction are accepted by participating clients and servers. Transactions are normally committed when the `ACID`[1] properties of the transaction have not been violated. Transactions are rolled back otherwise.

Chapter 3

Related Work

This chapter describes related work on transformation languages.

3.1 Kermeta

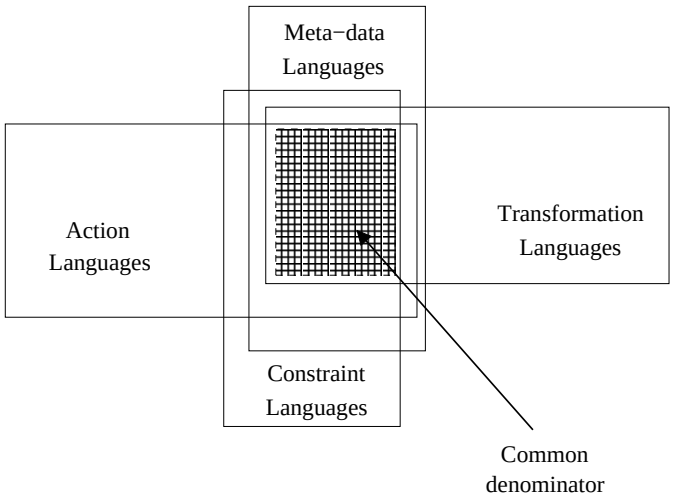

Figure 3.1: Features of Kermeta [50]

Kermata [49] is an open-source imperative non-graphical metamodelling language. A metamodelling language differs from a general purpose programming language like Java, in that a metamodelling language is designed with features to support the design of other transformation languages. Kermeta, for exam-

ple, is designed with features to support the representation and manipulation of metamodels.

Kermeta is designed to be a foundation for implementing metadata languages, action languages, constraint languages and transformation languages as shown in figure 3.1. In the figure, the common denominator are those features that support the implementation of any of these languages. Kermeta is an Eclipse [3] plugin that is developed as an extension to the OMG's Essential Meta-Object Facilities (EMOF) [28].

Many metamodel-based model transformation languages support the specification of software structure but provide no support for the specification of behavior. Therefore, the operational semantics of metamodels cannot be specified using these languages. Kermeta addresses this weakness by adding execution semantics to EMOF. Therefore, Kermeta can be used to define both the structure and behavior of user-designed metamodels.

Kermata differs from the model transformation language proposed in this dissertation in several ways: (1) Kermeta is a general-purpose transformation language while the language proposed in this dissertation is specific to UML class models. (2)Kermeta is designed to be used to specify other transformation languages and the language proposed in this dissertation is not designed for this purpose. (3) Kermeta is a textual transformation language while the language proposed in this dissertation is graphical. (4) The language proposed in this dissertation uses imperative transformation directives. Such directives do not exist in Kermata.

Since Kermeta provides an environment for implementing transformation languages, Kermeta may be used to implement the model transformation language defined in this dissertation.

3.2 The ATLAS Transformation Language (ATL)

The ATLAS Transformation Language (ATL) [13] is a hybrid declarative and imperative model transformation language. In ATL, the source metamodel, target metamodel and transformation metamodel, all conform to the OMG's Meta Object Facility (MOF) core specification [28]. The language is designed to encourage a declarative transformation approach. However, imperative language constructs are provided for cases where a complete declarative approach is difficult. A transformation is effected by executing a transformation definition written in ATL. Transformations are unidirectional, converting read-only source models to write-only target models. Bi-directional transformations are implemented as two unidirectional transformations.

ATL differs from the model transformation language proposed in this dissertation in that ATL is a textual rather than a graphical language. In addition, ATL does not provide a transformation specification syntax based on the syntax of UML class models nor does ATL use imperative transformation directives.

3.3 Visual Model Transformation (VMT)

VMT [34] is a visual, declarative, model transformation language that supports the specification, composition and reuse of model transformation rules. VMT transformation rules uses a visual notation and the Object Constraint Language (OCL) [52] to support the creation, modification and deletion of model elements.

The VMT transformation approach is based on graph transformations. Specifically, a VMT transformation is defined by a set of transformation rules. Each transformation rule is a mapping from source UML diagram elements to target UML diagram elements. A transformation rule is described by a `rule`

`specification` consisting of two parts: a `matching schema` and a `result schema`. The matching schema is represented by a graph that defines:

- The conditions under which transformation rules can fire.
- The input arguments for the transformation.
- The input arguments that will be deleted when the rule is executed.

The result schema is represented by a graph that defines the target model elements based on the elements specified in the matching schema. The VMT approach also supports the use of a `rule ordering schema` that specifies the order of execution of rules.

While VMT is a graphical model transformation language, VMT does not provide a transformation specification syntax based on the syntax of UML class models nor does VMT use imperative transformation directives.

3.4 MOLA

MOLA [15, 17, 18] (**MO**del transformation **LA**nguage) is a graphical model transformation language created at the University of Latvia. The primary goal of the language is the creation of easily readable graphical transformation specifications that make use of simple iterative control structures. MOLA combines traditional structured programming language constructs with pattern-based model transformation rules to create a graphical notation for specifying model transformations.

A MOLA program transforms a source model into a target model where the source model is an instance of a source metamodel and the target model is an instance of a target metamodel. A MOLA program consists of a sequence of graphical statements (rounded rectangles) linked by arrows, and somewhat like a structured flowchart. Two statement types are `rules` and `loops`.

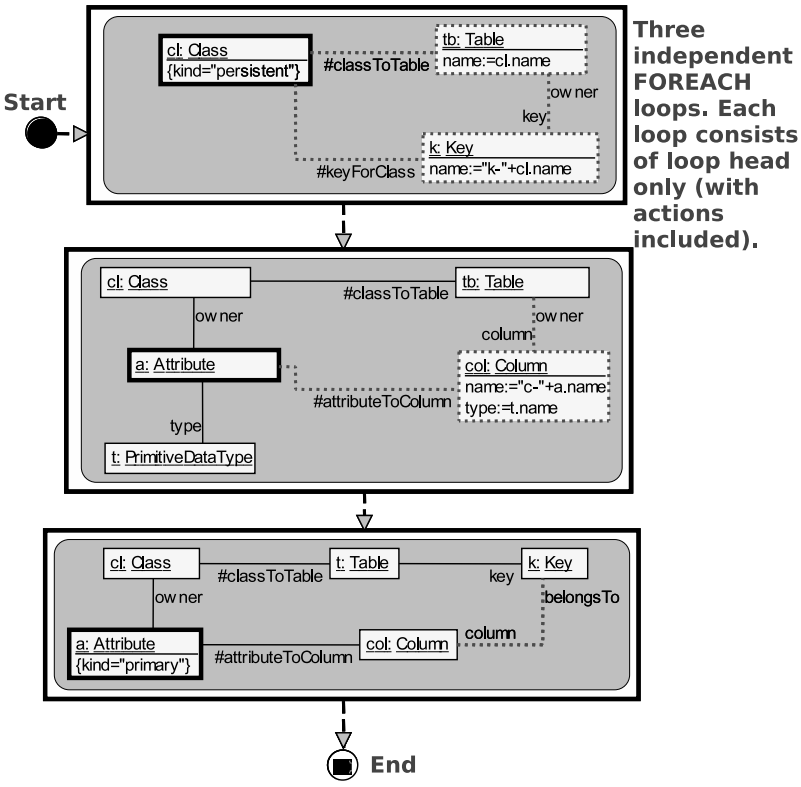

Figure 3.2: MOLA Specification to Transform A Class to A Table [16].

A rule is the simplest kind of statement. Each rule has two parts: a `pattern` and an `action` specification. A pattern is a set of elements representing class and association instances. The pattern is built in accordance with the source metamodel and may have OCL constraints associated with model elements in the pattern. An action specification is a description of new class or association instances to be created or deleted, and the modified attribute values. The semantics of a rule requires the actions to be applied for all pattern instances in the source model.

A loop is represented graphically as a bold-lined rectangle, containing a sequence of statements. This sequence begins with a special statement called a *loop head*, represented by a grey rounded rectangle. A loop head is a pattern, with a

single element, *the loop variable*. A loop variable is highlighted by a bold frame and represents an arbitrary element of the given class. The semantics of a loop requires the execution of the loop for all loop variable instances that satisfy the conditions specified by the pattern. Figure 3.2 shows a MOLA specification to transform a class to a table. Tool support for MOLA has been built [18].

While MOLA is a graphical model transformation language, MOLA does not provide a transformation specification syntax based on the syntax of UML class models nor does MOLA use imperative transformation directives.

3.5 Tefkat

Tefkat [22] is a declarative logic-based transformation language designed for the transformation of MOF models. Tefkat is based on the use of patterns and rules. A Tefkat transformation specification asserts a set of constraints that should hold over a collection of source and target models. These constraints are used to create a set of target models that satisfy the constraints.

Three kinds of models can be represented in a mapping: a set of source models, a set of target models and a single tracking model. A transformation rule may query source models and the tracking models. A transformation rule may also make assertions about target models and the tracking model. The tracking model is therefore the only model that can be both queried and constrained. A rule has two parts: the query and the constraint. A rule also has two sets of variables: those that occur in the query and those that occur only in the constraint. A transformation specification may contain class definitions, rules, pattern definitions and template definitions.

Tefkat differs from the model transformation language proposed in this dissertation in that Tefkat is a textual rather than a graphical language. In addition,

Tefkat does not provide a transformation specification syntax based on the syntax of UML class models and Tefkat does not use imperative transformation directives.

3.6 QVT

The Query View Transformation (QVT) specification [24] includes declarative and operational parts. The declarative part consists of a two-level architecture that has (1) a `relations language` and an associated metamodel and (2) a `core language` and an associated metamodel. The operational part consists of an `operational mappings language` and an associated metamodel.

3.6.1 The Relations Language

The relations language supports the specification of transformations as a set of relations among domains where a domain is a pattern of objects. The relations must hold in order for the transformation to be successful. A relation is a subset of a N-ary product of sets `A1 x A2 x ... x An` where each set Ai is a model type, for example a MOF type. This model type is called a `domain`.

A domain has a pattern that describes valid candidate models, where a candidate model is any model that conforms to the model type. Each domain pattern is a template for locating, modifying and creating objects and their properties in a candidate model to satisfy the relation. Relations also include definition of `rules` for determining the model elements that are to be related. Rules include variables of MOF types, object template patterns, OCL constraints over domains and variables of the relation, and assertions that other relations hold. Before a transformation can be effected, the domain pattern is matched against objects in the candidate model.

The relations language has equivalent textual and graphical forms. Figure

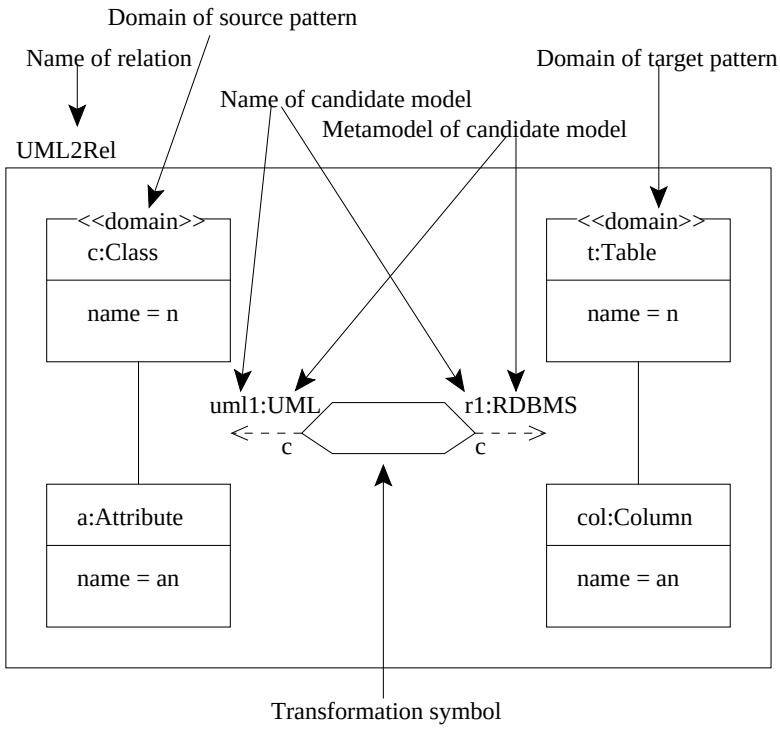

Figure 3.3: UML Class to Relational Table Relation [26].

3.3 shows a class-to-relational-table relation specified using the graphical syntax. In the graphical syntax a relation is described by two patterns: a source domain pattern and a target domain pattern. Each pattern is an object diagram consisting of objects, links and values. In the Figure, the source domain is the `c:Class` domain and the target domain is the `t:Table` domain.

The target domain of a relation may be marked as `checkonly` or `enforced`. A checkonly domain is marked with a `c` on the transformation symbol as shown in the figure. An enforced domain is marked with a `C` on the transformation symbol . When a transformation executes in the direction of a checkonly domain, candidate models are checked to determine if they satisfy the constraints defined by the relation. When a transformation executes in the direction of an enforced domain, if a candidate model does not satisfy a relational constraint, then the target model

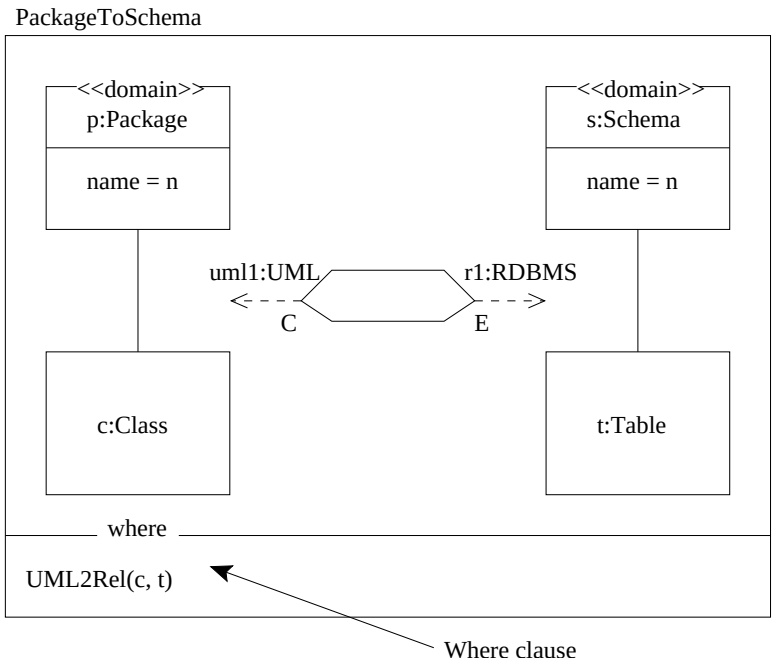

Figure 3.4: QVT Relation With `where` Clause [25].

is modified so that the relational constraint is satisfied. The equivalent textual syntax for Figure 3.3 is the following:

```
relation UML2Rel {
checkonly domain uml1 c:Class {name = n, attribute =
a:Attribute{name = an}}
checkonly domain r1 t:Table {name = n, column =
col:Column{name = an}}
}
```

The relational language allows constraints on relations to be specified using the `when` and `where` clauses. Each `when` clause specifies conditions under which the relation must hold. A `where` clause specifies conditions that must be satisfied by all model elements participating in the relation. Figure 3.4 shows an example

of a `where` clause. The `UMLeRel` relation must hold for each class participating in the `PackageToSchema` relation.

3.6.2 Comparison

While QVT provides a graphical syntax, QVT does not provide a transformation specification syntax based on the syntax of UML class models nor does QVT use imperative transformation directives. Difficulties with using QVT to specify transformation models are presented in Section 1.1.

3.7 Summary and Discussion

Model transformation approaches that support a graphical notation (e.g., MOLA) are desirable in MDD because mappings specified in the notation may be used to visualize and communicate information about a transformation in a way that cannot be done as conveniently with textual notation. VMT, MOLA and QVT support a graphical notation, however, none of these languages have a graphical notation closely related to the notation of the target models of the transformation.

Metamodel-based transformation approaches have many benefits, such as support for domain-specific modeling. However, these approaches may be difficult to use when the metamodels are large and fragmented and transformations are specified using metaclasses. For example, the UML metamodel for classes and interactions as specified in the UML 2 is fragmented, and the fragments are tied together via several other metamodel packages. As a result, using the UML metamodel to specify transformations can be difficult, time consuming and tedious. Moreover, specifying models using the RBML template metamodel, as is done for the model-to-model transformation technique, is more concise that the use of UML metaclasses because the RBML template metamodel elements use the

notation of the UML model elements.

In addition to being graphical and using a notation closely related to the notation of the target models, the model-to-model transformation language provides a small set of transformation directives with well-defined semantics. Although graphical, neither VMT, MOLA nor QVT is based on the use of transformation directives.

Chapter 4

Model-To-Model Transformation

This chapter presents a technique for transforming class models from one level of abstraction to another. We first provide an introduction to the transformation technique, followed by an example of a transformation schema. We then describe the form of transformation schemas, followed by transformation directives used in transformation schemas. Class transformation metamodels, a grammar for the language and a transformation algorithm are also presented.

4.1 Introduction

The transformation technique is illustrated in Figure 4.1. In the figure a `Source Pattern` is created during the `Develop Source Pattern` activity. A `Source Pattern` describes valid source models.

The transformation of a source model into a target model is specified by a class `Transformation Schema`. Transformation schemas are developed during the `Specify Model Transformations` activity. A transformation schema contains imperative statements called directives, that stipulate how target model elements are formed. One can view a transformation schema as a program that takes a source model as input and produces a target model. A transformation is effected by applying a transformation schema to a source model.

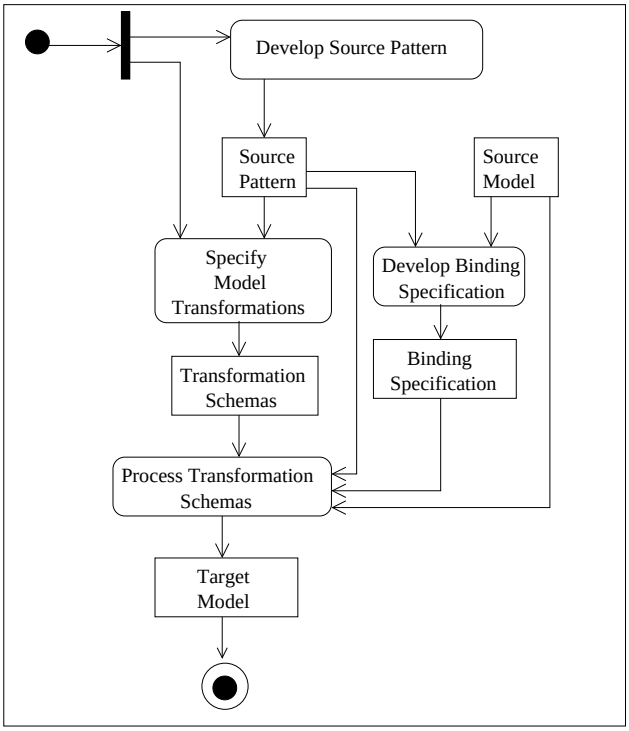

Figure 4.1: Model Transformation Process.

Target models are developed during the `Process Transformation Schemas` activity. The input to this activity are the source pattern, the source models and a `Binding Specification`. Before a transformation can be effected, the source model element that each source pattern element represents must be identified. This is accomplished using the `Binding Specification`. This specification is a listing of source pattern model elements and the corresponding source model element that each source pattern model element represents. The `Binding Specification` is created during the `Develop Binding Specification` activity.

A source model conforms to a source pattern. The notion of conformance used in this dissertation is that established by Kim et al. [19]. A source model conforms to a source pattern if the source model satisfies the structural and semantic properties defined in the source pattern. A source model element conforms to (or

plays the role of) a source pattern element if the source model element satisfies the properties defined in the source pattern element. A source model element that plays the role of a source pattern model element is said to be bound to the source pattern model element.

Source models are described using UML and source patterns are specified using a template version of the Role-Based Metamodeling Language(RBML) [7]. RBML Class diagram templates have template model elements that are explicitly marked using the "|" symbol.

4.2 Class Transformation Example

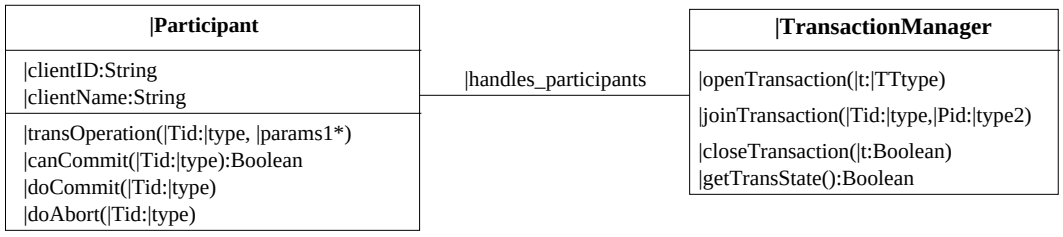

Figure 4.2: Source Pattern for Simple Transaction Service.

Figure 4.2 shows a source pattern for a simple transaction service. The |Participant class template provides service to clients through instances of its |transOperation operation template and initiates a transaction whenever an instance of this operation template is invoked. The |TransactionManager class template manages transactions.

|Participant provides the |canCommit, |doCommit and |doAbort operation templates to indicate: (1) its readiness to commit a transaction, (2) to commit a transaction and (3) to abort a transaction respectively. |TransactionManager provides the |openTransaction operation template to allow new transactions to be created and the |joinTransaction operation template to allow participating objects

to join a transaction. The |closeTransaction operation template is used to commit or abort a transaction depending on the value of its argument.

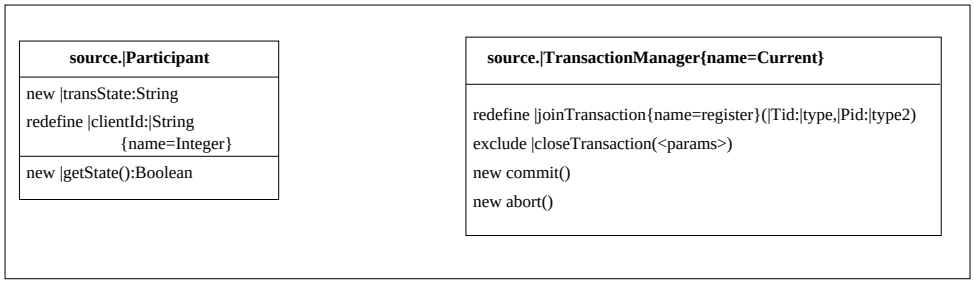

Figure 4.3: Transformation Schema.

Figure 4.3 shows a transformation schema for transforming source models that conform to this source pattern. The transformation schema has two `source` directives, an `exclude` directive, two `redefine` directives, four `new` directives and three `rename` directives.

This transformation schema may be applied to any source model that conforms to the pattern shown in Figure 4.2. An example of such a source model is presented in Figure 4.4 and the binding specification for this source model is presented in Table 4.1. Note that UserManagement and |UserRepository have no corresponding source pattern model elements. UserManagement and |UserRepository represent mechanisms for authenticating users of the transaction service.

The target model is obtained by processing the transformation schema. Whenever a directive in the transformation schema references a model element from the source pattern, the corresponding source model element listed in the binding specification is processed. For example, a reference to the `|Participant` class template in the transformation schema results in the AccountManager class being processed. The `source.|Participant` transformation schema class is processed as follows.

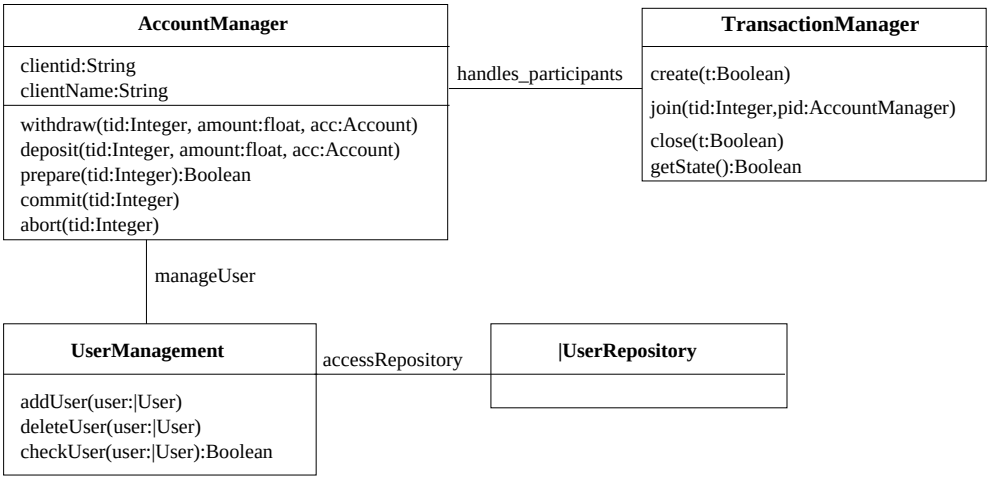

Figure 4.4: Source Model for Simple Transaction Service.

1. The `source.|Participant` directive is processed. The `source` directive is used to copy model elements from the source model to the target model. The `source.|Participant` directive results in the `AccountManager` class being copied to the target model.

2. The `new |transState:String` directive is processed. The `new` directive is used to create a new model element. The `new |transState:String` directive results in the attribute template: `|transState:String` being created and inserted into the `AccountManager` class in the target model.

3. The `new |getState():Boolean` directive results in the operation template: `|getState():Boolean` being created and inserted into the `AccountManager` class in the target model.

4. The directive, `redefine |clientID:String{name=Integer}` changes the type of the `|clientID:String` attribute template in the `AccountManager` class from String to Integer.

Figure 4.5 (a) shows the target model after all directives in the

Table 4.1: Binding Specification.

Source Pattern Element	**Source Model Element**
|**Participant**	AccountManager
|**Participant::clientID**	AccountManager::clientID
|**Participant::clientName**	AccountManager::clientName
|**Participant::**|**transOperation**	AccountManager::withdraw
|**Participant::**|**transOperation**	AccountManager::deposit
|**Participant::**|**canCommit**	AccountManager::prepare
|**Participant::**|**doCommit**	AccountManager::commit
|**Participant::**|**doAbort**	AccountManager::abort
|**Participant::**|**Tid**	AccountManager::tid
|**Participant::**|**type**	AccountManager::Integer
|**TransactionManager**	TransactionManager
|**TransactionManager:: openTransaction**	TransactionManager::create
|**TransactionManager:: joinTransaction**	TransactionManager::join
|**TransactionManager:: closeTransaction**	TransactionManager::close
|**TransactionManager::getTransState**	TransactionManager::getState
|**TransactionManager::**|**t**	TransactionManager::t
|**TransactionManager::**|**TType**	TransactionManager::Boolean
|**TransactionManager::**|**Tid:**|**type**	TransactionManager::tid:Integer
|**TransactionManager::**|**Pid::**|**type2**	TransactionManager:: pid:AccountManager
|**handles_participants**	handles_participants

`source.|Participant` transformation schema class are processed.

The `source.|TransactionManager{name=Current}` class template is processed as follows:

1. The `source.|TransactionManager` class is bound to the TransactionManager class. As a result, the `source.|TransactionManager{name=Current}` directive results in the `TransactionManager` class being copied to the target model and the name of the copied class template being changed to `Current`.

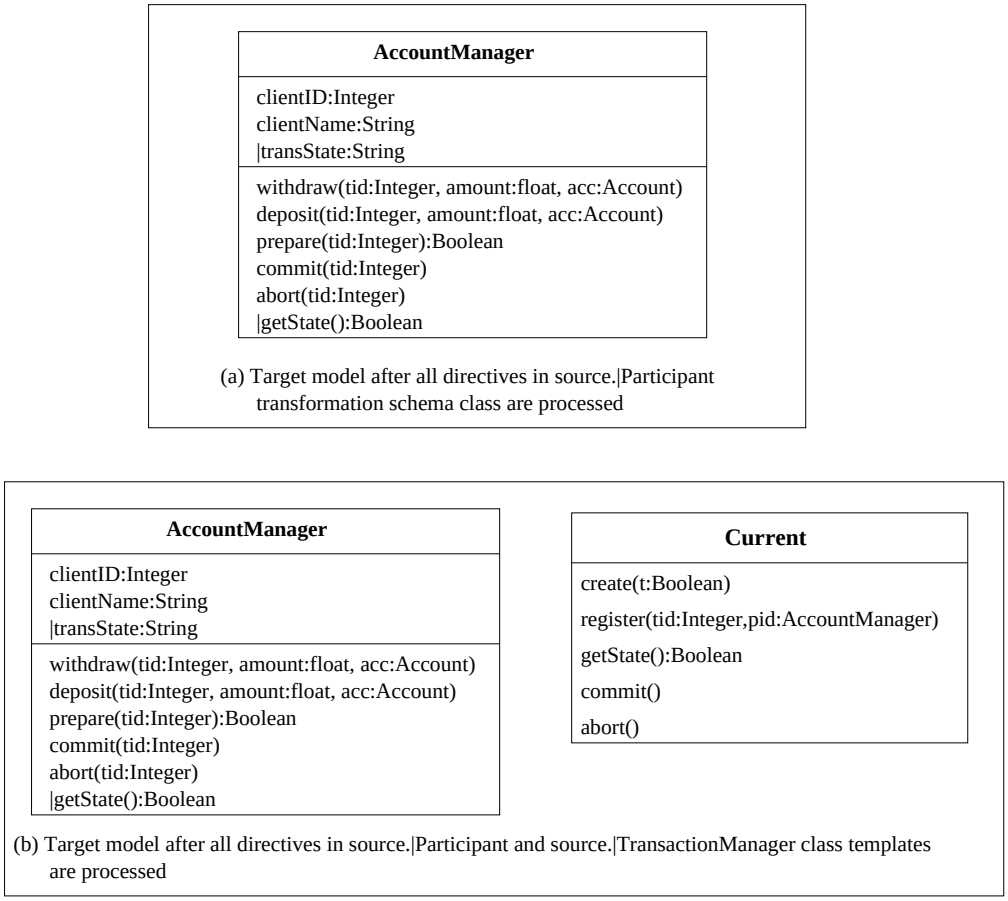

Figure 4.5: Target Model After Transaction Schema Classes are Processed.

2. The `redefine |joinTransaction{name=register}(|Tid:|type, |Pid:|type2)` directive results in the name of the `join` operation in the `Current` class being changed to `register`.

3. The `exclude |closeTransaction(<params>)` directive results in the close operation being removed fron `Current`.

4. The `new commit()` and `new abort()` directives results in two new operations: `commit()` and `abort()` being created and inserted into the `Current` class.

Figure 4.5 (b) shows the target model after all directives in the `source.|TransactionManager{name=Current}` transformation schema class are processed. At this point all the directives in the transformation schema have been processed. The other model elements not referenced by a transformation directive (i.e., `UserManagement, |UserRepository, handles_participant, manageUser, accessRepository`) are copied to the target model. The model elements not referenced by a transformation directive may be identified using the binding specification which does not list these items. The complete target model is shown in Figure 4.6.

In this example, the source directive is used to copy the AccountManager and TransactionManager classes to the target model. The source directive is used when modifications are to be made to a source model element. Each source model element may be transformed in one of three ways: (1) a source model element may be deleted using an exclude directive, (2) a source model element may be copied without modification, or (3) a source model element may be copied and modified using a source directive. When a source model element is to be copied without modification, the use of a transformation directive is not required since model elements that are not referenced in the transformation schema are automatically copied to the target model.

4.3 Form of Class Transformation Schemas

Model elements in a class transformation schema are either composite or non-composite. Transformation schema classes and transformation schema interfaces are composite elements while transformation schema operations, transformation schema attributes and transformation schema relationships are non-composite. A composite transformation schema model element is divided into compart-

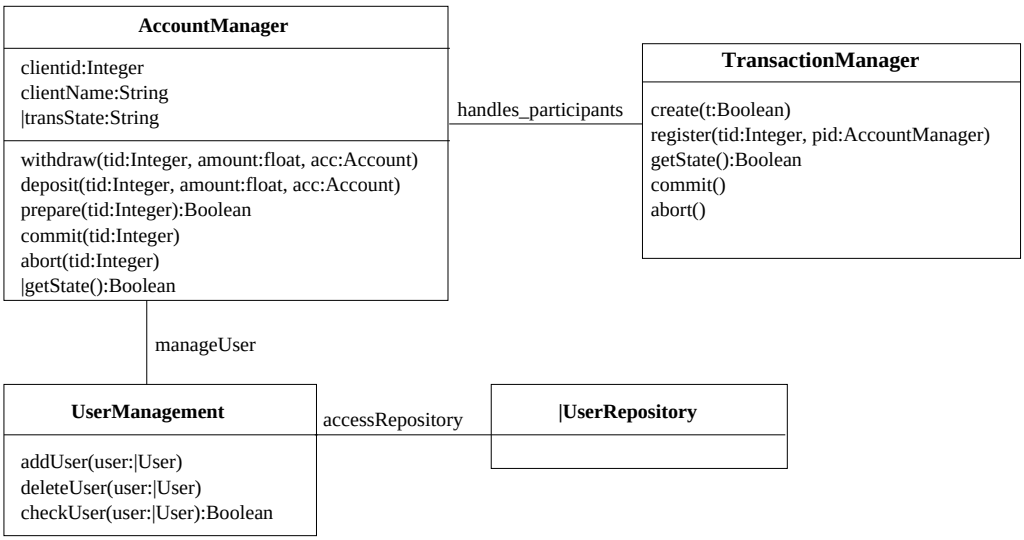

Figure 4.6: Target Model for Simple Transaction Service.

ments. A transformation schema class is divided into an `AttributeDirective Compartment`, an `OperationDirective Compartment` and a `NameDirective Compartment`. Transformation schema interfaces have `AttributeDirective Compartments` and `OperationDirective Compartments` only.

Except for exclude directives, each directive in a name directive compartment results in a model element name when processed. Except for exclude directives, each directive in an attribute directive compartment results in an attribute or attribute template when processed, and except for exclude directives, each directive in an operation directive compartment results in an operation or operation template when processed. Figure 4.7 (a) gives a pictorial view of transformation schema compartments, and Figure 4.7 (b) shows a transformation schema class diagram. Transformation schema compartments may contain directives as follows:

1. A name directive compartment may contain source directives, rename directives, an exclude directive or a new directive.

2. Attribute directive compartments and operation directive compartment may

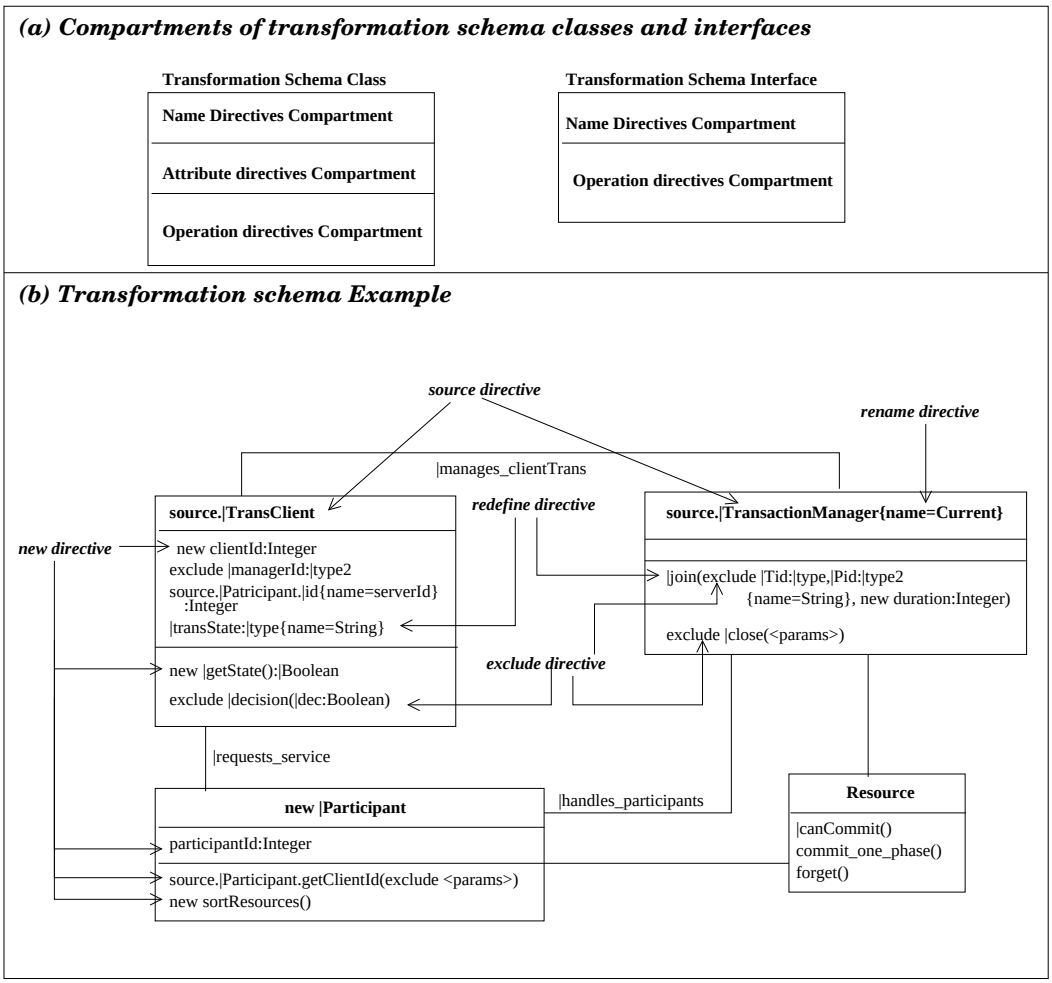

Figure 4.7: Transformation Schema Compartments.

contain any directive: `source, redefine, new, exclude` or `rename`.

Transformation schema relationships may have source, new or exclude directives. Figure 4.8 shows the model elements used to describe class transformation schemas. Each transformation schema model element is a stereotype of a UML metaclass. For example, a transformation schema class is a stereotype of a UML class, a transformation schema interface is a stereotype of a UML interface, a transformation schema operation is a stereotype of a UML operation and a transformation schema attribute is a stereotype of a structural feature.

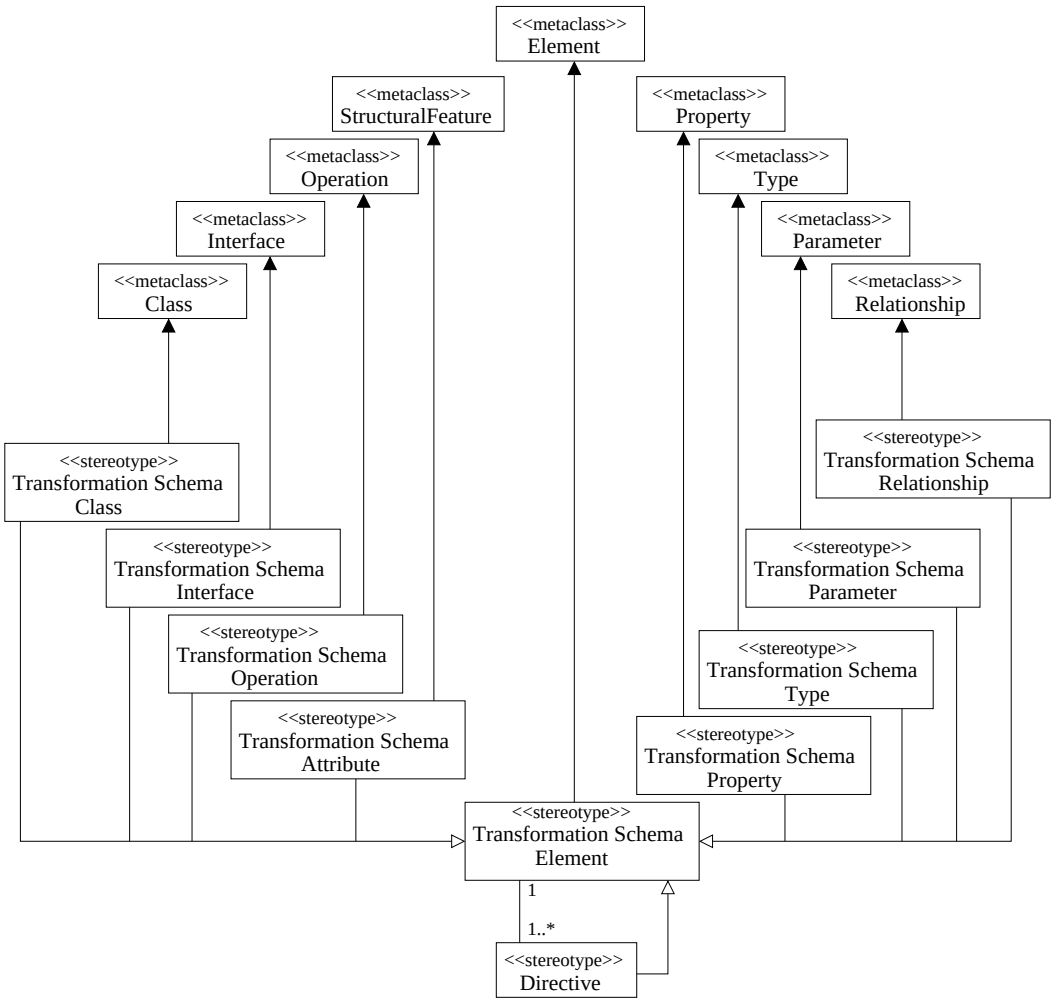

Figure 4.8: Transformation Schema Stereotypes.

The directives in name-directive compartments, attribute-directive compartments and operation-directive compartments are processed in order from top to bottom. Different ordering of directives may result in different target models. A name-directive compartment may contain multiple source directives. The semantics of these directives is that the source model class or interface referenced by the first source directive is copied, after which the attributes and operations in the classes or interfaces referenced by the other source directives are merged into the copied class or interface in any arbitrary order. The merge operation preserves

the integrity of the target namespace. The order in which transformation schema classes, interfaces and relationships are processed is determined manually.

4.4 Transformation Directives

The transformation directives that can be used in a class transformation schema are: `source, rename, exclude, redefine` and `new`. Each directive is described using the following format [30]:

- `Directive Name`: this section states the name of the directive.
- `Purpose`: this section describes the purpose of the directive.
- `Form`: this section describes the syntactic form of the directive and the model elements that the directive operates on. Items in square brackets, '[]', are optional and keywords are italicized.
- `Constraint`: this section gives the conditions that must hold if the directive is to have the intended effect.
- `Effect`: this section describes the effect of the directives on the target model.

Transformation directives are described in the subsections that follow. In order to simplify the presentation of directives, source model elements are given the same name as the source pattern elements to which they are bound with the exception of the '|'. For example, a |`ServiceParticipant` class template in the source pattern is bound to a `ServiceParticipant` class in the source model.

4.4.1 The `rename` Directive

Directive Name: `rename`

Purpose: The `rename` directive is used to provide a platform-specific name for a model element.

Form: The `rename` directive has the form: `ModelElement {`*name*`=modelElementName}`.

`ModelElement` is a reference to a model element in the source pattern and `modelElementName` is the platform-specific name to be given to the source model element bound to `ModelElement`. For example, middleware protocols typically define specific names for their model elements. The Jini transaction service, for example, specifies a `TransactionManager` interface while the CORBA transaction service specifies `Current, Control, Coordinator` and `Terminator` interfaces. The name directive may be used to associate these platform-specific names with platform-independent model elements.

Constraint: The model element referenced by `ModelElement` must exist.

Effect: The name of the model element bound to `ModelElement` has been changed to `modelElementName`.

Examples: Examples of the `rename` directive will be given in the sections that follow.

4.4.2 The `source` Directive

Directive Name: `source`

Purpose: The `source` directive is used to copy a source model element or meta-attribute to the target model. When a model element is copied, any

constraint associated with the model element is also copied.

Form: The source directive has the following forms.

1. *source*`.Parent[RenameDirective]`. `Parent` is a reference to a composite source pattern model element and `RenameDirective` is an optional `rename` directive. The source directive stipulates that the model element bound to `Parent` should be copied to the target model. All properties of the model element bound to `ModelElement` are copied.

2. *source*`.Parent.ModelElement[RenameDirective]`. `Parent` is a reference to a composite source pattern model element and `ModelElement` is a sub-element of `Parent`. `ModelElement` must be bound to an operation, operation template, attribute or attribute template, in the model element bound to `Parent`.

   When an operation (or operation template) is copied into a different namespace from the one in which it was defined, the developer must ensure that all constraints associated with the operation hold, and that all references to the operation are valid. The same is true of attributes, attribute templates, and other source model elements such as class templates, that are copied and modified.

3. *source*`.ModelElement.`*Property*`[.SubElement].MetaAttribute`.
   `SubElement` is a reference to an optional subelement of `ModelElement`. A sub-element of a model element is an element defined in the model element, for example, an operation defined in a class. `MetaAttribute` is a meta-attribute of `ModelElement` or `SubElement`.

Constraint: The model element bound to `ModelElement` must exist. When

`Parent` and `SubElement` occur in the source directive, the model elements bound to `Parent` and `SubElement` must exist.

Effect: For the first forms of the directive, a copy of the model element bound to `Parent` is present in the target model. If a `rename` directive is present, the name of the copied model element is the platform-specific name specified in the `rename` directive, otherwise, the name of the newly created model element remains the name of the model element bound to `ModelElement`. For the second form of the directive, a copy of the model element bound to `ModelElement` is present in the target model. For the third form of the directive, the specified the meta-attribute value has been copied to the target model.

Examples: Some examples of `source` directive are given in section 4.2.

4.4.2.1 Merging Classes Using the Source Directive

The `source` directive may be used to merge multiple classes or multiple interfaces. For example, Figure 4.9 shows a source pattern with two class templates and a transformation schema with a single transformation schema class specified using two source directives: `source.|AccountParticipant` and `source.|ServiceParticipant`. The effect of the directives is to merge that attributes and operations of the class to which `|ServiceParticipant` is bound into a copy of the class to which `|AccountParticipant` is bound.

The name of the new merged model element in the target model is the name associated with the source model element that is bound to the first source directive. As a result the name of the class in the target model in Figure 4.9 is: `AccountParticipant`. Merging preserves the integrity of the namespace by ensuring that an operation or attribute is only added to the class or interface if a model element with that signature is not already present.

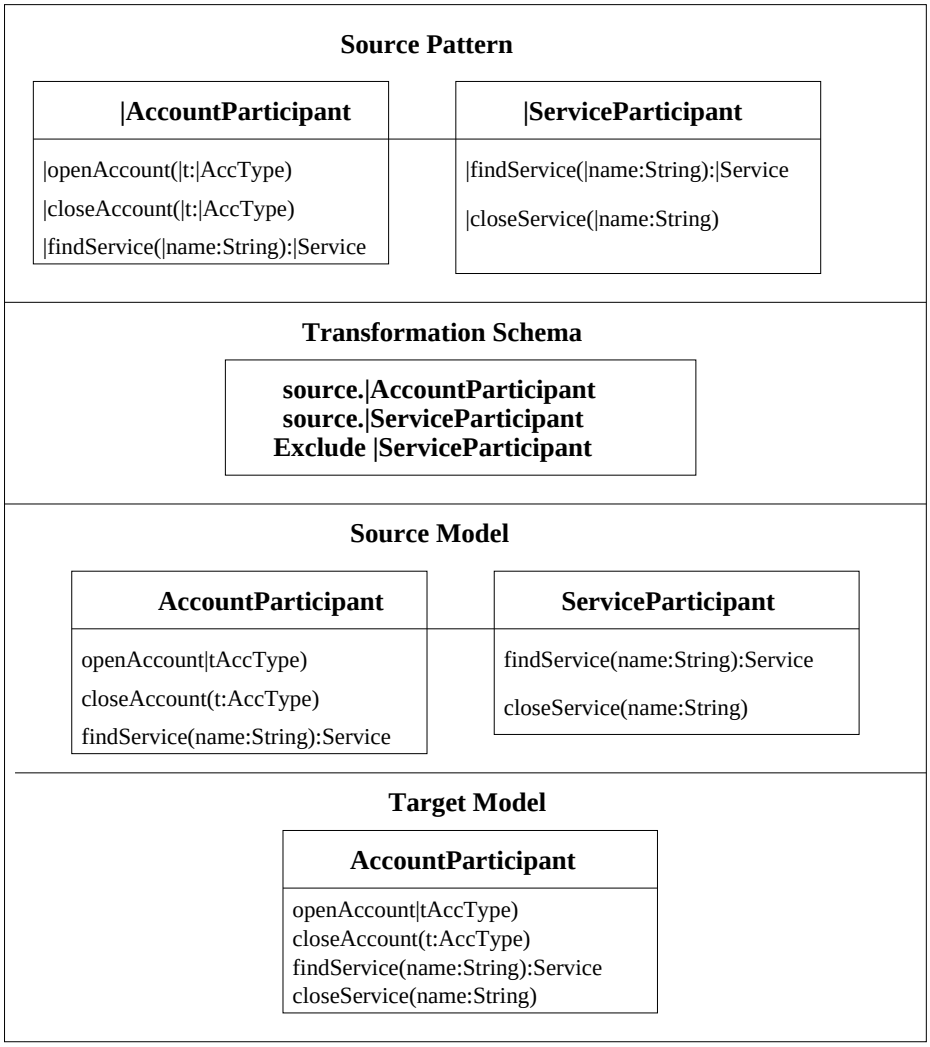

Figure 4.9: Merging Model Elements Using The `source` Directive.

The effect of the two source directives is illustrated in the target model in the figure. The target model has four operations: three from the AccountParticipant class and the `closeService(name:String)` operation from the ServiceManager class. The `findService` operation in the ServiceManager class has the same signature as the `findService` operation in the AcountManager so only one copy of this operation is added.

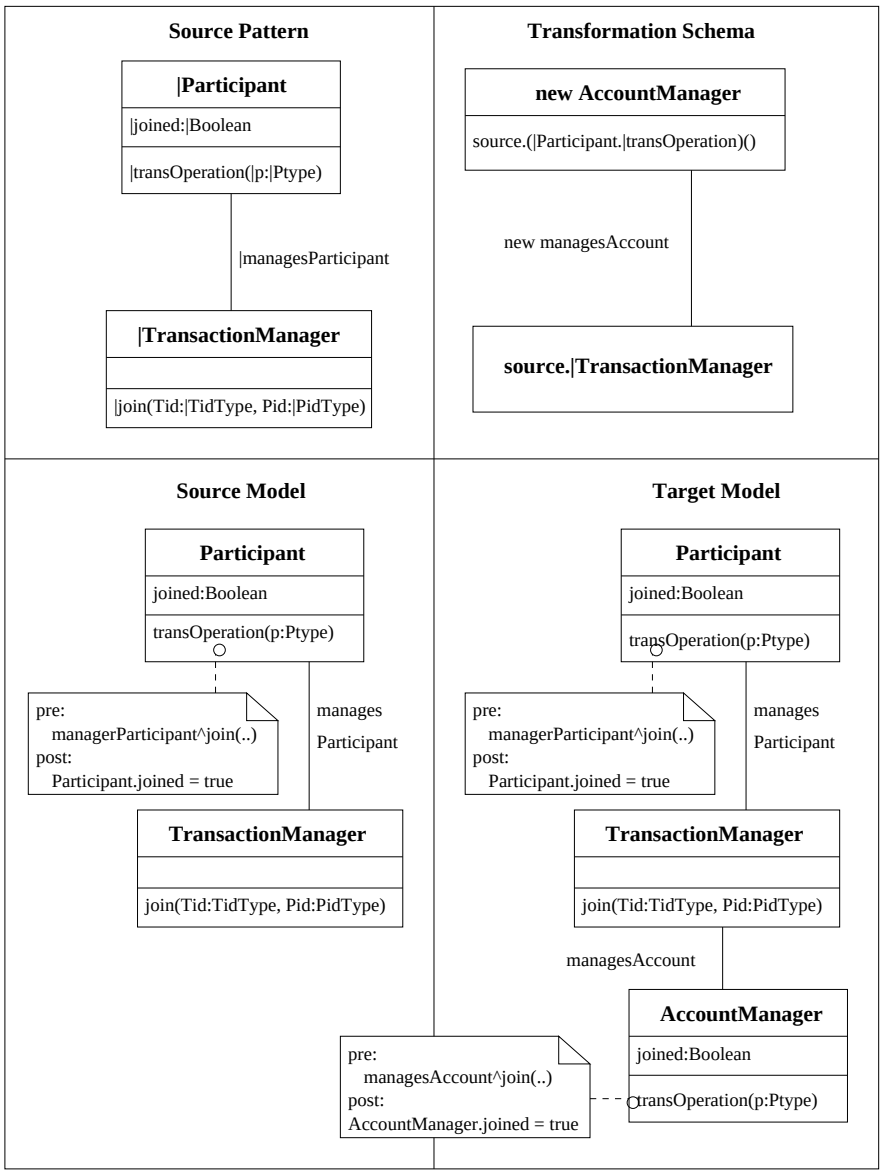

Figure 4.10: The *source* Directive Applied to Operation Templates.

4.4.2.2 Operation Template `source` Directive

Figure 4.10 provides an example of an operation template source directive. The figure illustrates the use of the source directive to copy an operation to a target class. In the figure, the `new AccountManager` transformation schema class has the directive: `source.|Participant.|transOperation`. The

`new AccountManager` directive stipulates the creation of a new class. The `source.|Participant.|transOperation` directive stipulates that an operation from the source model should be copied to the new class. The operation to be copied is the one bound to the `|Participant.|transOperation` operation template. In the example, `Participant` class from the source model is bound to the `|Participant` class template, `TransactionManager` is bound to `|TransactionManager`, `|joined:|Boolean` is bound to `joined:Boolean` and so on.

The effect of the source directive is illustrated in the target model which has a new AccountManager class. The target model shows that the `transOperation` operation, including its pre and post conditions were copied from the Participant class to the AccountManager class. The pre and post conditions were updated to reflect the context of the AccountManager class. Specifically, `managesParticipant` was changed to `managesAccount` and `Participant.joined` was changed to `AccountManager.joined`.

A metaattribute may be copied using the second form of the source directive. For example, the `isAbstract` property of the class to which the |Participant class template is bound may be copied to a target model class or class template using the directive: `source.|Participant.property.isAbstract`.

4.4.3 The `redefine` Directive

`Directive Name: redefine`

Purpose: The `redefine` directive is used to modify a model element that is copied to the target model using the `source` directive.

Form: The redefine directive has two forms.

1. *redefine* `TSModelElement` where `TSModelElement` is a reference to a source pattern operation or attribute. When `TSModelElement` is an operation, it must have an associated exclude, new or rename directive. The associated rename directive may be applied to a transformation schema parameter of the transformation schema operation. The associated rename directive may also be applied to the name of the transformation schema operation. The associated new or exclude directive may be applied to a transformation schema parameter of the transformation schema operation. When `TSModelElement` is a transformation schema attribute, it must have an associated rename directive.

   When an operation (or operation template) is redefined, the developer must ensure that all constraints associated with the operation hold, and that all references to the operation are valid. The same is true of an attribute or attribute template.

2. *redefine*`[ModelElement.]`*Property*`[.SubElement].MetaAttribute=val`, where `val` is the value to be given to the meta-attribute. This form of the redefine directive is used to change the value of a meta-attribute.

Constraint: The model element bound to `TSModelElement` must exist. When `Parent` and `SubElement` occur in the redefine directive, the model elements bound to `Parent` and `SubElement` must exist. The types of `MetaAttribute` and `newValue` must be compatible. A `redefine` directive may only be specified in a composite transformation schema model element specified using a source directive.

Effect: For the first form of the directive, the model element bound to `TSModelElement` has been modified according to the associated exclude, new or rename directives used. For example, if a rename directive is applied to the name

of a transformation schema operation, then the name of the model element referenced by `TSModelElement` is changed to the platform-specific name specified in the rename directive. If an exclude directive is applied to the name of a transformation schema parameter, then the transformation schema parameter is eliminated from the model element bound to `TSModelElement` .

For the second form of the directive, the meta-attribute in the target model has the value `newValue`.

Example:

In Figure 4.11, the `Participant` class from the source model is bound to the `|Participant` class template. The `redefine |committed:Boolean{name=Integer}` transformation schema attribute redefines the `committed:Boolean` attribute defined in `Participant` by changing its type to Integer. Similarly, the `redefine |canCommit{name=prepare}(|Pid:|PidType) :|commitVal{name=Vote}` transformation schema operation redefines the `canCommit` operation defined in `Participant` by specifying platform-specific names for the operation name and the return type. The target model shows the effect of the two redefine directives.

A metaattribute may be modified using the second form of the redefine directive. For example, in Figure 4.11, the `isAbstract` property of the Participant class in the target model may be changed by specifying the directive: `redefine property.isAbstract = false` or `redefine property.isAbstract = true` in the transformation schema.

4.4.4 The `new` Directive

Directive Name: `new`

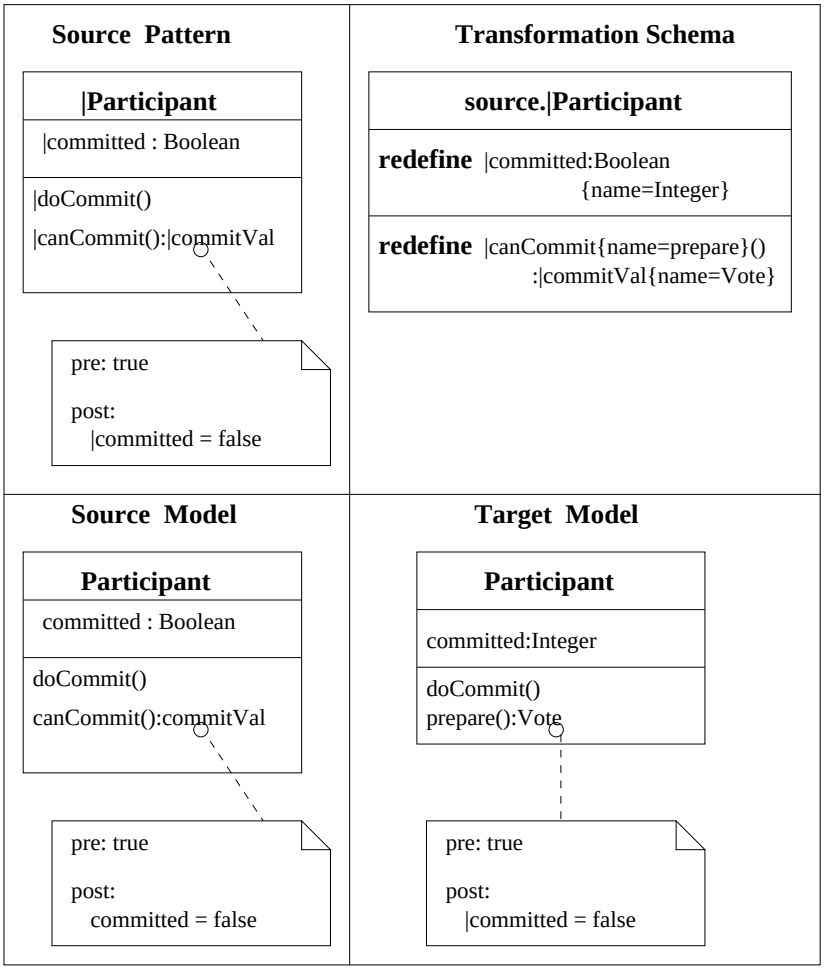

Figure 4.11: The `redefine` Directive.

Purpose: The `new` directive is used to specify a new model element or a value for a metaattribute.

Form: The `new` directive has three forms.

1. `[`*new*`] TSModelElement` where `TSModelElement` is the specification of a new class, class template, interface, interface template, operation, operation template, attribute, attribute template, parameter, parameter template, relationship or relationship template.

2. *new*. In this form, the directive is attached to a transformation schema association or transformation schema relationship that is to be created.

3. [*new*] *Property*.`metaAttribute = newValue`.

Constraint: A model element with the signature of `TSModelElement` must not exist in the target namespace.

Effect: For the first form of the directive, a new model element with the signature of `TSModelElement` is present in the target namespace. For the second form of the directive, a new model element corresponding with that associated with the directive is present in the target namespace. For the third form of the directive, the specified meta-attribute of the model element in the target model has the value: `newValue`.

Example:

Figure 4.12 shows a source model with the |`Participant` class template. This class template defines one attribute template and two operation templates. The figure also shows a transformation schema with the following features:

1. The directive, `new participantId:Integer` defines a new attribute.

2. The directive, `new getStatus():String`, defines a new operation.

3. The directive, `new quickSort(records:String[])`, defines a new operation.

The effect of the three new directives can be seen in the target model which shows that a new attribute and two new operations are added to the Participant class from the source model.

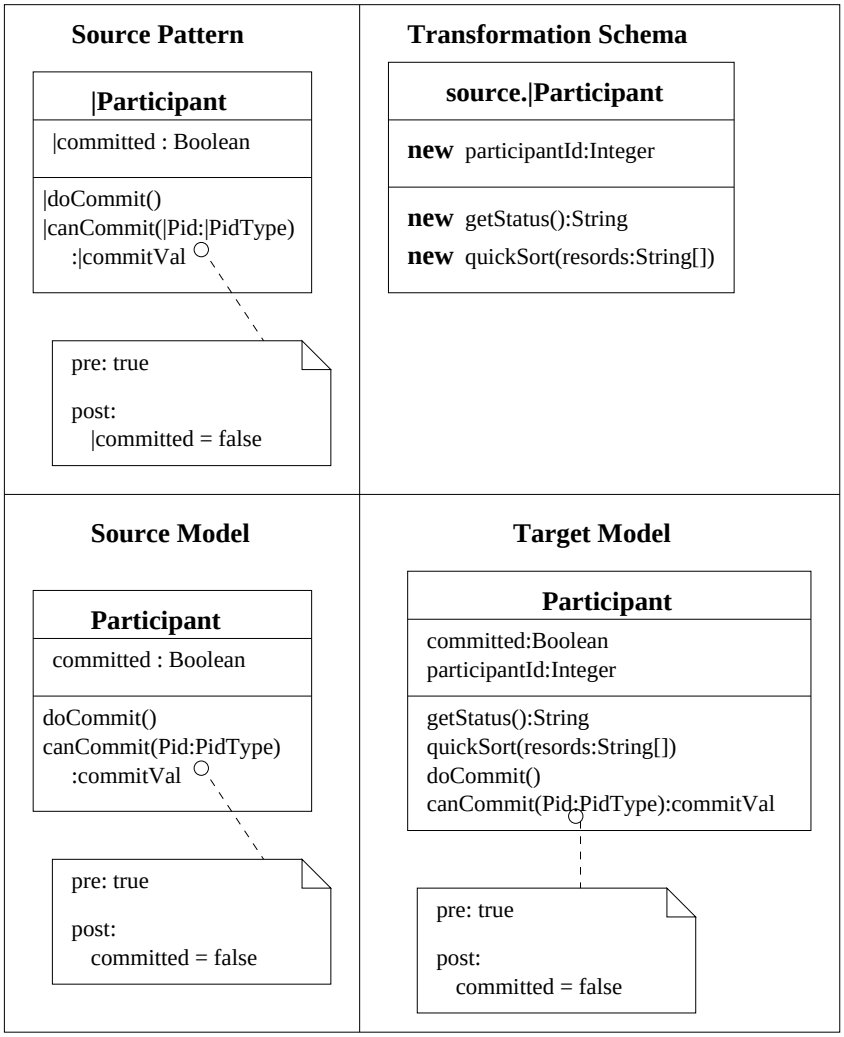

Figure 4.12: The `new` Directive.

A transformation schema model element that is specified without using a leading directive keyword defaults to '`new`' as the leading directive keyword. A leading directive keyword is the first word in a directive. `Source, exclude, new` and `redefine` are leading directive keywords. The specification of a model element without using a leading directive keyword is referred to as the implicit form of the new directive. For example, in Figure 4.13 `Current` is a new class that may be specified as: `new Current`. Similarly, the `|joined:Boolean` transformation

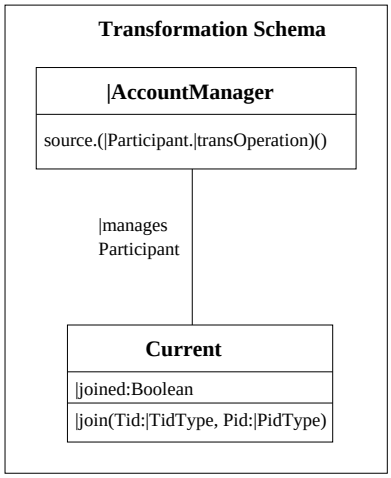

Figure 4.13: Implicit Use of The `new` Directive.

schema attribute template, the `|join(Tid:|TidType, Pid:|PidType)` transformation schema operation template and the `|managesParticipant` transformation schema association template are new model elements.

4.4.5 The `exclude` Directive

Directive Name: `exclude`

Purpose: The `exclude` directive is used to exclude source model elements from inclusion in the target model.

Form: The `exclude` directive has the two forms: (1) *exclude* `ModelElement` and (2) `exclude`. In the first form, `ModelElement` is a reference to a source pattern model element. In the second form, the directive is attached to a transformation schema association or transformation schema relationship that is to be excluded from the target model. The `exclude` directive may be applied to any model element.

Constraint: The model element bound to `ModelElement` must exist in the source model. A model element can only be omitted from a namespace if the model element is visible within that namespace.

Effect: For the first form of the directive, a copy of the model element bound to `ModelElement` is not present in the target model. For the first form of the directive, the source model element corresponding to the transformation schema model element to which the directive is attached, is not present in the target model.

Example:

Figure 4.14 shows a source model with two classes: `TransactionManager` and `Participant`. TransactionManager contains four operations, two of which (`timeOut` and `initiateVotingPhase`) have no CORBA equivalents. The figure also shows a transformation schema defined for CORBA. The transformation schema is specified as a class template using a source directive. Based on this source directive, all four operation templates are copied to the target model including the two operations that are undefined in CORBA. These two operations are eliminated from inclusion in the target model using two exclude directives: `exclude |timeOut(|Tid:|TidType)` and `exclude |initiateVotingPhase(|Tid:|TidType`. The other exclude directive in the figure: `|join(exclude |Tid:|TidType)` results in this parameter template being eliminated from the operation. The effect of the directives can be seen in the target model.

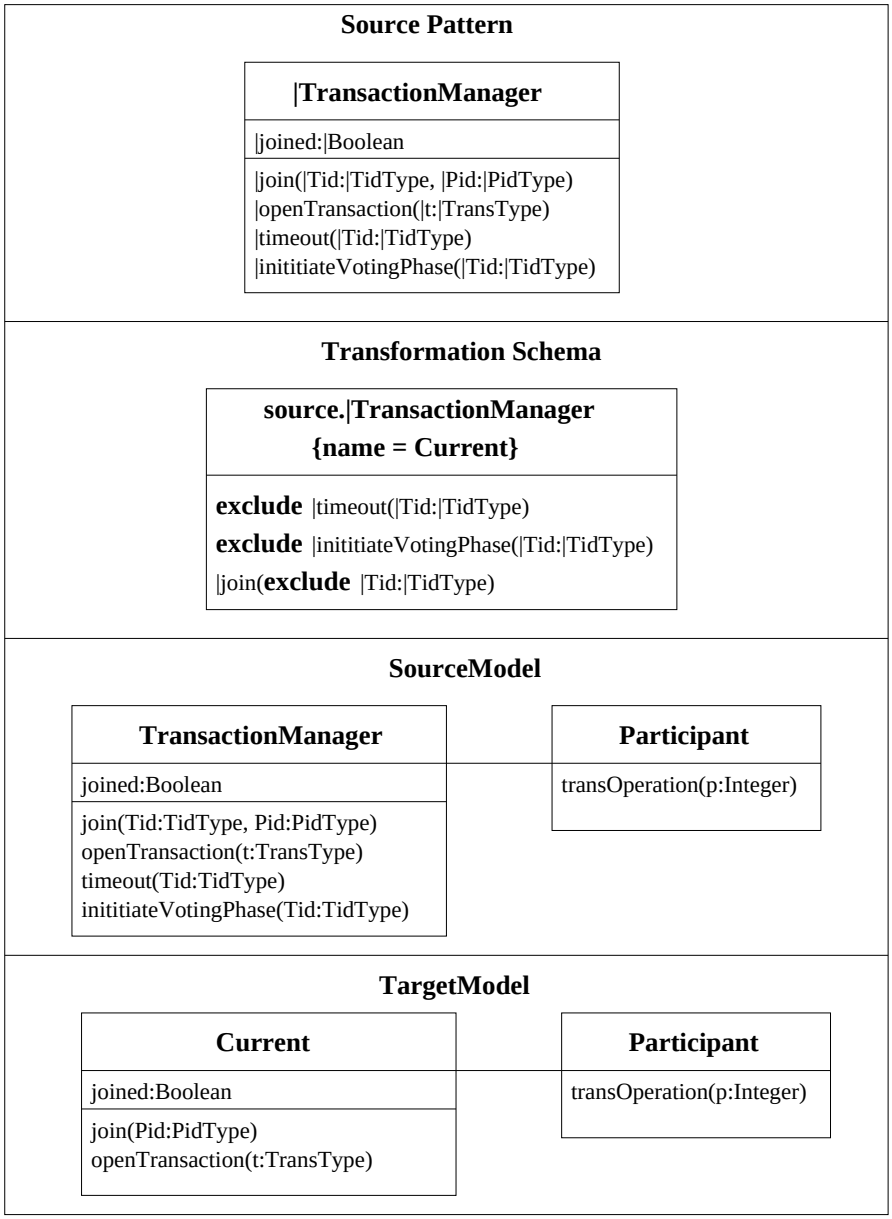

Figure 4.14: The `exclude` Directive.

4.4.6 Applying Directives to Relationships

The source, new and exclude directives may be applied to relationships. Figure 4.15 illustrates:

- The application of the source directive to the |`vehicleDetails` transforma-

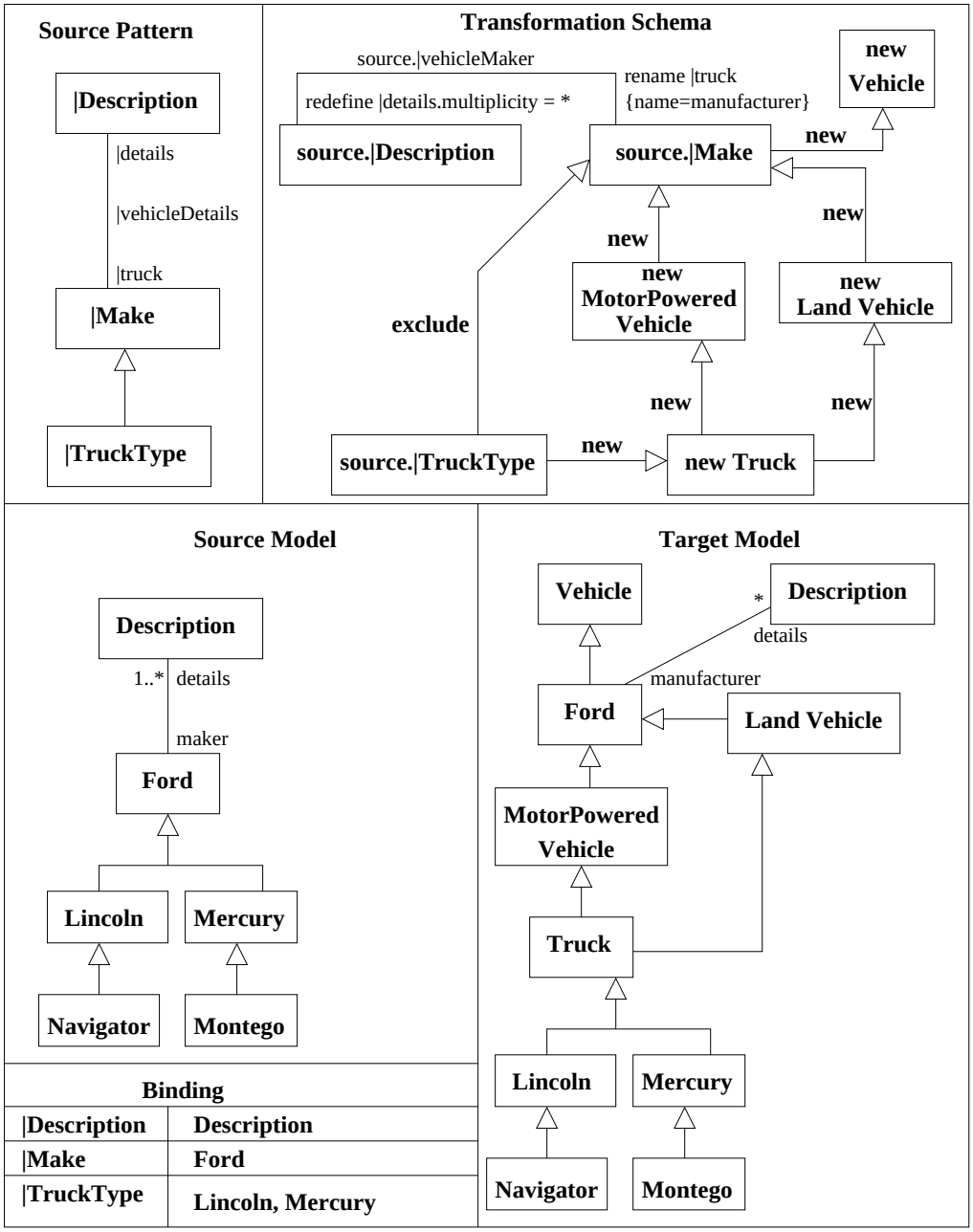

Figure 4.15: Applying Directives to UML Relationships.

tion schema association.

- The application of the redefine directive to the |`details` and |`truck` transformation schema association ends.

- The application of the exclude directive to the transformation schema generalization between `source.|TruckType` and `source.|Make`.

- Six applications of the new directive to transformation schema generalizations.

A directive may be attached directly to an association end, or an association end may be accessed using the syntax, `associationName.AssociationEndName` or `AssociationEndName`. `AssociationName` is the name of an association and `AssociationEndName` is the name of one of its association ends.

The multiplicity at an association end is accessed using the syntax, `associationName.AssociationEndName.multiplicity`, `AssociationEndName.multiplicity` or `multiplicity`. For example, in Figure 4.15, the directive `redefine |details.multiplicity = *`, results in the multiplicity of the corresponding association end in the target model being changed to `*`.

4.5 Class Diagram Transformation Metamodels

The relationships among transformation concepts participating in the model-to-model transformation of class models are illustrated in Figure 4.16.

The `Source Metamodel` and the `Target Metamodel` is the class template metamodel presented in Figure 2.3 in Section 2.3. The `Source Pattern` describe a subset of instances of the `Source Metamodel`. The source pattern describes the properties expected of each valid source model. A source model element is transformed into a target model element using the directives associated with the source pattern element to which the source model element is bound. In effect, source models may have model elements not described by the source pattern.

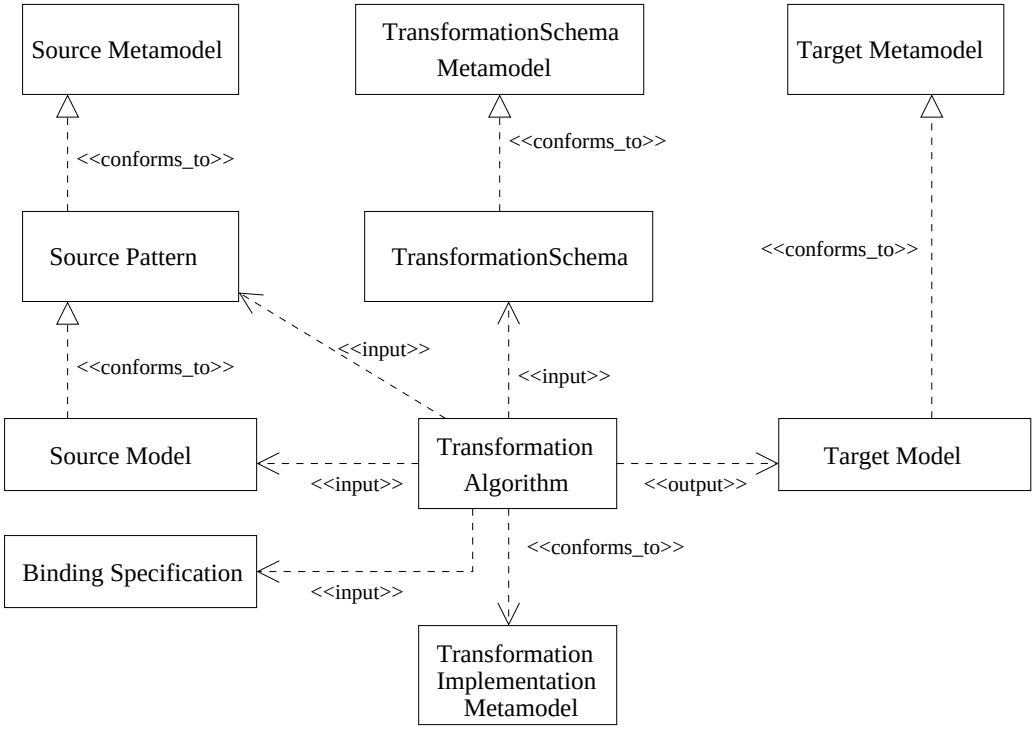

Figure 4.16: Model-to-model Transformation Conceptual Model.

These model elements that are not described by any model element in the source pattern are retained in the target model.

A transformation is effected by executing the `Transformation Algorithm`. The input to the algorithm are the source model, the source pattern, the binding specification and the transformation schema. The algorithm outputs the target model. The `Transformation Algorithm` is an instance of the `Transformation Implementation Metamodel`.

A model transformation maps source model elements to target model elements. The `Transformation Implementation Metamodel` describes these mappings by specifying relationships between: (1) source metamodel elements and transformation schema metamodel elements, and (2) transformation schema metamodel elements and target metamodel elements.

The transformation schema metamodel and the transformation implementation metamodel are described in the subsections that follow. An object diagram of a transformation schema is also presented to illustrate the conformance of transformation schemas to the transformation schema metamodel.

4.5.1 Transformation Schema Class Diagram Metamodel

The relationships between transformation schema directives and other transformation schema model elements are illustrated in the transformation schema class diagram metamodel shown in Figure 4.17. A `Directive` is a `ComplexDirective` or a `Rename` directive. A `ComplexDirective` is an `Exclude` directive or a `CreateDirective`. `Source`, `Redefine` and `New` are `CreateDirective`s. A `CreateDirective` may contain other subdirectives where the first directive is the primary directive and the subdirectives are secondary directives.

A transformation schema relationship may have a source, new, or exclude directive. Transformation schema attributes, transformation schema operations, transformation schema types and transformation schema parameters must have an associated complex directive.

4.5.2 Transformation Schema Object Diagram

This section presents an object diagram of a transformation schema based on the transformation schema class diagram metamodel specified in Figure 4.17. Figure 4.18(a) shows a transformation schema with a single transformation schema class. The directives in the transformation schema class are numbered from 1 to 12. Figure 4.18(b) shows an object diagram of the transformation schema class that conforms to the metamodel shown in Figure 4.17. The original transformation schema class may be recreated from the object diagram using the following algorithm:

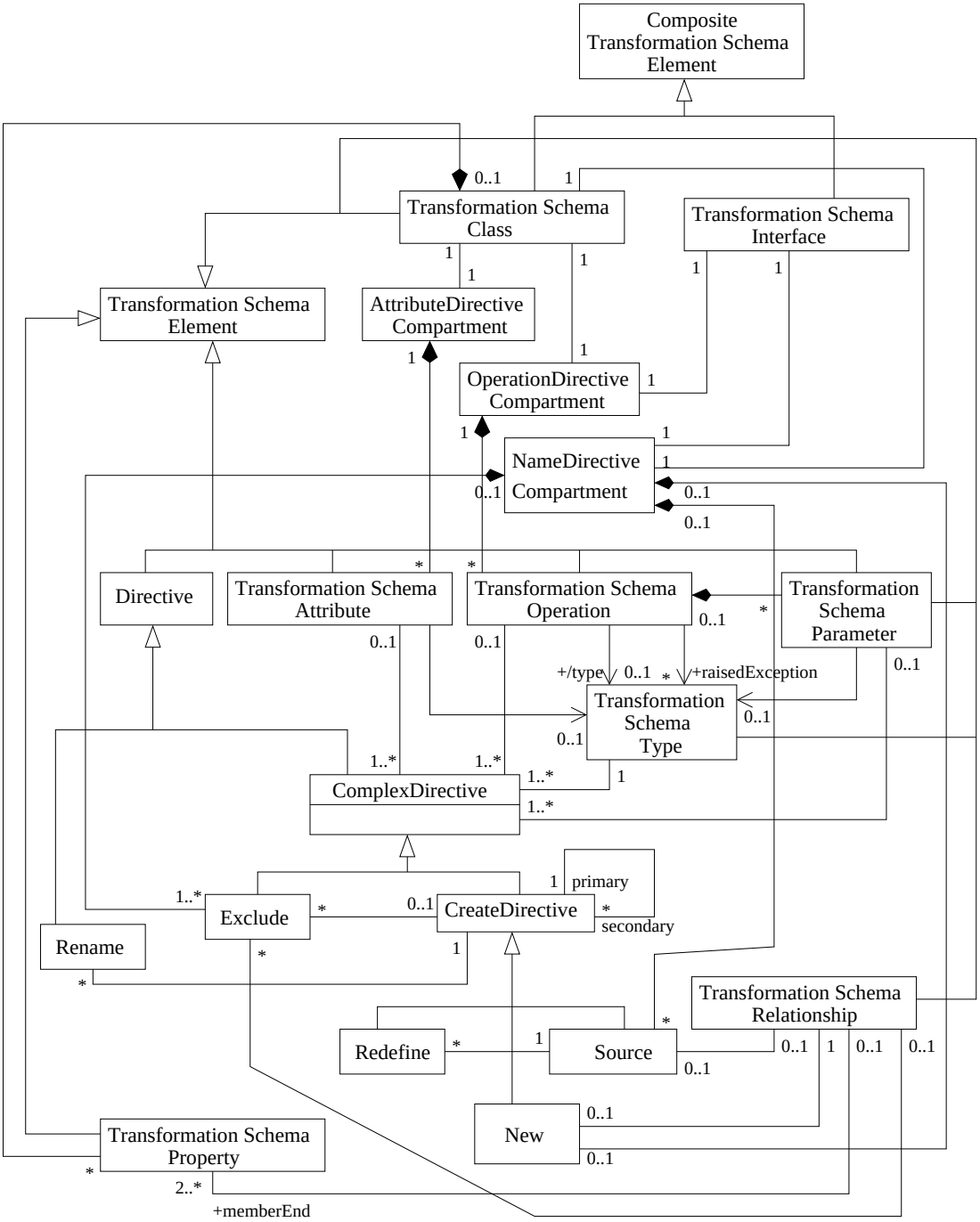

Figure 4.17: Transformation Schema Class Diagram Metamodel.

1. Create a new transformation schema class.

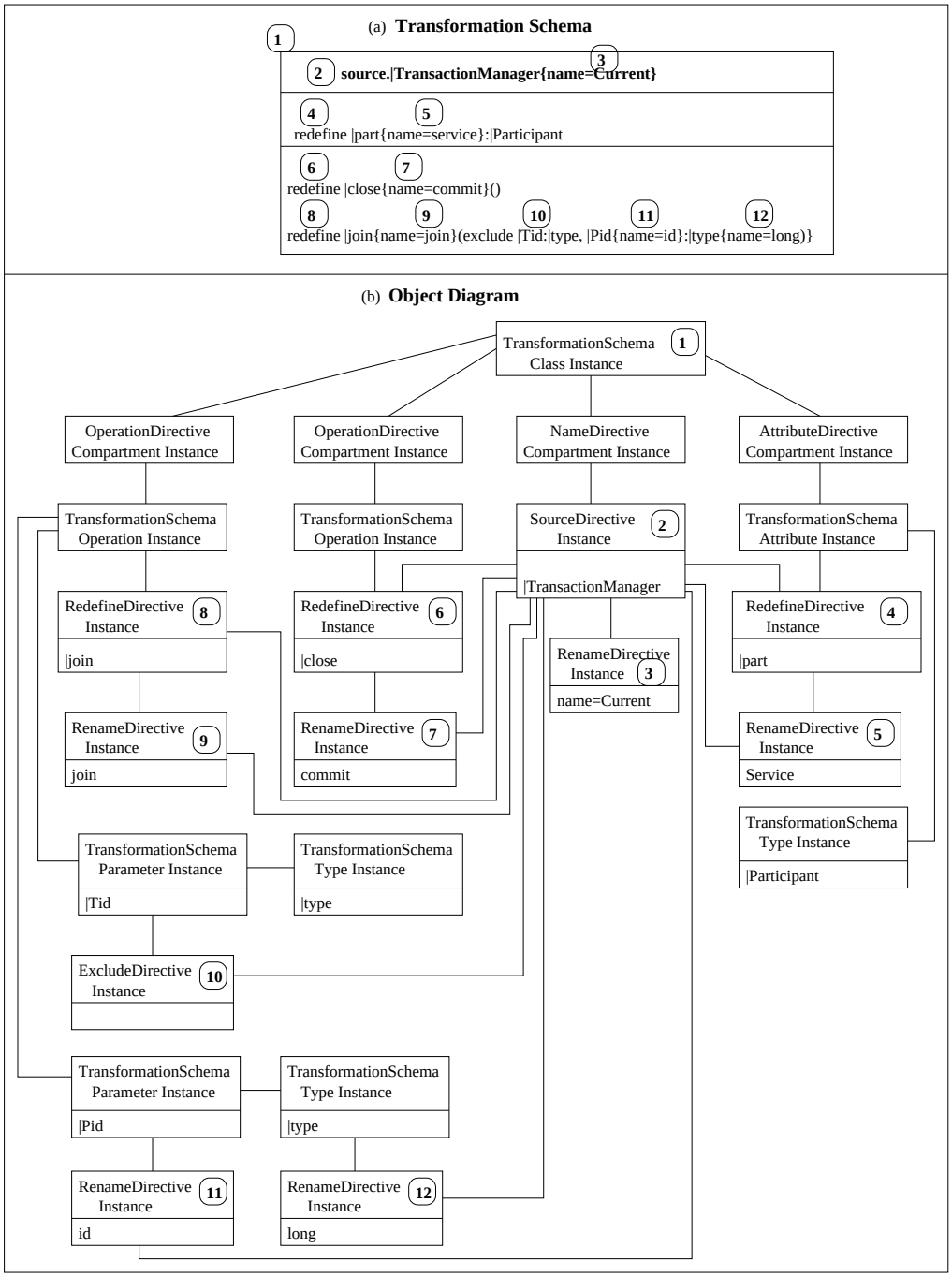

Figure 4.18: Transformation Schema Object Diagram.

2. Obtain the directive for the name directive compartment from the objects connected to the name directive compartment instance.

3. Obtain the directive for the attribute directive compartment from the objects connected to the attribute directive compartment instance.

4. Obtain the directives for the operation directive compartment from each section the object diagram that begins with an operation directive compartment instance.

Ambiguities can be resolved my marking each element in the object diagram once the element has been visited and noting that a rename directive instance associated with a source directive in the name directive compartment, will always have a single link to the source directive instance.

4.5.3 Transformation Implementation Metamodel

The transformation implementation metamodel is shown in Figure 4.19 and Figure 4.20. The first figure illustrates two sets of relationships. The `+canBeAppliedTo` association describes relationships between model elements of the source metamodel, and model elements of the transformation schema class diagram metamodel. A Transformation schema interface, for example, may be applied to one or more interface templates. An association end template labeled `+source` is connected to a model element from the source metamodel.

The `+canBeDerivedFrom` association describes the relationship between model elements of the transformation schema class diagram metamodel and model elements of the target metamodel. An interface template, for example, may be derived from a transformation schema interface, by processing the directives in the transformation schema interface. The `+canBeDerivedFrom` is a one-to-one relationship. An association end template labeled `target` is connected to a model element from the target metamodel.

Figure 4.20 augments the metamodel shown in Figure 4.19 by adding new

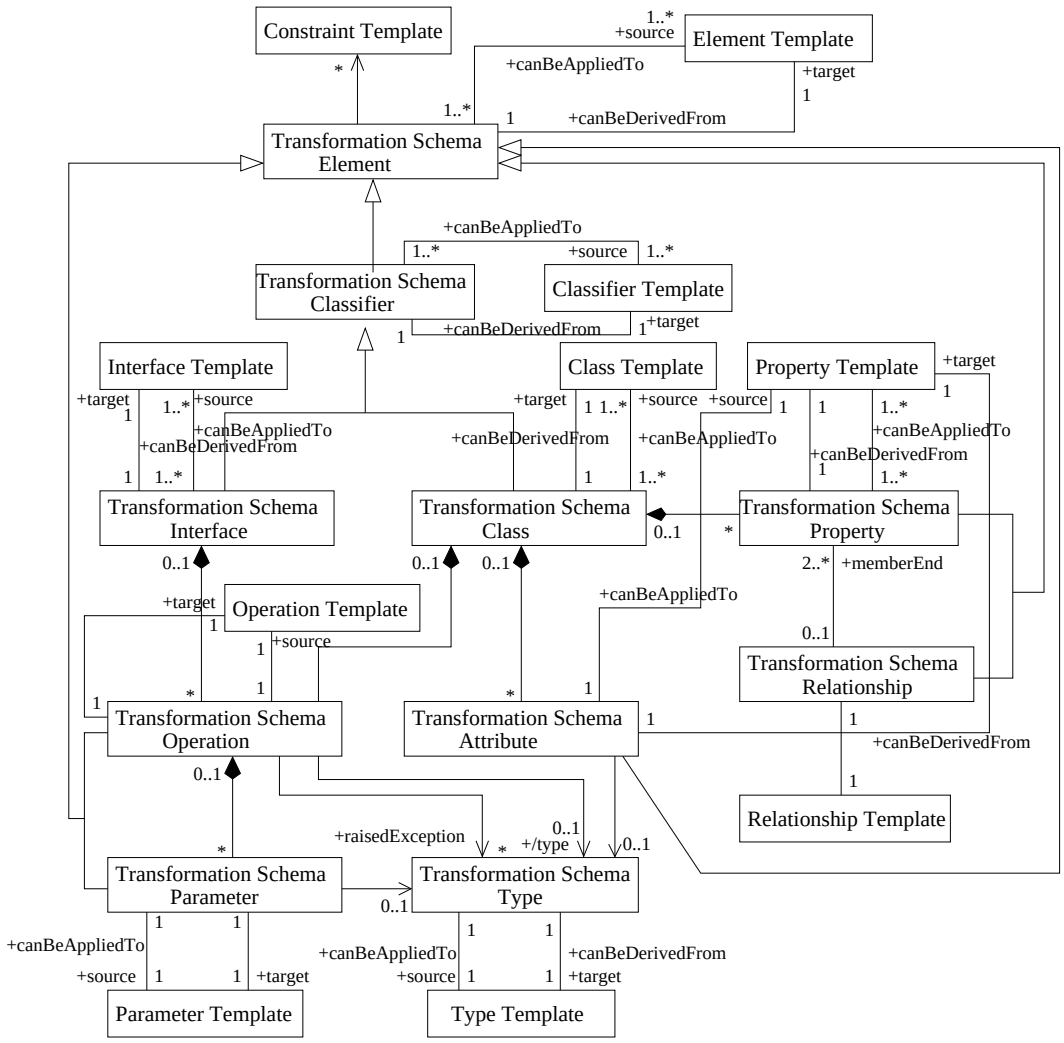

Figure 4.19: Transformation Implementation Metamodel Showing Relationships.

metaclasses and behavioral features to support the transformation of class template models. Figure 4.20 only shows those model elements of Figure 4.19 to which behavioral features are added.

4.6 A Grammar for Transformation Directives

Figures 4.21 and 4.22 show a grammar for class transformation schema directives written in ExtendedBackus-Naur Form (EBNF). The grammar uses the following

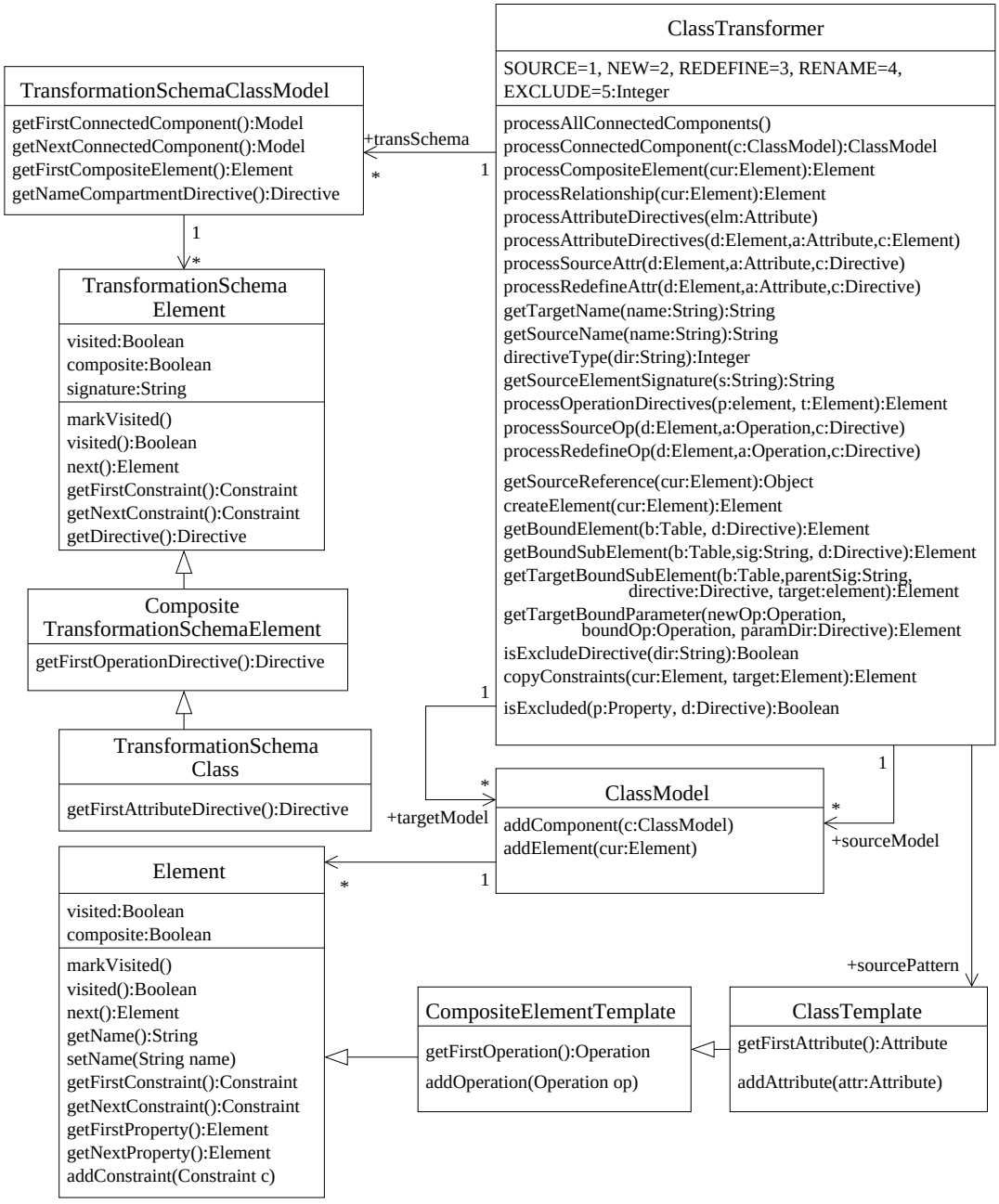

Figure 4.20: Transformation Implementation Metamodel Showing Behavioral Features.

syntax rules:

- Optional items are enclosed in: '[', ']' pairs.

```
Directives := SOURCE | RENAME | NEW | EXCLUDE | REDEFINE

SOURCE := ClassNameSource | AttrSource | OpSource | RelationshipSource
        | PropertySource
ClassNameSource := "source." elementName [classNameExpr] [RENAME]
classNameExpr := "source." elementName [classNameExpr]
elementName := templateId | identifier
templateId := ("|" identifier)
identifier := letter{[letter | digit]}
letter = A-Z | a-z
digit = 0-9
RENAME := ("{name=" elementName "}")

AttrSource := ("source." [elementName.] attrDef)
attrDef := (attributeName ":" attributeType ["=" defaultValue])
attributeName := (elementName [RENAME])
attributeType := (elementName [RENAME])
defaultValue := numericLiteral | stringLiteral
numericLiteral := (digit[{digit}] [ "." digit [{digit}] ])
stringLiteral := """ [ {[digit] [letter] } ] """

OpSource := ("source." [elementName "."] opDef
opDef := (operationName "(" allParams | "exclude <params>" ")" [":" returnType] )
operationName := (elementName [RENAME])
allParams := [params] {["," params]} | ["exclude <params>" ]
params := aParam1 | aParam2  | aParam3 | ParamExclude | ["new"] newParam | pStar | many
aParam1 := ( ["redefine"] elementName RENAME ":" ["redefine"] elementName)
aParam2 := ( ["redefine"] paramID ":" ["redefine"] elementName RENAME)
aParam3 := ( ["redefine"] paramID RENAME ":" ["redefine"] elementName RENAME)
many := "<params>"
paramID := (elementName RENAME)
paramType :=  (elementName RENAME)
paramExclude := "exclude" elementName ":"elementName
newParam := "new" elementName ":"elementName
pStar := ( ["redefine"] nameStar RENAME) | ( ["redefine"] nameStar LongRENAME)
nameStar := identifier [integer] "*"
integer := digit [{digit}]
LongRENAME := ("{name=" elementName:elementName
                                [{,elementName:elementNam}] "}")
returnType :=  (["redefine"] elementName RENAME)
        | (["new" | "exclude"] elementName)

RelationshipSource:="source." elementName [RENAME] [associationEnd][associationEnd]
associationEnd:= "redefine" elementName [RENAME]
        | "redefine" elementName ".multiplicity=" multiplicityValue
multiplicityValue := number ".." number
number = digit[{digit}]

PropertySource := "source." elementName  ["."subElement] ".property" "." metaAttribute
subElement := templateId | identifier
metaAttribute := identifier
```

Figure 4.21: EBNF Grammar for Transformation Directives.

- Repetition is indicated by: '{', '}' pairs.

```
NEW := NewClassName | NewAttr | NewOp | NewRelationship | NewProperty
NewClassName := ("new" elementName)
NewAttr := “new” (elementName “:” elementName [“=” defaultValue])
NewOp :=  “new” (elementName “(” [ “new” newParam [ {“,” “new”
        newParam} ] ] “)” [“:” elementName] )
NewRelationship := “new” elementName
NewProperty := “new” “property.” elementName “=” newValue
newValue := identitier | defaultValue

EXCLUDE := ClassExclude | AttrExclude | OpExclude | RelationshipExclude
ClassExclude := (“exclude” elementName)
AttrExclude := (ClassNameSource “exclude” [elementName “.”] dirFreeAttrDef)
OpExclude := (ClassNameSource “exclude” [elementName “.”] dirFreeOpDef)
RelationshipExclude := (“exclude” elementName)

REDEFINE := OpRedefine | AttrRedefine | PropertyRedefine
OpRedefine := ( ClassNameSource “redefine” OpDef )
AttrRedefine := ( ClassNameSource “redefine” attrRedef )
attrRedef := attrRedef-1 | attrRedef-2 | attrRedef-3
attrRedef-1 := (elementName RENAME “:” elementName [“=” attrDefaultValue])
attrRedef-2 := (elementName “:” elementName [“=” attrDefaultValue] RENAME)
attrRedef-3 := (elementName RENAME “:”
        elementName [“=” attrDefaultValue] RENAME)
PropertyRedefine := (“redefine” [elementName”.”]
        “property [“.” elementName] “.” identifier “=” value)
value := identifier | defaultValue
```

Figure 4.22: EBNF Grammar for Transformation Directives (part 2).

- Items are grouped using pairs of parenthesis: ‘(’, ‘)’.

The grammar shows the rules for the `source, redefine, new, exclude` and `rename` directives. The directives are organized divided into parts based on how the directives are used with model elements. The `SOURCE` directive has the following sub-rules:

- `ClassNameSource` describes the use of source directives with transformation schema classes and interfaces.
- `AttrSource` describes the use of source directives with transformation schema attributes.

- `OpSource` describes the use of source directives with transformation schema operations.
- `RelationshipSource` describes the use of source directives with transformation schema relationships.
- `PropertySource` describes the use of source directives with meta-attributes.

The `new` directive has the following sub-rules:

- `NewClassName` describes the use of the `new` directive to give a name to a class or interface.
- `NewAttr` describes the use of the `new` directive with transformation schema attributes.
- `NewOp` describes the use of the `new` directive with transformation schema operations.
- `NewRelationship` describes the use of `new` directives with transformation schema relationships.
- `NewProperty` describes the use of `new` directives with meta-attributes.

The `exclude` directive has the following sub-rules:

- `ClassExclude` describes the use of exclude directive with transformation schema classes and interfaces.
- `AttrExclude` describes the use of the exclude directive with transformation schema attributes.
- `OpExclude` describes the use of the exclude directive with transformation schema operations.

- `ParamExclude` describes the use of the exclude directive with formal parameters of a transformation schema operation.

- `RelationshipExclude` describes the use of exclude directive with transformation schema relationships.

The `redefine` directive has the following sub-rules:

- `AttrRedefine` describes the use of the redefine directive with transformation schema attributes.

- `OpRedefine` describes the use of the redefine directive with transformation schema operations.

- `PropertyRedefine` describes the use of the redefine directive with meta-attributes.

4.7 A Transformation Directive Processing Algorithm For Class Models

An algorithm for transforming source class models by processing directives in a transformation schema is shown in Figures 4.25 - 4.34. The algorithm is based on the metamodel shown in Figure 4.19 and Figure 4.20. The algorithm assumes class diagrams are represented as graphs and performs the following steps:

1. If source model conforms to source pattern then

   (a) Repeat for each connected component in the transformation schema graph:

      i. Repeat for each model element in a connected component:

         - Get a model element from the component.
         - Process the directives associated with the model element.

   (b) Copy all source model elements not referenced in the transformtion schema to the target model.

Figure 4.23 shows a partial call graph of the algorithm and Figure 4.24 shows a sequence diagram for the algorithm. The algorithm is based on the assumption that the source model conforms to the source pattern. Model conformance is beyond the scope of this dissertation. The core operations in the algorithm performs the following tasks:

1. The `main` operation in Figure 4.34 is the first operation executed. The command-line arguments supplied to `main` are the name of the file containing the source pattern, the name of the file containing the source model and the name of the file containing the class model transformation schema.

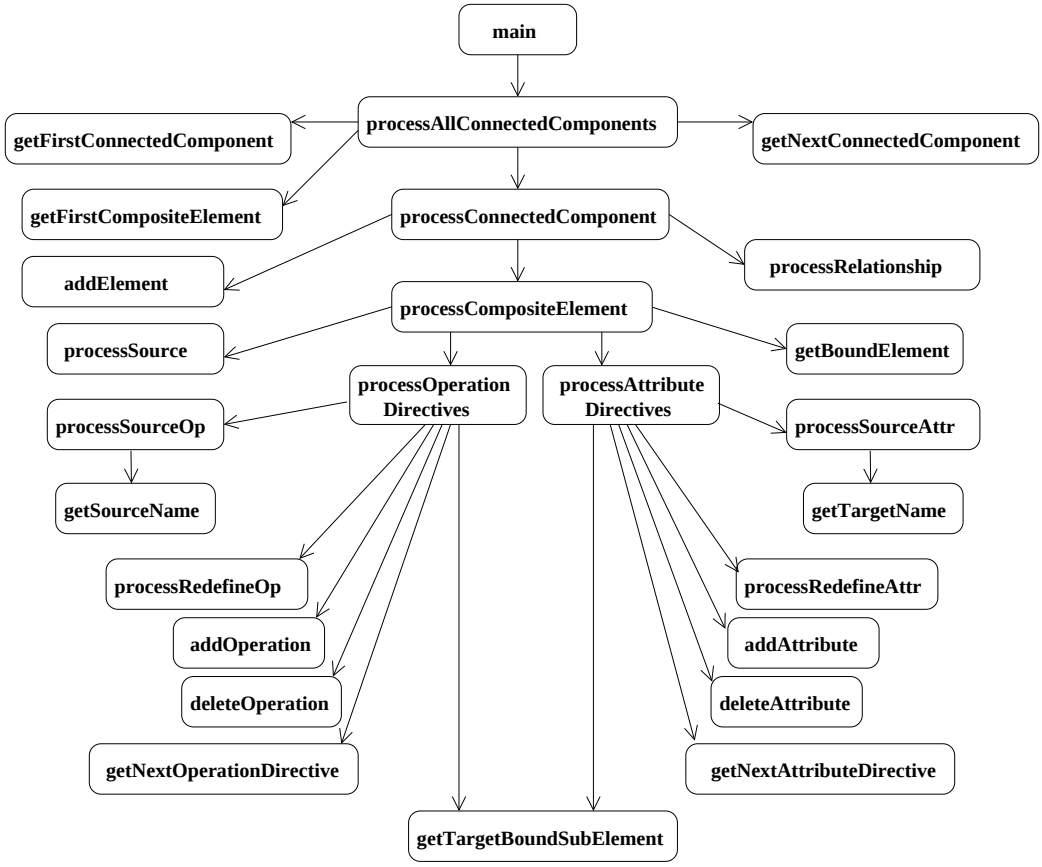

Figure 4.23: Call Graph of Algorithm for Processing Class Transformation Schemas.

The class constructor `ClassTransformer`, creates internal representations of these input files. The source class pattern is represented in the ClassTransformer class by `sourcePattern`, the class model transformation schema is represented by `transSchema`, the source class model is represented by `sourceModel` and the binding specification is represented by `bindings`. In the body of the `main` operation the transformation of the source model is initiated with the invocation of the `processAllConnectedComponents` operation. The class constructor is represented in the sequence diagram by the `create message` invocation on the `ClassTransformer` instance.

2. `processAllConnectedComponents` processes each connected graph component, that is, each set of model elements connected by relationships. Enumerated templates for example, normally do not have direct associations with other model elements and are therefore isolated components.

3. `processConnectedComponent` processes one component.

4. `processCompositeElement` processes a composite model element such as a transformation schema class or transformation schema interface.

5. `processNameDirectives` processes a transformation schema class or interface that has a source directive in the name compartment.

6. `processAttributeDirectives` processes all attribute directives for a transformation schema class.

7. `processSourceAttr` processes a transformation schema attribute source directive.

8. `processRedefineAttr` processes a redefine directive applied to an attribute.

9. `getTargetName` returns the target model name from a transformation directive. For example, the operation returns `Current` for a directive `source.|TransactionManager{name=Current}`.

10. `getTargetType` returns the target model type from a attribute or operation directive. For example, the operation returns `Current` for a directive `val:|TransactionManager{name=Current}`.

11. `processOperationDirectives` processes all operation directives for a transformation schema class or interface.

12. `copyConstraints` copies constraints from one model element to another.

13. `processSourceOp` processes a transformation schema operation source directive.

14. `processRedefineOp` processes a transformation schema operation redefine directive.

15. `getBoundElement` returns a reference to the source model element bound to a source pattern model element referenced by a directive.

16. `getBoundSubElement` returns a reference to an attribute, operation.

17. `getTargetBoundSubElement` returns a reference to an attribute or operation of a newly created class or interface. For example assume the `source.pCurrent` transformation schema class has the directive, `redefine someOp(x:float)`. If the `Current` class in the source model is bound to `pCurrent`, then the source directive results in the Current class being copied before the redefine directive is applied. Therefore the `getTargetBoundSubElement` operation is used to return a reference to the `someOp(x:float)` operation of the copied Current class rather that Current in the source model.

18. `getTargetBoundParameter` returns a reference to a parameter of an operation in a newly created class or interface. The operation is used, for example, when a redefine directive is applied to a parameter of an operation that is in a class to which a source directive is applied.

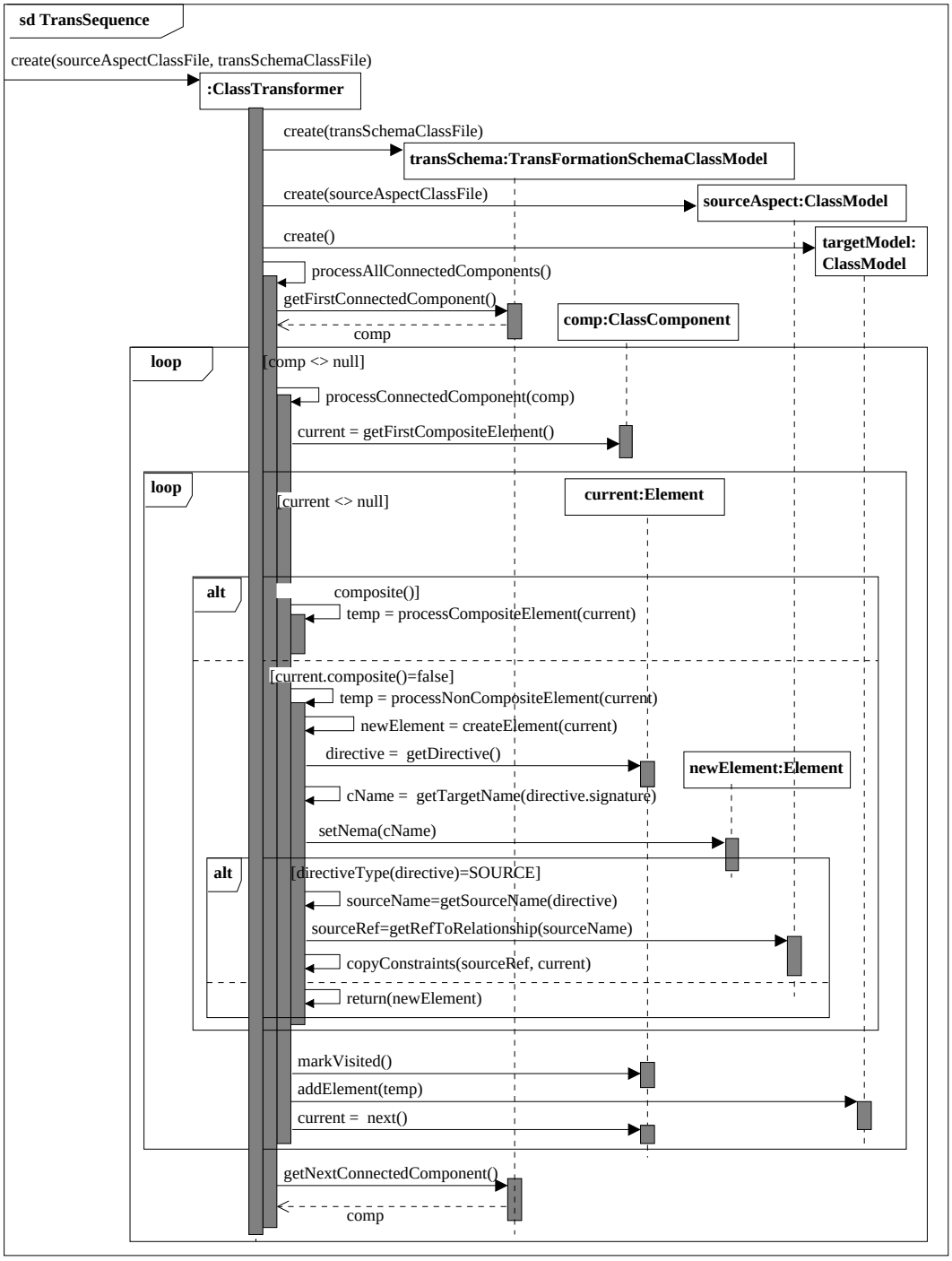

Figure 4.24: Sequence Diagram for Transformation Algorithm.

```
Class ClassTransformer begin
    ClassModel sourcePattern, sourceModel, targetModel
    TransformationSchemaClass transSchema
    Vector errorList, deletedList
    final int SOURCE=1, NEW=2, REDEFINE=3, RENAME=4, EXCLUDE=5
    TypeDefine TransformationSchemaElement TSElement

 public ClassTransformer(String patternFile, String modelFile, String schemaFile,
String[] bindingSpec)
  begin
    sourcePattern = new ClassModel(patternFile)
    sourceModel = new ClassModel(modelFile)
    transSchema = new TransformationSchemaClassModel(schemaFile)
    targetModel = new ClassModel()
    Table bindings = new Table(bindingSpec)
    errorList = new Vector()
    deletedList = new Vector()
  endMethod

   public void processAllConnectedComponents() begin
          ClassComponent comp = transSchema.getFirstConnectedComponent()
          while(comp != null)
             ClassComponent newModel = processConnectedComponent(comp)
             targetModel.addComponent(newModel)
              comp = transSchema.getNextConnectedComponent()
          endwhile
          copyUnreferencedElements(bindings, sourcePattern, sourceModel,  targetModel)
   endMethod

   public ClassComponent processConnectedComponent(ClassComponent comp)
begin
      ClassComponent newModel =new ClassModel()
      TSElement current = comp.getFirstElement() //TS class, TS interface or relationship
      while(current != null )
          if(current.composite()) //class or interface
                    Element temp = processCompositeElement(current)
          else // relationship
                    Element temp =processRelationship(current)
          endif
          current.markVisisted()
          if(temp != null)
                    newModel.addElement(temp)
          endif
          current = curent.next()
      endWhile
      removeDeletedElements(newModel)

      return newModel
   endMethod

public void removeDeletedElements(ClassComponent newModel) begin
      Integer firstElement = 0
      while(!deletedList.isEmpty())
          Element elm = deletedList.remove(firstElement)
          Element d = findElement(elm, newModel)
          newModel.deleteElement(d)
      endWhile
      deletedList.clear()
endMethod
```

Figure 4.25: Transformation Algorithm for Class Models (part 1).

```
public Element processCompositeElement(TSElement current) begin
Element newElement = null
Directive directive = current.getFirstNameCompartmentDirective()
Element boundElement = getBoundElement(bindings, directive)
if(boundElement == null)
        String s = directive.getSignature()
        errorList.add("Nonexistent  class or interface: " + s )
        return null
endif
Integer num = directive.getType()//SOURCE, NEW, etc.
  switch(num){
    case SOURCE:
        newElement = processNameDirectives(boundElement, current)
        if(current.type == CLASS )
                processAttributeDirectives(newElement, boundElement, current)
        endif
        processOperationDirectives(newElement, boundElement, current)
       break;
    case EXCLUDE: deletedList.add(boundElement)
       break;
    case NEW: newElement = createElement(current)
       break;
    default: Str s2=directive.getSignature()
        String s1 = "Invalid name-compartment directive encountered while processing: "
        errorList.add(s1 + s2)
  endSwitch
  return newElement
endMethod

public Element processNameDirectives(Element bound, TSElement current, Directive
directive) begin
   Element newElement = copyElement(bound)
   directive = current.getNextNameCompartmentDirective()
   while(directive != null)
      if(directive.getType() = SOURCE)
        bound = getBoundElement(bindings, directive)
        newElement.merge(bound)
      else
        String s1 = "Invalid name-compartment directive: " + directive.getSignature()
        String s2 = " encountered while processing: " + bound.getName()
        errorList.add(s1 +s1)
      endif
      directive = current.getNextNameCompartmentDirective()
   endwhile
   return newElement
endMethod

public Element copyElement(Element source) begin
        Element target = new Element(source.getElementType())
        property p = source.getFirstProperty()
        while(p != null) begin
                target.addProperty(p)
                p.markVisited()
                p = source.getNextProperty()
        endWhile
        return target
endMethod
```

Figure 4.26: Transformation Algorithm for Class Models (part 2).

```
public void processAttributeDirectives(Element newElm, Attribute boundElm,
TSElement current) begin
   Attribute newAttr, boundAttr
   Directive parent = current.getFirstNameCompartmentDirective()
   String psig = parent.getSignature()
   Directive attrDir = current.getFirstAttributeDirective()
   while(attrDir != null)
      Integer num = attrDir.getType()//SOURCE, NEW, etc.
      switch(num)
         case SOURCE:
                  String s1 = attrDir.getSignature()
                  String s2 =  boundElm.getName() + ":" + attrSig
                  boundAttr = getBoundSubElement(bindings, psig, attrDir)
                  if(boundAttr = null)
                           errorList.add("Nonexistent attribute: " + s2 )
                  else
                      newAttr = processSourceAttr(boundElement, boundAttr, attrDir)
                      newElm.addAttribute(newAttr)
                  endif
            break;
         case REDEFINE:
                  boundAttr = getTargetBoundSubElement(bindings, psig, attrDir, newElm)
                  processRedefineAttr(boundAttr, attrDir)
            break;
         case EXCLUDE:
                  boundAttr = getTargetBoundSubElement(bindings, psig, attrDir, newElm)
                  Element temp = newElm.deleteAttribute(boundAttr)
                  if(temp == null)
                      String s = boundAttr.getName()
                      errorList.add("attempt to delete nonexistent attribute: " + s )
                  endif
            break;
         case NEW:
                  newAttr = createAttribute(attrDir)
                  newElm.addAttribute(newAttr)
     default: Str s2=attrDir.getSignature() + " in " + psig
         String s1 = "Invalid attribute directive encountered while processing: "
         errorList.add(s1 + s2)
      endSwitch
      attrDir = current.getNextAttributeDirective()
   endWhile
endMethod

public Attribute processSourceAttr(Element boundElm, Attribute boundAttr,
AttributeDirective opDir) begin
         Attribute newAttr = copyElement(boundAttr)
         String attrSig = attrDir.getSignature()
         if(newAttributeName(attrSig)==true)
                  String attrName = getTargetName(attrSig)
                  newAttr.setName(attrName)
         endif
         if(newAttributeType(attrSig)==true)
                  String attrType = getTargetType(attrSig)
                  newAttr.setType(attrType)
         endif
         String attrValue = getDefaultValue(attrSig)
         if(attrValue != null)
                  newAttr.setDefaultValue(attrValue)
         endif

         return newAttr
endMethod
```

Figure 4.27: Transformation Algorithm for Class Models (part 3).

```
public void processRedefineAttr(Attribute boundAttr, AttributeDirective attrDir)
begin
        Boolean someChange = false
        String attrSig = attrDir.getSignature()
        if(newAttributeName(attrSig)==true)
                String attrName = getTargetName(attrSig)
                boundAttr.setName(attrName)
                someChange = true
        endif
        if(newAttributeType(attrSig)==true)
                String attrType = getTargetType(attrSig)
                boundAttr.setType(attrType)
                someChange = true
        endif
        String attrValue = getDefaultValue(attrSig)
        if(attrValue != null)
                boundAttr.setDefaultValue(attrValue)
                someChange = true
        endif
        if(someChange = false)
           errorList.add("Useless redefine directive: " + attrSig)
        endif
endMethod

// returns the middleware-specific name of "name". For example for the
// directive source.|TransManager{name=Current}, this operation returns Current.
  public String getTargetName(String sig) begin
        if(directiveType(sig)=SOURCE)
                return getDirectiveName(sig, "source.")
        else
            if(directiveType(sig)=REDEFINE)
                return getDirectiveName(sig, "redefine")
            else
                return getDirectiveName(sig, "")
 endMethod

   public String getDirectiveName(String sig, String sepatator) begin
        String[] str1 = sig.split(separator, 2)//e.g. source.TxMgr{name=Current}
        String[] str2 = str1[1].split(":", 2)//e.g. source.val:Integer
        String[] str3 = str1[1].split(")", 2)//e.g. source.op(...)
        if(str2.length == 1 AND str3.length == 1)
                String result = str1[1]
        else
           if(str3.length == 1)//attribute source directive
                String result = str2[0]
           else
              if(str2.length == 1)//operation source directive
                String result = str3[0]
        endif
        String[] str4 = result.split("name=", 2)//e.g. source.op(...)
        if(str4.length == 1)
                return result
        else
             result = str4[1].trim()//e.g. Current}
             return result.substring(0, result.length()-1)//e.g. Current
        endif
  endMethod
```

Figure 4.28: Transformation Algorithm for Class Models (part 4).

```
    public String getAttributeType(String sig, String sepatator) begin
          String[] str1 = sig.split(separator, 2)//e.g. source.val:Object{name=X}
          String[] str2 = str1[1].split(":", 2)//e.g. "val", "Object{name=X}"
          String[] str3 = str2[1].split("("name=", 2)//e.g. "Object{name=X}"
          if(str2.length == 1)//error
                return null
          else
             if(str3.length == 1)//attribute source directive
                return str2[1]
             else
               return str3[1].substring(0, str3[1].length()-1)//e.g. "X"
          endif
    endMethod

public void processOperationDirectives(Element newElm, Element boundElm,
TSElement current) begin
    Operation newOp, boundOp
    String psig = current.getSignature()
    Directive opDir = current.getFirstOperationDirective()
    String s1 = opDir.getSignature()
    String s2 =  boundElm.getName() + ":" + opSig
    String s3 = s1 + " of " + s2
    while(opDir != null)
       Integer num = opDir.getType()//SOURCE, NEW, etc.
       switch(num){
          case SOURCE:
                sourceOp = getBoundSubElement(bindings, boundElm, opDir)
                if(sourceOp == null)
                    errorList.add("Nonexistent operation in source model: " + s3 )
                else
                     newOp = processSourceOp(boundElement, sourceOp, opDir)
                     newElm.addOperation(newOp)
                endif
             break;
          case REDEFINE:
                sourceOp = getBoundSubElement(bindings, boundElm, opDir)
                if(sourceOp == null)
                    errorList.add("Nonexistent operation in source model: " + s3 )
                else
                   boundOp = getTargetBoundSubElement(bindings, psig, OpDir, newElm)
                   processRedefineOp(boundElement, boundOp, sourceOp, opDir)
                endif
             break;
          case EXCLUDE:
                boundOp = getTargetBoundSubElement(bindings, psig, OpDir, newElm)
                Element temp = newElm.deleteOperation(boundOp)
                if(temp == null)
                     String s = boundOp.getName()
                     errorList.add("attempt to delete nonexistent operation: " + s )
                endif
             break;
          case NEW:
                newOp = createOperation(opDir)
                newElm.addOperation(newOp)
          default: Str s2=opDir.getSignature() +" in "+current.getDirective.getSignature()
                String s1 = "Invalid operation directive encountered while processing: "
                errorList.add(s1 + s2)
       endSwitch
       opDir = current.getNextOperationDirective()
    endWhile
endMethod
```

Figure 4.29: Transformation Algorithm for Class Models (part 5).

```
public Operation processSourceOp(Element boundElm, Operation sourceOp,
OperationDirective opDir) begin
   Operation newOp = copyElement(sourceOp)
   processOpRenameDir(newOp, opDir)
   processParameters(sourceElm, newOp, sourceOp, opDir)

   return newOp
endMethod

public void processParameters(Element sourceElm, Operation newOp, Operation
sourceOp, OperationDirective opDir) begin
   Directive paramDir = opDir.getFirstParameterDirective()
   while(paramDir != null)
      Integer num = paramDir.getType()//REDEFINE, NEW, etc.
      switch(num)
         case RENAME:
                  boundParam = getTargetBoundParameter(newOp, boundOp, paramDir)
                  if(boundParam == null)
                     String s1 = paramDir.getSignature()
                     String s2 =  sourceElm.getName() + ":" + opSig
                     String s3 = s1 + " of " + s2
                     errorList.add("Nonexistent parameter in source model: " + s3 )
                  else
                     String paramSig = paramDir.getSignature()
                     String paramName = getTargetName(paramSig)
                     boundParam.setName(paramName)
                     String paramType = getTargetType(paramSig)
                     boundParam.setType(paramType)
                  endif
            break;
         case EXCLUDE:
                  String s1 = paramDir.getSignature()
                  String s2 =  sourceElm.getName() + ":" + opSig
                  String s3 = s1 + " of " + s2
                  boundParam = getTargetBoundParameter(newOp, boundOp, paramDirr)
                  if(boundParam == null)
                        errorList.add("attempt to delete nonexistent parameter: " + s3 )
                        continue
                  endif
                  Element temp = boundOp.deleteParameter(boundParam)
                  if(temp == null)
                        errorList.add("attempt to delete nonexistent parameter: " + s3 )
                  endif
            break;
         case NEW:
                   newParam = createParameter(paramDir)
                   position = paramDir.getParameterPosition()
                   newOp.addParameter(newParam, position)
         default:
               Str s2=paramDir.getSignature() + " of  " + opSig
               String str3 = str2  + " in " + boundElement.getName()
               String s1 = "Invalid parameter directive encountered while processing: "
               errorList.add(s1 + s3)
      endSwitch
      paramDir = opDir.getNextParameterDirective()
   endWhile
endMethod

public void processRedefineOp(Element sourceElm, Operation newOp, Operation
sourceOp, OperationDirective opDir) begin
   processOpRenameDir(newOp, opDir)
   processParameters(sourceElm, newOp, sourceOp, opDir)
endMethod
```

Figure 4.30: Transformation Algorithm for Class Models (part 6).

```
public void processOpRenameDir(Operation boundOp, OperationDirective
opDir) begin
        Boolean someChange = false
        String opSig = opDir.getSignature()
        if(newOperationName(opSig)==true)
                String opName = getTargetName(opSig)
                boundOp.setName(opName)
                someChange = true
        endif
        if(newOperationType(opSig)==true)
                String opType = getTargetType(opSig)
                boundOp.setType(opType)
                someChange = true
        endif
        if(someChange = false)
           errorList.add("Useless redefine directive: " + opSig)
        endif
endMethod

public Attribute createAttribute(Directive d) begin
   if(d != null)
        if(d.getType() == NEW)
           String sig = d.getSignature()
           String elmName = getTargetName(sig)
           String elmType = getTargetType(sig)
           String defaultValue = getDefaultValue(sig)
           Vector list = d.getMetaAttributes()
           Attribute newAttribute = new Attribute(elmName, elmType, defaultValue,
list)
           copyConstraints(newAttribute, d)
           return newAttribute
        endif
   endif
   return null
endMethod

public void createAttributesFromNewDirectives(Element newElement, TSElement
current) begin
   Directive d = current.getFirstAttributeDirective()
   while(d != null)
        if(d.getType() == NEW)
             Attribute newAttr = createAttribute(d)
             newElement.addAttribute(newAttr)
        endif
         d = current.getNextOperationDirective()
   endwhile
endMethod

public void copyConstraints(Element target, Element source) begin
        property p = source.getFirstConstraint()
        while(p != null)
                target.addConstraint(p)
                p.markVisited()
                p = source.getNextConstraint()
        endWhile
endMethod
```

Figure 4.31: Transformation Algorithm for Class Models (part 7).

```
public Element getBoundElement(Table bindings, Directive directive) begin
   String sig = directive.getSignature()
   String str = getTargetName(sig)
   Element ref = bindings.getElementBoundToName(str)

   return ref
endMethod

public Element getBoundSubElement(Table bindings, String parentSig, Directive
directive) begin
   String sig = directive.getSignature()
   String str = getTargetName(sig)
   String parent = getTargetName(parentSig)
   Element ref = bindings.getElementBoundToName(str, parent)

   return ref
endMethod

public Element getTargetBoundSubElement(Table bindings, String parentSig,
Directive directive, Element target) begin
   String sig = directive.getSignature()
   String str = getTargetName(sig)
   String parent = getTargetName(parentSig)
   Element temp = bindings.getElementBoundToName(str, parent)
   Element ref = target.findElement(temp)

   return ref
endMethod

public Element getTargetBoundParameter(Operation newOp, Operation
boundOp, Directive paramDir) begin
   String sig = paramDir.getSignature()
   String paramName = getTargetName(sig)
   Parameter p = boundOp.getParameter(paramName)
   Element ref = newOp.getParameter(p)

   return ref
endMethod

 public String getTargetType(String sig) begin
        if(directiveType(sig)=SOURCE)
                return getAttributeType(sig, "source.")
        else
            if(directiveType(sig)=REDEFINE)
                return getAttributeType(sig, "redefine")
            else
                return getDirectiveName(sig, "")
 endMethod
```

Figure 4.32: Transformation Algorithm for Class Models (part 8).

```
public Element createElement(TSElement current) begin
   Directive d = current.getDirective()
   String sig = d.getSignature()
   String elmName = getTargetName(sig)
   Vector list = current.getMetaAttributes()
   Element newElement = new Element(elmName, list)
   copyConstraints(newElement, current)
   if(current.getElementType() == OPERATION)
         createAttributesFromNewDirectives(newElement, current)
   endif
   if(current.getElementType() == OPERATION ||
                  current.getElementType() == INTERFACE)
         createOperationsFromNewDirectives(newElement, current)

   return newElement
endMethod

public Operation createOperation(Directive d) begin
   if(d != null)
         if(d.getType() == NEW)
            String sig = d.getSignature()
            String elmName = getTargetName(sig)
            String returnType = getTargetType(sig)
            Vector exceptions  = getExceptions(sig)
            Vector list = d.getMetaAttributes()
            Operation newOp = new Operation(elmName, returnType, exceptions, list)
            copyConstraints(newOp, d)
            return newOp
         endif
   endif
   return null
endMethod

public void createOperationsFromNewDirectives(Element newElement, TSElement
current) begin
   Directive d = current.getFirstOperationDirective()
   while(d != null)
         if(d.getType() == NEW)
               Operation newOp = createOperation(d)
               newElement.addOperation(newOp)
         endif
          d = current.getNextOperationDirective()
   endWhile
endMethod

public Parameter createParameter(Directive d) begin
   if(d != null)
         if(d.getType() == NEW)
            String sig = d.getSignature()
            String elmName = getTargetName(sig)
            String elmType = getTargetType(sig)
            Vector list = d.getMetaAttributes()
            Parameter newParam = new Parameter(elmName, elmType, list)
            copyConstraints(newParam, d)
            return newParam
         endif
   endif
   return null
endMethod
```

Figure 4.33: Transformation Algorithm for Class Models (part 9).

```
public Element processRelationship(TSElement current) begin
  Relationship sourceRel = getBoundElement(bindings, directive)
  Directive directive = current.getDirective()
  String sig = directive.getSignature()
  Integer num = directive.getType()
  switch(num)
    case EXCLUDE:
          if(sourceRel == null)
                errorList.add("attempt to delete nonexistent relationship: " + sig )
          else
                deletedList.add(sourceRel)
          endif
          return null
    break
    case NEW:
          Element newElement = createElement(sourceRel)
          return newElement
    break
    case SOURCE:
          Relationship sourceRel = getBoundElement(bindings, directive)
          if(sourceRel == null)
                errorList.add("attempt to delete nonexistent relationship: " + sig )
                return null
          else
              Element newElement = copyElement(sourceRel)
              if(newElement.getName() != getTargetName(sig))
                  newElement.setName(getTargetName(sig))
              endif
              return newElement
          endif
    break
    default:
          errorList.add("Invalid relationship directive: " + sig )
          return null
  endSwitch
endMethod

public main(String[] args) begin
     String pattern = args[0]
     String sourceFile = args[1]
     String transSchema = args[2]
     String bindings = args[3]
     ClassTransformer obj =
        new ClassTransformer(pattern, sourceFile, transSchema, bindings)
        obj.processAllConnectedComponents()
  endMethod
endClass
```

Figure 4.34: Transformation Algorithm for Class Models (part 10).

```
class Element begin
    public void merge(Element sourceElm) begin
        if(getElementType() != sourceElm.getElementType())
                return null
        Operation op = sourceElm.getFirstOperation()
        while(op != null)
            if(operationIsNotPresent(op))
                addOperation(op)
            endif
            op = op.next()
        endwhile
        if(getElementType() == CLASS)
            if(sourceElm.getElementType() == CLASS)
                Attribute attr = sourceElm.getFirstAttribute()
                while(attr != null)
                    if(attributeIsNotPresent(attr))
                        addAttribute(attr)
                    endif
                    attr = attr.next()
                endwhile
            endif
        endif
    endMethod
endClass
```

Figure 4.35: The `merge` Operation.

4.8 How the Algorithm Implements Rules for Processing Transformation Directives

A class model is defined as a collection of connected graph components, where each component is a set of classes and interfaces connected by relationships. The transformation algorithm processes one component on each iteration of the while-loop of the `processAllConnectedComponents` operation. On each iteration of the loop, the `processConnectedComponent` operation is called to effect the transformation of a component.

The `processConnectedComponent` operation processes a transformation schema class, a transformation schema interface or a transformation schema relationship during each iteration of its while-loop. The algorithm determines the order in which transformation schema classes, transformation schema interfaces and transformation schema relationships are processed using the `getFirstElement` and `next` operations. The first transformation schema model element to be processed is identified by the operation call: `TSElement current = comp.getFirstElement()`. Thereafter, model elements are selected to be processed by the call: `current = current.next()`. If the selected model element is a transformation schema class or interface, the model element is processed by the operation call: `Element temp = processCompositeElement(comp)`. Otherwise the selected model element is a transformation schema relationship. Transformation schema relationships are processed by the operation call: `Element temp = processRelationship(current)`.

Transformation schema model elements (classes, interfaces, etc) consist of one or more transformation directives. Directives are processed according to the transformation rule defined for each directive. The subsections that follow describe general rules for transforming class models and rules specific to each directive.

For each set of rules, (e.g., general rules), a list of rules is presented in one subsection (e.g. section 4.8.1.1) followed in the next subsection (section 4.8.1.2) by an explanation of how the algorithm implements each rule. Explanations are ordered based on the list of rules in the previous subsection. The first explanation is for the first rule, the second explanation is for the second rule, and so on.

4.8.1 Transformation Rules

Transformation rules are classified into general rules and specific rules.

4.8.1.1 General Rules

1. Directives in the name-directive compartment are processed before directives in the attribute-directive and operation-directive compartments.

2. For each name-directive, attribute-directive and operation-directive compartment, directives are processed in the order specified from top to bottom.

3. A rename directive is always associated with either a source or a redefine directive.

4. A name-directive compartment may contain a new directive, an exclude directive or source directives.

5. An attribute-directive compartment may contain a new directive, an exclude directive, a redefine directive or a source directive.

6. An operation-directive compartment may contain a new directive, an exclude directive, a redefine directive or a source directive.

7. Transformation schema relationships may have new, exclude and source directives only.

4.8.1.2 How the Algorithm Implements the General Rules

The algorithm implements the general transformation rules as follows:

1. Transformation schema classes or interfaces are being processed by the `processCompositeElement` operation (see Figure 4.26). Within the operation, directives in the name-directive compartment are processed as follows:

   - When the directive in the name-directive compartment is a `source` directive, the `Element newElement = processNameDirectives(boundElement, current)` operation call first processes name-compartment directives, before directives in the attribute-directive compartment (for transformation schema classes) and operation-directive compartment are processed by the `processAttributeDirectives(..)` and `processOperationDirectives(..)` operation calls respectively (see Figure 4.26).
   - When the directive in the name-directive compartment is an `exclude` directive, the exclude directive is processed, but attribute directives and operation directives are ignored. Exclude directives are processed by the `EXCLUDE` case of the switch statement in `processCompositeElement`.
   - When the directive in the name-directive compartment is a `new` directive, the `createElement` operation is called in the `NEW` case of the `switch` statement to process the transformation schema class or interfaces (see Figure 4.26). The `createElement` operation processes the new directive in the name-directive compartment before calling `createAttributesFromNewDirectives` and

`createOperationsFromNewDirectives` to process attribute directives and operation directives respectively (see Figure 4.33).

2. The algorithm enforces processing order of directives in compartments as follows. For the name-directive compartments, the first directive is obtained using the `getFirstNameDirective` operation call. This operation returns the directive at the top of the name-directive compartment. Other directives in the name-directive compartment are obtained by calls to the `getNextNameDirective` operation (see Figure 4.26). This operation returns directives in the order specified from top to bottom. Directives are processed in attribute-directive compartments and operation-directive compartments similarly. The `getFirstAttributeDirective` operation returns the first attribute directive and other attribute directives are obtained by calls to `getNextAttributeDirective` (see Figure 4.27). Similarly, `getFirstOperationDirective` returns the first operation directive and other operation directives are obtained by calls to `getNextOperationDirective` (see Figure 4.29).

3. A rename directive is only processed if it appears as part of a source or a redefine directive. Rename directives are processed by four operations: `processSourceAttr, processRedefineAttr, processSourceOp, processRedefineOp`. These operations are only called during the processing of a source directive or a redefine directive. The operations are called by the `processAttributeDirectives(..)` and `processOperationDirectives(..)` operations (see Figure 4.27 - Figure 4.30).

4. Directives in name-directive compartments are identified by the `switch`

statement in the `processCompositeElement` operation, and by the `if` statement in the `processNameDirectives` operation. Any directive other than `source, new` and `exclude` are trapped as errors in this code (see Figure 4.26 and Figure 4.27). The `processNameDirectives` operation identifies situations where the first directive in name-compartments is a source directive but a subsequent directive is not.

5. Directives in attribute-directive compartments are identified by the `switch` statement in the `processAttributeDirectives` operation. Any directive other than new, exclude, source or redefine are trapped as an error by the `default` case of the `switch` statement (see Figure 4.27).

6. Directives in operation-directive compartments are identified by the `switch` statements in the `processOperationDirectives`, `processSourceOp` and `processRedefineOp` operations (see Figure 4.29 and Figure 4.30). When processing operation-directives, any directive other than new, exclude, source or redefine are trapped as an error by the `default` case of the `switch` statement in `processOperationDirectives`. When processing parameters, any directive other than new, exclude, source or rename are trapped as an error by the `default` case of the `switch` statements in `processSourceOp` and `processRedefineOp`.

7. Directives associated with transformation schema relationships are identified by the `default` case of the `switch` statement in the `processRelationship` operation. Any directive other than a new, exclude or source directive is trapped as an error by the `default` case of the `switch` statement (see Figure 4.34).

4.8.1.3 Specific `source` Directive Rules

1. Source directives effect a deep copy of model elements so that when a model element is copied, associations and constraints are also copied.

2. When a name-directive compartment has multiple source directives, the name of the transformed model element is the name associated with the first source directive. Attributes and operations of model elements referenced by source directives other than the first, are merged with attributes and operations of the model element referenced by the first directive.

4.8.1.4 How the Algorithm Implements `source` Directive Rules

The algorithm implements the transformation rules for source directives as follows:

1. When a source directive is being processed, model elements are copied using the `copyElement` operation (see Figure 4.26). The operation copies all properties associated with a model element. This operation is called by: (1) `processNameDirectives` to copy a class or interface, (2) `processSourceAttr` to copy a class attribute, (3) `processSourceOp` to copy an operation, and (4) `processRelaionship` to copy a relationship (see Figure 4.26, Figure 4.27, Figure 4.30 and Figure 4.34 respectively).

2. The `processNameDirectives` operation processes the source directives in a name-directive compartment (see Figure 4.26). The source model element referenced by the first source directive is copied using the statement: `Element newElement = copyElement(bound)`, where `bound` is a reference to the source model element and `newElement` is a reference to the new model element created. The source model element referenced by other source directives are merged into the copied model element by the operation call:

`newElement.merge(bound)`, where `bound` is a reference to the source model element that is being added. The `merge` operation merges attributes and operations that are not already present in the `newElement` namespace (see Figure 4.35).

4.8.1.5 Specific `redefine` Directive Rules

1. A redefine directive associated with a transformation schema attribute is executed by: (a) processing a rename directive associated with the name of the attribute, and/or (b) processing a rename directive associated with the type of the attribute, and/or (c) changing the default value of the referenced attribute.

2. A redefine directive associated with a transformation schema operation is executed by: (a) processing a rename directive associated with the name of the operation, and/or (b) processing a rename directive associated with the return type of the operation, and/or (c) processing rename directives associated with parameters, and/or (d) processing new directives associated with parameters and/or (e) processing exclude directives associated with the parameters.

3. When used with transformation schema attributes and operations, a redefine directive must appear in a transformation schema class defined using a source directive in the name-directive compartment.

4.8.1.6 How the Algorithm Implements `redefine` Directive Rules

The algorithm implements the transformation rules for redefine directives as follows:

1. Transformation schema attribute redefine directives are processed

in the `REDEFINE` case of the `switch` statement in the `processAttributeDirectives` operation (See Figure 4.27). Processing is done in two steps:

- A reference to the attribute to be transformed is obtained by a call to the `getTargetBoundSubElement` operation.
- The `processRedefineAttr` operation (see Figure 4.28) is called to transform the attribute as follows: (a) The name of the attribute is changed in the first `if` statement. The `newAttributeName` operation returns true when the attribute directive has a rename directive attached to the name of the attribute. The statement, `String attrName = getTargetName (attrSig)` returns the name specified in the rename directive. For example, for the directive: `redefine counter{name=index }:Integer`, the operation returns `index`. The name of the attribute being redefined is changed using the statement: `boundAttr.setName(attrName)`.

  (b) The type of the attribute is changed in the second `if` statement. The `newAttributeType` operation returns true when the attribute directive has a rename directive attached to the type of the attribute. The statement, `String attrType = getTargetType(attrSig)` returns the name specified in the rename directive. For example, for the directive: `redefine counter:Integer{name=Long }`, the operation returns `Long`. The type of the attribute being redefined is changed using the statement: `boundAttr.setType(attrType)`.

  (c) The default value of the attribute is changed in the third `if` state-

ment in the operation. The last `if` statement identifies redefine attribute directives that do not effect any change to the target attribute.

2. Transformation schema operation redefine directives are processed in the `SOURCE` case of the `switch` statement in the `processOperationDirectives` operation (See Figure 4.29). Processing is done in three steps:

   - A reference to the source model operation that is being transformed is obtained by a call to the `getBoundSubElement` operation.
   - A reference to the copy made of the source model operation is obtained by a call to the `getTargetBoundSubElement` operation.
   - The operation is transformed by calling the `processRedefineOp` operation (See Figure 4.30). The operation: (a) processes rename directives associated with the name and the return type of the operation using the operation call: `processOpRenameDir(newOp, opDir)`, (b) processes all directives associated with parameters of the operation using the operation call: `processParameters (sourceElm, newOp, sourceOP, opDir)`.

3. The algorithm ensures that redefine directives for transformation schema attributes and operations always appear in a transformation schema class or interface that is defined using a source directive.

Directives in the name-directive compartment are processed by the `processCompositeElement(..)` operation. Redefine directives in the attribute-directive compartment and operation-directive compartment are processed by the `processRedefineAttr(..)` and `processRedefineOp(..)` operations respectively. The `processRedefineAttr(..)` operation is called

by `processAttributeDirectives(..)` and `processRedefineOp(..)` is called by `processOperationDirectives(..)`.

The `processAttributeDirectives(..)` and `processOperationDirectives(..)` operations are only called in the `SOURCE` case of the switch statement in `processCompositeElement(..)`. These operation calls are made after directives in the name-directive compartment have been processed by the operation call `Element newElement = processNameDirectives(boundElement, current)` (see Figure 4.26).

4.8.1.7 Specific `exclude` Directive Rules

1. A source model element that is referenced by an exclude directive must be eliminated from the target model.

2. When used with transformation schema attributes and operations, an exclude directive must appear in a transformation schema class defined using a source directive in the name compartment.

4.8.1.8 How the Algorithm Implements `exclude` Directive Rules

1. The exclude directive may be used with (1) transformation schema classes, interfaces and relationships, (2) transformation schema class attributes, (3) transformation schema operations, and (4) transformation schema parameters.

    - Exclude directives in name-directive compartments of transformation schema classes and interfaces, and exclude directives associated with transformation schema relationships do not result in any immediate change to the target model. Instead, references to

these model elements are stored to be processed after the transformation is complete. When the transformation is complete, the `removeDeletedElements` operation is called to determine if the target model has any excluded class, interface or relationship. This operation call, (`removeDeletedElements(newModel)`) is made in the `processConnectedComponent` operation (See Figure 4.25).

References to excluded classes and interfaces are stored using the statement: `deletedList.add(boundELement)`, in the `EXCLUDE` case of the `switch` statement in the `processCompositeElements` operation (See Figure 4.26). In the statement, `deletedList` is a vector and `boundElement` is a reference to the element to be excluded.

References to relationships are stored using the statement: `deletedList.add (sourceRel)`, in the `EXCLUDE` case of the `switch` statement in the `processRelationship` operation (see Figure 4.34). In this statement, `deletedList` is a vector and `sourceRel` is a reference to the relationship to be excluded.

- Exclude directives associated with transformation schema attributes are processed in the `EXCLUDE` case of the `switch` statement in the `processAttributeDirectives` operation (see Figure 4.27). Processing is done in two steps: (1) A reference to the attribute to be removed is obtained by the operation call: `boundAttr = getTargetBoundSubELement(bindings, psig, attrDir, newElm)`, and (2) The attribute is removed by the operation call: `newElm.deleteAttribute(boundAttr)`, where

`boundAttr` is a reference to the attribute that is being removed.

- Exclude directives associated with transformation schema operations are processed in the `EXCLUDE` case of the `switch` statement in the `processOperationDirectives` operation (see Figure 4.29). Processing is done in two steps: (1) A reference to the operation to be removed is obtained by the operation call: `boundOp = getTargetBoundSubELement (bindings, psig, opDir, newElm)`, and (2) The operation is removed by the operation call: `newElm.deleteOperation(boundOp)`, where `boundOp` is a reference to the operation that is being removed.
- Exclude directives associated with transformation schema parameters are processed in the `EXCLUDE` case of the `switch` statement in the `processParameters` operation. A reference to the parameter to be removed is obtained by a call to the `getTargetBoundParameter` operation. The parameter is then removed by the a call to the `deleteParameter` operation (see Figure 4.30).

2. As was the case for redefine directives, the algorithm ensures that exclude directives for transformation schema attributes and operations always appear in a transformation schema class or interface that is defined using a source directive. This is done by processing exclude directives in the attribute-directive compartment and operation-directive compartment after a source directive is identified in the name-directive compartment (see Figure 4.26).

4.8.1.9 Specific `new` Directive Rules

1. A new directive results in a new model element being created.

2. When a new directive is used in a name-directive compartment, all directives

in the attribute-directive compartment and the operation-directive compartment must be new directives.

4.8.1.10 How the Algorithm Implements `new` Directive Rules

1. The new directive may be used with (1) transformation schema classes, interfaces and relationships, (2) transformation schema class attributes, (3) transformation schema operations, and (4) transformation schema parameters.

    - A new directive in the name-directive compartment is processed in the `NEW` case of the `switch` statement in the `processCompositeElement` operation. A new class or interface is created and all directives in the transformation schema class or interface are executed by a call to the `createElement` operation (see Figure 4.26).
    - New directives associated with transformation schema attributes where the name-directive compartment has a source directive, are processed in the `NEW` case of the `switch` statement in the `processAttributeDirectives` operation. A new class attribute is created by a call to the `createAttribute` operation (see Figure 4.27).
    - New directives associated with transformation schema operations where the name- directive compartment has a source directive, are processed in the `NEW` case of the `switch` statement in the `processOperationDirectives` operation. A new operation is created by a `createOperation` operation call (see Figure 4.29).
    - New directives associated with transformation schema parameters where the name- directive compartment has a source directive, and the transformation schema operation has a source directive, are processed

in the `NEW` case of the `switch` statement in the `processParameters` operation. A new parameter is created by a `createParameter` operation call (see Figure 4.30).

- New directives associated with transformation schema parameters where the name-directive compartment has a source directive, and the transformation schema operation has a redefine directive, are processed in the `NEW` case of the `switch` statement in the `processParameters` operation. A new parameter is created by a `createParameter` operation call (see Figure 4.30).
- New directives associated with transformation schema relationships are processed in the `processRelationship` operation. A new relationship is created by a `Element newElement = createElement(sourceRel)` operation call, where `sourceRel` is a reference to the source model relationship being transformed.

2. The algorithm ensures that all attribute directives and operation directives that appear in a transformation schema class or interface defined using a new directive are new directives. The `createElement` operation is called in the `NEW` case of the `switch` statement in the `processCompositeElement` operation to process transformation schema classes or interfaces (see Figure 4.26). The `createElement` operation processes the new directive in the name-directive compartment before calling `createAttributesFromNewDirectives` and `createOperationsFromNewDirectives` to process attribute directives and operation directives respectively (see Figure 4.33). An error condition is recorded when the `createAttributesFromNewDirectives` and `createOperationsFromNewDirectives` operations detects a directive that

is not a new directive.

4.8.2 Summary

In addition to implementing the transformation rules, the algorithm also traps a number of error conditions including the following:

1. Attempts to delete a non-existent operation, attribute, parameter or relationship. This is done by examining the model element returned by `deleteOperation`,`deleteAttribute`, `deleteParameter` and `deleteRelationship` operation calls respectively. For example, in the `EXCLUDE` case of the `switch` statement in the `processOperationDirectives` operation, the statement: `if(temp == null){...}` records an attempt to delete a non-existent operation.

2. The absence of a required model element in the source model. For example, the absence of a required class or interface is detected by the following statements in `processCompositeElement` (see Figure 4.26):

```
if(boundElement == null)
    String s = directive.getSignature()
    errorList.add('Non-existent class or interface:  '  + s)
```

3. Invalid operation, attribute, parameter directives and invalid direcives in name-directive compartments. For example, invalid direcives in name-directive compartments are identified by the `default` case of the `switch` statement in the `processCompositeElement` operation, and in the `else` condition of the `if` statement in the `processNameDirectives` operation (see Figure 4.26).

It is important to note that combinations of different parts of the algorithm do not violate the transformation rules. Several parts of the algorithm can only affect a rule if a target model element may be produced by applying the directives associated with two or more transformation schema model element to the same source model element. For example, if an operation may be created by applying two or more operation directives to the same source model operation. This is not possible. Each time a directive is applied to a source model element, a target model element results. Target model elements are never modified by any directive in the transformation schema.

4.9 Lessons Learned

The order of directives in transformation schema compartments is important. This is illustrated by the transformations described in Figure 4.36, Figure 4.37 and Figure 4.38.

Figure 4.36 shows two transformation schemas, each containing a transformation schema class with an exclude directive and a source directive. `Transformation Schema A` has the exclude directive before the source directive. `Transformation Schema B` has the directives in reverse order: the source directive followed by the exclude directive.

The source model contains two classes, `MoneyTransfer` and `TransManager`. `MoneyTransfer` is bound to `|MoneyTransfer` in the source pattern and `TransManager` is bound to `|TransManager`. The `commit(x:int):boolean` operation in `MoneyTransfer` is bound to the `|commit(|x:type):|commitVal` operation template in `|MoneyTransfer`. Similarly, the `commit(x:int):boolean` operation in `TransManager` is bound to the `|commit(|y:type2):|commitVal2` operation template in `|TransManager`.

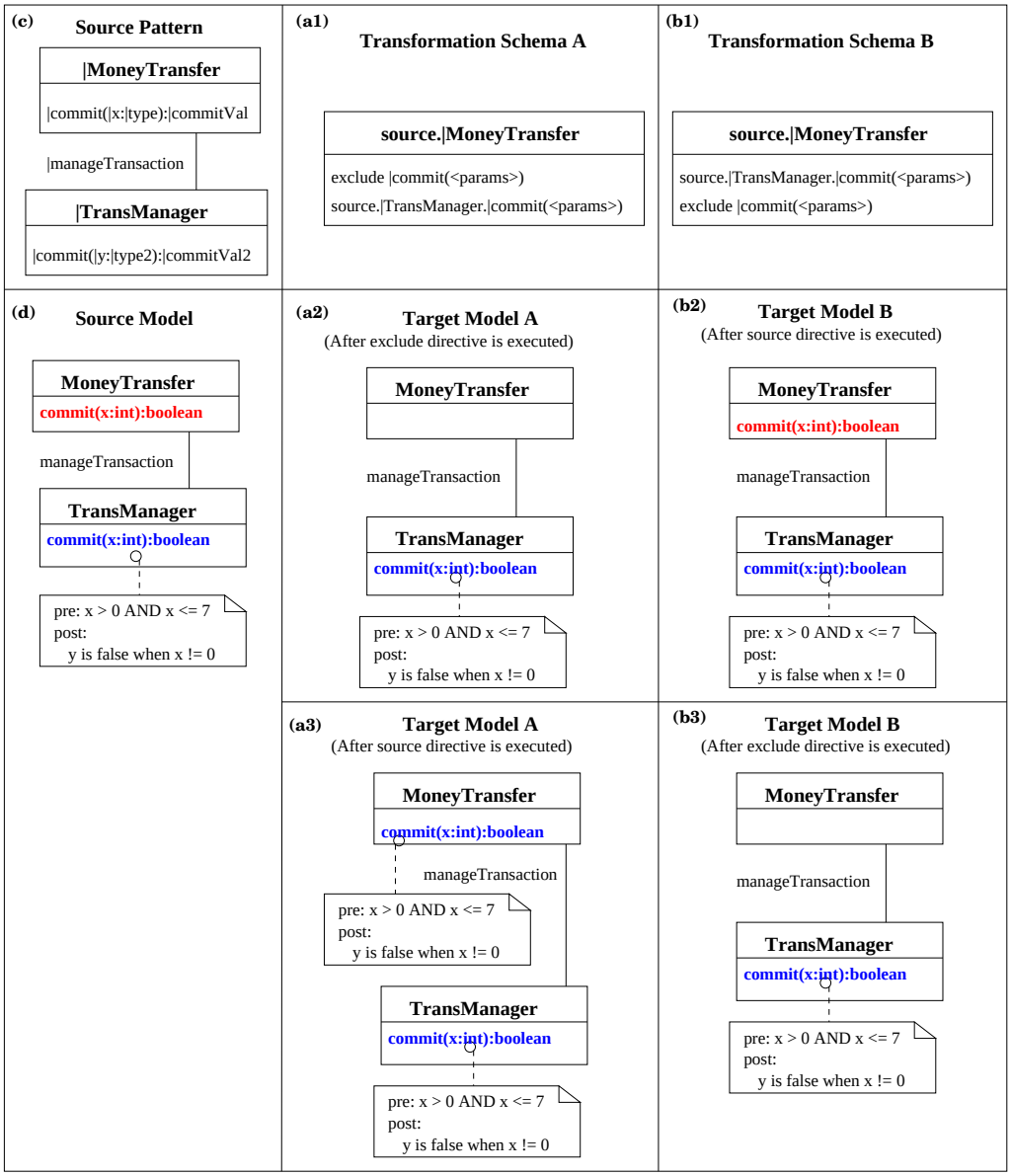

Figure 4.36: Ordering Transformation Directives - Example 1.

Figure 4.36 (a2) and Figure 4.36 (a3) show the effect of executing the directive in the order specified in `Transformation Schema A` and Figure 4.36 (b2) and (b3) show the effect of executing the directive in the order specified in `Transformation Schema B`.

For `Transformation Schema A`, the exclude directive results in the dele-

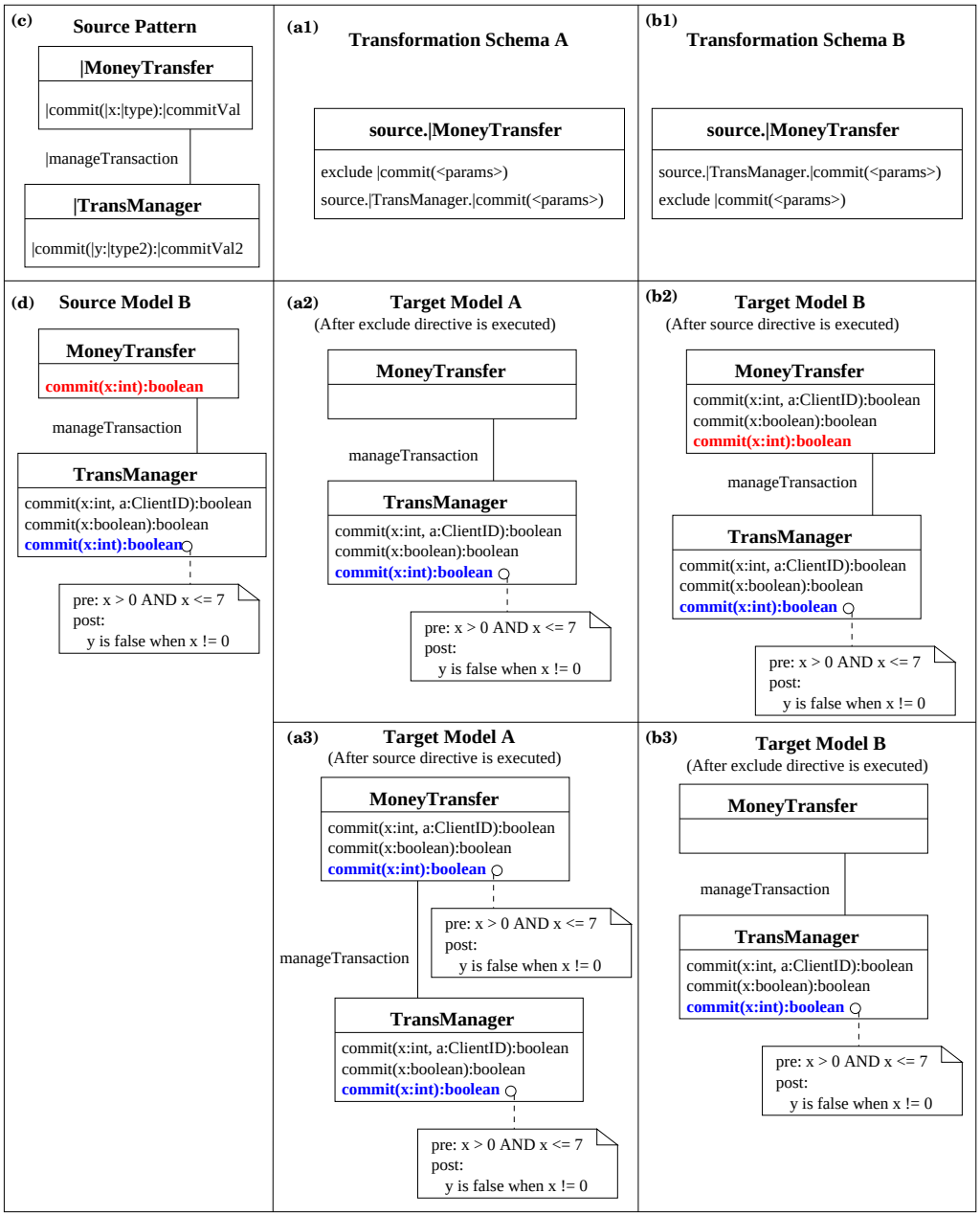

Figure 4.37: Ordering Transformation Directives - Example 2.

tion of the `commit` operation copied from the source model by the `source.MoneyTransfer` directive (see Figure 4.36 (a2)). The source directive then results in the `commit` operation being copied from the `TransManager` class.

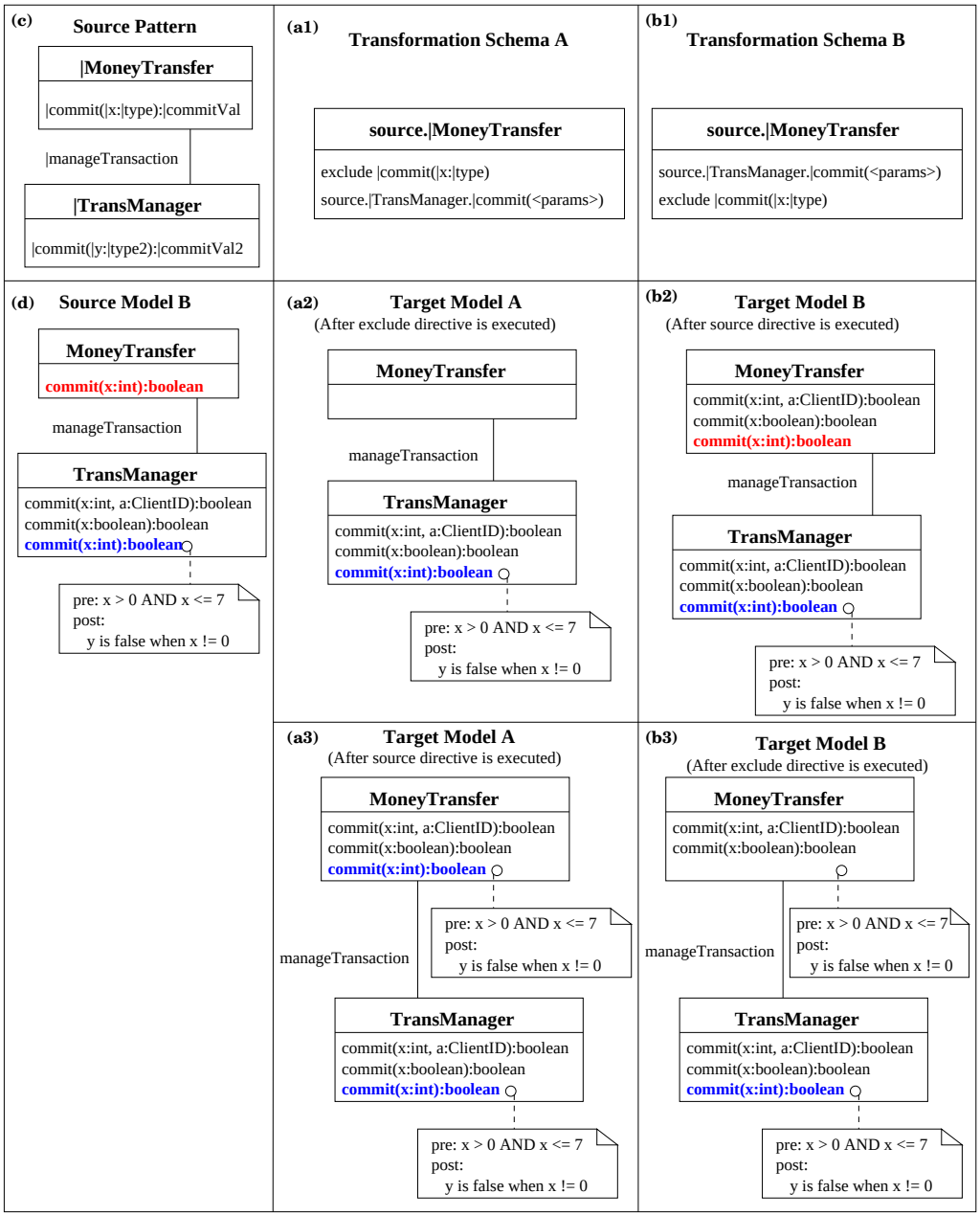

Figure 4.38: Ordering Transformation Directives - Example 3.

Therefore, after the source directive in `Transformation Schema A` is executed, the `MoneyTransfer` class has a copy of the `commit` operation defined in the `TransManager` class.

For `Transformation Schema B`, the source directive is first executed. The execution of the directive does not result in any change in the state of the `MoneyTransfer` class in Figure 4.36 (b2) because an operation with the identical signature to the one being copied from the `TransManager` class already exists in the `MoneyTransfer` class. When the exclude directive is executed, the `commit` operation in the `MoneyTransfer` class is deleted. Therefore, in contrast with the transformed model for `Transformation Schema A` where the `MoneyTransfer` class has an operation (Figure 4.36 (a3)), the transformed model for `Transformation Schema B` does not have an operation in the `MoneyTransfer` class (Figure 4.36 (b3)).

Figure 4.37 defines the same transformations specified in Figure 4.36 but uses a different source model. In this case the order of directives defined in `Transformation Schema A` result in a transformed model with three new operations (see Figure 4.37 (a3)). In contrast, the order of directives in `Transformation Schema B` result in a transformed model with no operations in the `MoneyTransfer` class because the exclude directive results in the removal of the three operations that were copied from the `TransManager` class (see Figure 4.37 (b2) and (b3)).

If the goal of the modeler in the two examples above (Figure 4.36 and Figure 4.37) was to have the `MoneyTransfer` class in the transformed model having similar operations to `TransManager`, (for example, if `MoneyTransfer` is to act as a proxy for `TransManager`), then the order of directives specified in `Transformation Schema A` is the correct sequence. In these examples it is fairly easy to detect the error resulting from the order of directives shown in `Transformation Schema B` because the transformed `MoneyTransfer` class has no operations. Figure 4.38 illustrates a scenario where the error resulting from the execution order specified in `Transformation Schema B` is more difficult to detect.

In Figure 4.38, the `commit(x:int):boolean` operation in `MoneyTransfer` is bound to the `|commit(|x:type):|commitVal` operation template in `|MoneyTransfer`. In addition, the three `commit` operations in `TransManager` are all bound to the `|commit(|y:type2):|commitVal2` operation template in `|TransManager`. The order of directives specified by `Transformation Schema A` results in the `MoneyTransfer` class in the transformed model having three `commit` operations (Figure 4.38 (a2) and (a3)). The order of directives specified by `Transformation Schema B` results in one operation being deleted from `MoneyTransfer` class in the transformed model (Figure 4.38 (b2) and (b3)). In this scenario, it may be difficult for a modeler to identify that one `commit` operation is missing from the class.

4.10 Discussion: Use of Target Patterns to Validate Transformations

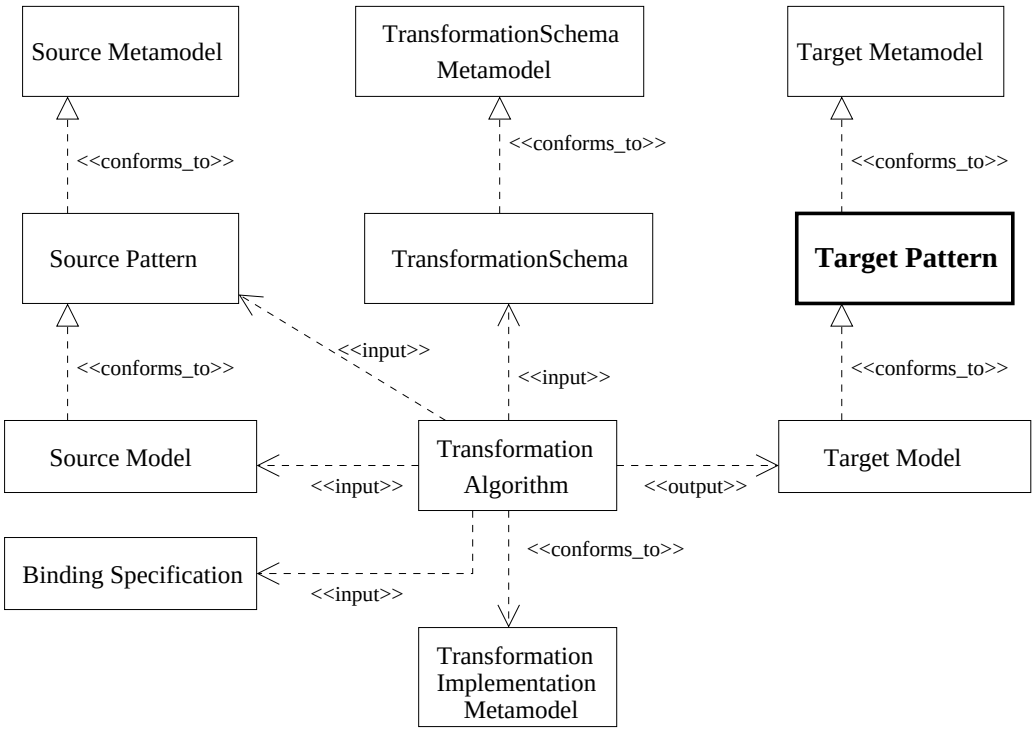

Figure 4.39: Transformation Conceptual Model With Target Pattern.

In declarative approaches to model transformation such as QVT, a transformation specification consists of a source model and a target model. The source model is a pattern that describes valid input models and the target model is a pattern that describes valid output models. In this dissertation, a transformation specification consists of a source pattern that describes valid input models and a transformation schema that describes how output (i.e., target) models are obtained from input models.

In addition to the source pattern, a target pattern may be used as an oracle to help determine the correctness of model transformations. This is illustrated in Figure 4.39. In the figure, `Target Pattern` is a description of valid `Target`

`Models` where a target model is a class diagram that describes the transformed software feature. Each target pattern describes the minimum set of properties expected of valid target models. The `Target Pattern` describe a subset of instance of the `Target Metamodel`.

A correct transformation algorithm rejects all invalid source models and so an incorrect target model may only result from (1) an incorrect source pattern, (2) an incorrect transformation schema, or (3) a faulty transformation algorithm. If the transformation algorithm is correct, then an incorrect target model may only result from a faulty source pattern or a faulty transformation schema. Therefore, a target pattern may be used to validate the correctness of source patterns, transformation schemas and transformation algorithm. This may be done in several ways:

1. An target model that does not conform to a correct target pattern indicates a faulty transformation schema when the source pattern and transformation algorithm are correct.

2. An target model that does not conform to a correct target pattern indicates a faulty algorithm when the source pattern and transformation schema are correct.

3. An target model that does not conform to a correct target pattern indicates a faulty source pattern when the transformation schema and transformation algorithm are correct.

This suggests a test case design approach. For example, a transformation algorithm should always produce correct output when exercised using source patterns and transformation schemas that are known to be correct.

A target pattern may contain three types of model elements: source pattern model elements copied without modification by transformation directives, source

pattern model elements modified by one or more transformation directives and new model elements created by one or more transformation directives. Therefore, all model elements in a target model result from use of transformation directives in the transformation schema. It is therefore possible to create a target pattern from the source pattern and the transformation schema. The ease or difficulty of this process will depend on the complexity of the transformation schema.

Chapter 5

Pilot Studies: Transforming Distribution Class Models

This chapter presents two pilot studies to illustrate the application of the transformation algorithm to a platform independent distribution class model. The first pilot study uses the transformation language to transform a platform independent distribution class model into a CORBA distribution class model. The second pilot study uses the transformation language to transform the platform independent distribution class model into a Jini distribution class model. The transformation context is illustrated by the activity diagram shown in Figure 4.1.

5.1 Pilot Study 1: Transforming Distribution Class Model to CORBA Class Model

This section illustrates the transformation a platform-independent server distribution class model into a CORBA class model. CORBA is an OMG standard for open distributed object computing.

5.1.1 CORBA Support For Server Distribution

Server distribution is the process of making a server object available to remote clients. In CORBA, distribution can be realized by: (1) initializing an object

request broker (ORB) through which clients communicate with the server, (2) initializing a portable object adapter that is used to manage object properties (e.g, object persistence) and to invoke operations on objects on behalf of the ORB and (3) registering the server with the CORBA naming service. CORBA provides a number of features to support the distribution of server objects, these include:

1. The CORBA object request broker (ORB): a collection of libraries, processes, and other infrastructure that connects objects requesting services to the objects providing the services and allows the objects to communicate by sending and receiving messages. In order for a server object to be distributed, the ORB must be initialized using its `init` operation and made to enter an infinite request processing loop by invoking its `run` operation.

2. An object adapter: the ORB component responsible for managing object references, object activation, and object state. An object adapter maintains a mapping of object references to actual objects, and provides support for invoking operations on the objects on behalf of the ORB. In CORBA, the Portable Object Adapter, or POA, is a CORBA object adapter designed to: (1) allow programmers to construct object implementations that are portable between different ORB products, (2) provide support for objects persistence, (3) support the transparent activation of objects and (4) allow a single object to support multiple object identities simultaneously. Since an application may have multiple POAs, the first POA, called the `rootPOA` is typically obtained from the ORB. CORBA provides a number of classes and interfaces to support the POA role, these include: a `POA` interface, a `POAHelper` class and a `POAManager` interface.

3. The CORBA interface definition language (IDL) [40]: an OMG defined lan-

guage standard for defining interfaces for CORBA objects. An IDL interface may contain operations, exceptions, and attributes. An IDL file specifying the operations that the server will provide, must be created in order for a server to use the services and facilities provided by the ORB.

5.1.2 Source Class Pattern

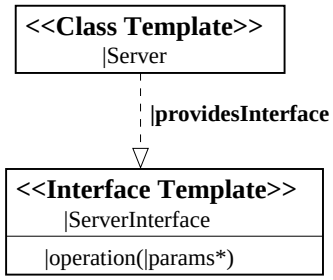

Figure 5.1: Source Pattern.

Figure 5.1 shows a source pattern that describes services that are to be distributed using middleware. The figure is expressed as an RBML class diagram template. The *Server* class template represents the application that is to be distributed. This application must have an associated provided interface represented by *ServerInterface*.

5.1.3 Specify CORBA Model Transformations

The mappings to transform the generic distribution class model into a CORBA distribution model is specified as a transformation schema as shown in Figure 5.2. The `POAManager` transformation schema interface and the `POAHelper` transformation schema class represent parts of the CORBA POA that support preparation of the middleware infrastructure. `POAHelper` provides the `narrow` operation that is used to provide a reference to the POA while the `POAManager` provides the

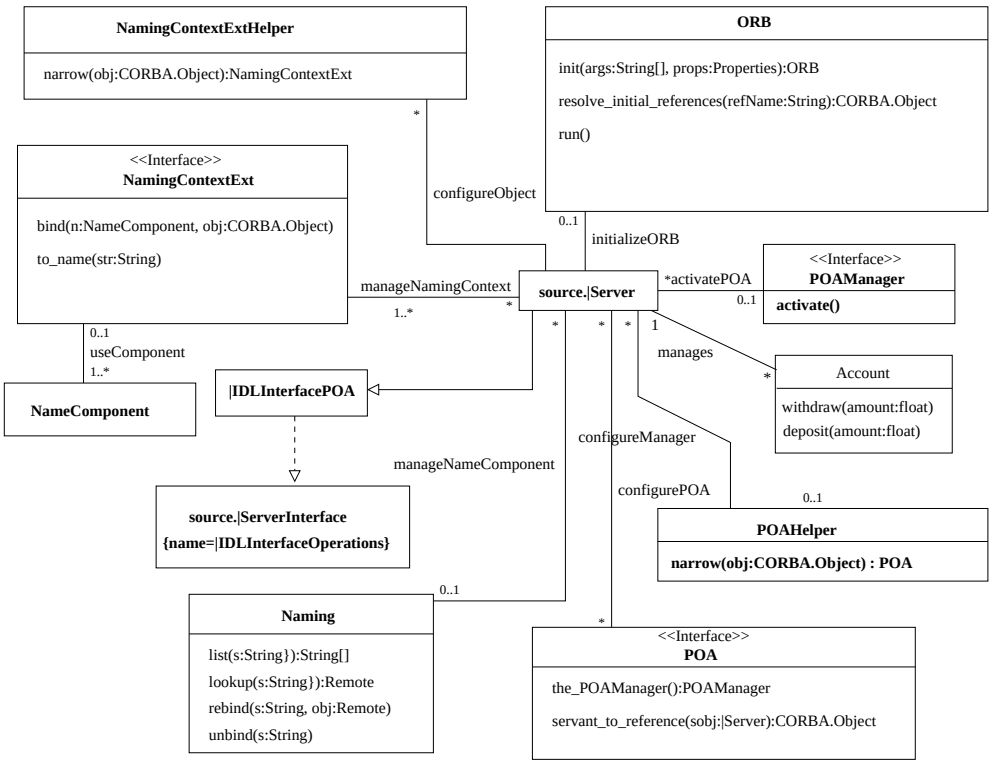

Figure 5.2: CORBA Class Transformation Schema.

`activate` operation that is used to change the state of the POA-manager to `active` and cause associated POAs to start processing requests.

CORBA requires the creation of an IDL interface file for the server. An IDL interface is similar to a Java interface but may also contain attributes and exceptions. The `source.|ServerInterface{name= |IDLInterfaceOperations}` transformation schema interface represents a Java interface generated from the IDL interface by the CORBA IDL compiler and the `|IDLInterfacePOA` transformation schema class represents a Java class generated.

5.1.4 Source Model and Binding Specification

Figure 5.3 shows a valid source model for the source pattern. The source model consists of the `AccountManager` and `Account` classes and the `AccountOperations`

interface. The `AccountManager` class is bound to `AccountManager` class template and the `AccountOperations` interface is bound to `AccountInterface`. Note that the `Account` class and `manages` association are not bound to any source pattern model elements. The complete binding specification is shown in Table 5.1.

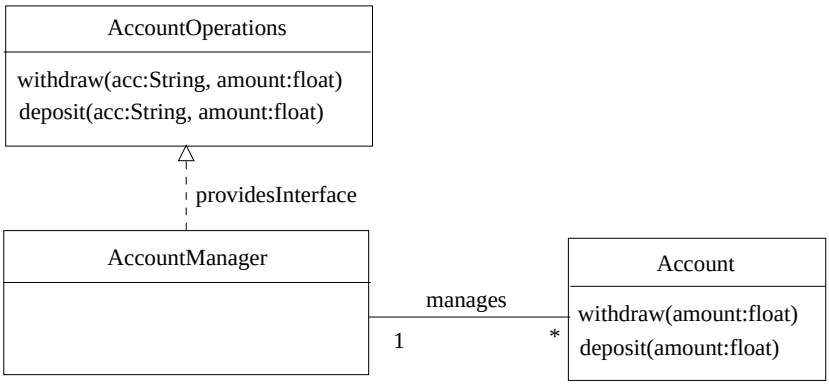

Figure 5.3: Source Model for Server Distribution.

Table 5.1: Binding Specification.

Source Pattern Element	**Source Model Element**
|Server	AccountManager
|ServerInterface	AccountOperations
|ServerInterface::|operation	AccountOperations::withdraw
|ServerInterface::|operation::|params	AccountOperations::withdraw::acc:String
|ServerInterface::|operation::|params	AccountOperations::withdraw::amount:float
|ServerInterface::|operation	AccountOperations::deposit
|ServerInterface::|operation::|params	AccountOperations::deposit::acc:String
|ServerInterface::|operation::|params	AccountOperations::deposit::amount:float
|providesInterface	providesInterface

5.1.5 Process Transformation Directives

In order to provide a systematic way of describing the order in which transformation schema model elements are processed, the following `element selection heuristic` is used.

1. The first element to be processed is selected arbitrarily.

2. After the first element has been processed, other model elements are selected to be processed based on the lexographic ordering of names of model elements. An element with a smaller name in lexographic ordering is chosen before an element with a larger name.

The transformation schema contains a single set of connected model elements and therefore the loop in the `processAllConnectedComponents` operation in the class model transformation algorithm will be executed only once. A new component is created prior to the `while` loop in `processConnectedComponent` and a transformed source model element is added to this new component for each iteration of the while-loop in `processConnectedComponent`. On each iteration of this loop, all the directives associated with the source model element that is being transformed are processed. Directives associated with a single source model element are expressed in one transformation schema model element. Each numbered item below lists the transformation schema model elements and directives processed during the corresponding iteration of the the while-loop in `processConnectedComponent` operation. For example, the `source.|Server` transformation schema class and the `source.|Server` transformation directive are processed during the first iteration of the loop.

- **Iteration 1:** The `source.|Server` transformation schema class is processed by a `processCompositeElement` operation call in the first iteration of the

`while loop` in `processConnectedComponent`. From the binding specification, the `|Server` class template in the source pattern is bound to the `AccountManager` class in the source model. Processing results in the `AccountManager` class being copied to the new component.

- **Iteration 2:** The `activatePOA` transformation schema association is the next item processed. Associations are processed by calls to `processNonCompositeElement`. A transformation schema model element that is specified without using a leading directive keyword defaults to '`new`' as the leading directive keyword. In this instance, processing results in a new association with the name `activatePOA` being created and inserted into the new component.

- **Iteration 3:** The `POAManager` transformation schema interface is processed, resulting in the creation of a new interface with the name `POAManager`. The `activate()` transformation schema operation is a new directive. Processing results in the creation of the operation: `activate()`. The operation is inserted into the newly created `POAManager` interface and the interface is inserted into the new component.

- **Iteration 4:** The `configureManager` transformation schema association is processed. Processing results in a new association with the name `configureManager` being created and inserted into the new component.

- **Iteration 5:** The `POAHelper` transformation schema class is processed resulting in the creation of a new class with the name `POAHelper`. The embedded directive: `narrow(obj:CORBA.Object):POA` results in the creation of the operation: `narrow(obj:CORBA.Object):POA`. The operation is inserted into the newly created `POAHelper` class and the class is inserted into

the new component.

- **Iteration 6:** The `configureObject` transformation schema association is processed. Processing results in a new association with the name `configureObject` being created and inserted into the new component.

- **Iteration 7:** The `NamingContextExtHelper` transformation schema class is processed resulting in the creation of a new class with the name `NamingContextExtHelper`. The embedded directive: `narrow obj:CORBA.Object):NamingContextExt` results in the creation of the operation: `narrow(obj:CORBA.Object):NamingContextExt`. The operation is inserted into the newly created `NamingContextExtHelper` class and the class is inserted into the new component.

- **Iteration 8:** The `configurePOA` transformation schema association is processed. Processing results in a new association with the name `configurePOA` being created and inserted into the new component.

- **Iteration 9:** The `POA` transformation schema interface is processed resulting in the creation of a new interface with the name `POA`. The embedded directives are processed as follows:

    1. The directive `the_POAManager():POAManager` is processed resulting in the creation of the operation: `the_POAManager():POAManager`. The operation is inserted into the newly created `POA` interface.
    2. The directive `servant _to_reference(sobj:|Server):CORBA.Object` is processed resulting in the creation of the operation: `servant_to_reference(sobj:|Server):CORBA.Object`. The operation is inserted into the newly created `POA` interface.

The POA interface is inserted into the new component.

- **Iteration 10:** The generalization relationship between source.|Server and |IDLInterfacePOA is processed. Processing results in a new generalization being created and inserted into the new component.

- **Iteration 11:** The |IDLInterfacePOA transformation schema class is processed. Processing results in a new class with the name |IDLInterfacePOA being created and inserted into the new component.

- **Iteration 12:** The realization dependency between |IDLInterfacePOA and source.|ServerInterface{name=|IDLInterfaceOperations} is processed. Processing results in a new realization dependency being created and added to the new component.

- **Iteration 13:** The source.|ServerInterface{name= |IDLInterfaceOperations} transformation schema interface is processed. Processing results in a new interface with the name |IDLInterfaceOperations being created. The |operation(|params*) operation template is bound to the withdraw and deposit operations. As a result these operations are copied to the new interface. The new interface is then inserted into the new component.

- **Iteration 14:** The initializeORB transformation schema association is processed. Processing results in a new association with the name initializeORB being created and inserted into the new component.

- **Iteration 15:** The ORB transformation schema class is processed. A new class with the name ORB is created. The embedded directives are processed as follows:

1. The directive `init(args:String[], props:Properties):ORB` is processed resulting in the creation of the operation: `init(args:String[], props:Properties):ORB`. The operation is inserted into the created ORB class.
2. The directive `resolve_initial_references(obj: String):CORBA.Object` is processed resulting in the creation of the operation: `resolve_initial _references(obj:String):CORBA.Object`. This operation is used to create a CORBA reference for a server object. The operation is inserted into the created ORB class.
3. The directive `run()` is processed resulting in the creation of the operation: `run()`. The operation is inserted into the created ORB class.

The ORB class is inserted into the new component.

- **Iteration 16:** The `manageNamingContext` transformation schema association is processed. Processing results in a new association with the name `manageNamingContext` being created and inserted into the target model.
- **Iteration 17:** The `NamingContextExt` transformation schema interface is processed resulting in the creation of a new interface with the name `NamingContextExt`. The embedded directives are processed as follows:

  1. The directive `bind (n:NameComponent, obj: CORBA.Object)` results in the creation of the operation: `bind(name=n:NameComponent, obj:CORBA.Object)`. The operation is inserted into the newly created `NamingContextExt` interface.

2. The directive `to_name (str:String)` results in the creation of the operation: `to_name(str:String)`. The operation is inserted into the newly created `NamingContextExt` interface.

The `NamingContextExt` interface is inserted into the new component.

- **Iteration 18:** The `useComponent` transformation schema association is processed. Processing results in a new association with the name `useComponent` being created and inserted into the new component.

- **Iteration 19:** The `NameComponent` transformation schema class is processed resulting in a new class with the name `NameComponent` being created and inserted into the new component.

- **Iteration 20:** The `manageNamingService` transformation schema association is processed. Processing results in a new association with the name `manageNamingService` being created and inserted into the new component.

- **Iteration 21:** The `Naming` transformation schema class is processed resulting in a new class with the name `Naming` being created. The embedded directives are processed as described below.

  1. The `list (s:String):String[]` transformation schema operation is processed. Processing results in the operation `list(s:String):String[]` being created and inserted into the new class.
  2. The `lookup (s:String):Remote` transformation schema operation is processed. Processing results in the operation `lookup(s:String):Remote` being created and inserted into the new class.

3. The `rebind (s:String, obj:Remote)` transformation schema operation is processed. Processing results in the operation `rebind(s:String, obj:Remote)` being created and inserted into the new class.

4. The `unbind (s:String)` transformation schema operation is processed. Processing results in the operation `unbind(s:String)` being created and inserted into the new class.

The `Naming` class is inserted into the new component.

The `Account` class and the `manages` association are not bound to any source pattern model elements. These items are copied to the new component and the new component is added to the target model. The resulting CORBA distribution class model is shown in Figure 5.4.

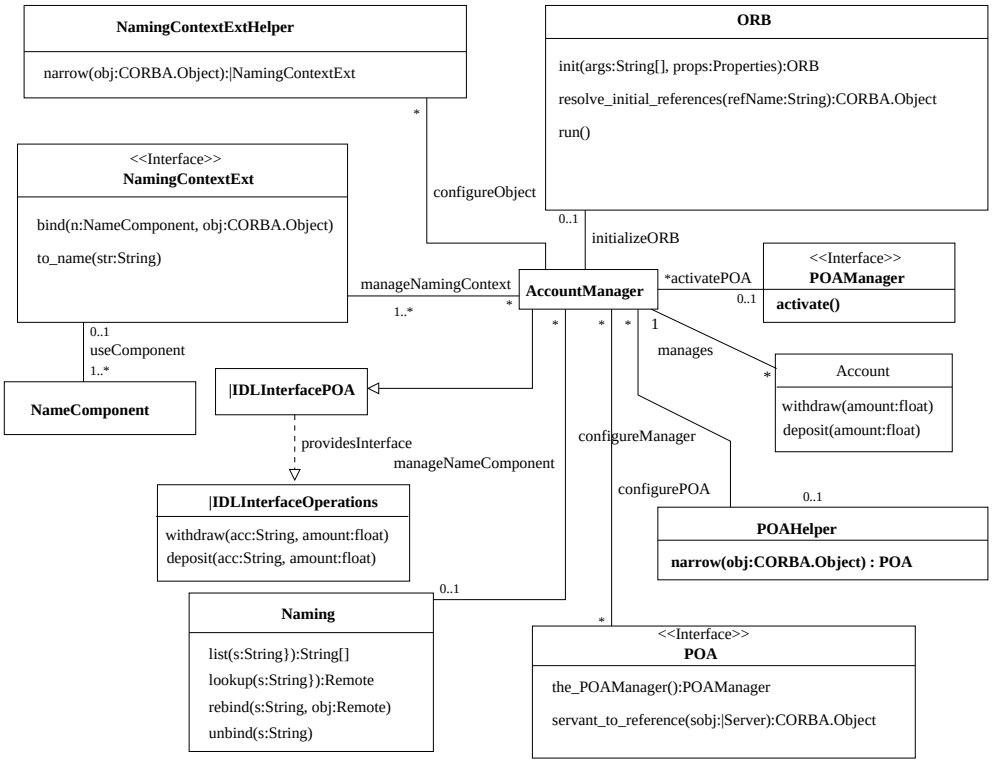

Figure 5.4: CORBA Distribution Target Model.

5.2 Pilot Study 2: Transforming Distribution Class Model to Jini Class Model

This section uses the model-to-model transformation technique to transform the platform independent source model shown in Figure 5.3 into a Jini distribution class model. The source pattern for the transformation is shown in Figure 5.1 and the binding specification for the source model is shown in Table 5.1.

5.2.1 Specify Model Transformations

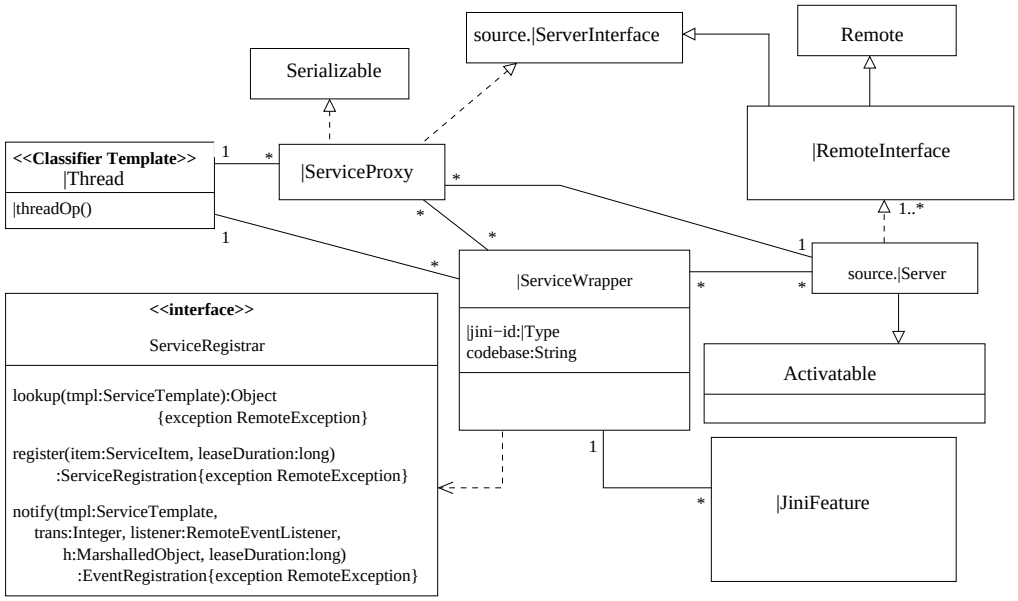

Figure 5.5: Class Transformation Schema.

The mappings to transform the generic distribution class model into a Jini distribution model is specified as a transformation schema as shown in Figure 5.5. The transformation schema has the following features:

1. The *Thread* classifier template represents a protocol for managing threads. The Thread classifier template can be instantiated using classes or interfaces.

2. The *ServiceProxy* class template is a proxy for the `Server` class template. A proxy is an entity that acts on behalf of another entity. Client programs interact with the Server transparently through the ServiceProxy. This shields the Server from direct manipulation and allows services to be added to the Jini Lookup Service, or removed from the Jini Lookup Service transparently. The federated Jini service is a combination of Server and ServiceProxy. They can be combined in three distinct ways. (1) The Proxy and the Server can share the service functionality, or (2) the complete functionality of the service can be implemented in the ServiceProxy or (3) The complete functionality of the service can be implemented in the Server. However, if either of the first two options are used then the proxy may have to be changed each time the service is updated. In order to make service proxies downlaodable by clients, Jini requires that service proxies be serializable. This is accomplished by causing the service proxy to provide the `Serializable` class template which facilitates object serialization.

3. The `Activatable` class template facilitate persistent access to the Server with activation on demand.

5.2.2 Process Transformation Directives

The transformation schema contains a single set of connected model elements and therefore the loop in the `processAllConnectedComponents` operation in the class model transformation algorithm will be executed only once. A new component is created prior to the `while` loop in `processConnectedComponent`. Model elements in the transformation schema are processed and added to this new component as described below.

1. |`ServiceWrapper` results in a new class template with the name |`ServiceWrapper` being created. The |`jini-id:`|`Type` and `codebase:String` directives result in new attribute templatess being created and inserted into the new class. The new class is then add to the new component.

2. The transformation schema association between |`ServiceProxy` and |`ServiceWrapper` is the next item processed. Processing results in a new association being created and inserted into the new component.

3. |`ServiceProxy` results in a new class with the name |`ServiceProxy` being created and added to the new component.

4. The transformation schema realization dependency between |`ServiceProxy` and |`Serializable` is the next item processed. Processing results in a new realizaion dependency being created and inserted into the new component.

5. `Serializable` results in a new class with the name `Serializable` being created and added to the new component.

6. The transformation schema association between |`ServiceProxy` and `source.`|`Server` is the next item processed. Processing results in a new association being created and inserted into the new component.

7. The `source.`|`Server` transformation schema class results in the `AccountManager` class being copied from the source model to the new component.

8. The transformation schema generalization between `source.`|`Server` and `Activatable` is the next item processed. Processing results in a new generalization being created and inserted into the new component.

9. `Activatable` results in a new class with the name `Activatable` being created and added to the new component.

10. The transformation schema realization dependency between `source.|Server` and `|RemoteInterface` is the next item processed. Processing results in a new realization dependency being created and inserted into the new component.

11. `|RemoteInterface` results in a new interface template with the name `|RemoteInterface` being created and added to the new component.

12. The transformation schema generalization between `|RemoteInterface` and `Remote` is the next item processed. Processing results in a new generalization being created and inserted into the new component.

13. `Remote` results in a new class with the name `Remote` being created and added to the new component.

14. The transformation schema generalization between `|RemoteInterface` and `source.|ServiceInterface` is the next item processed. Processing results in a new generalization being created and inserted into the new component.

15. The `source.|ServiceInterface` transformation schema interface results in the `AccountOperations` being copied from the source model to the new component.

16. The transformation schema association between `|ServiceWrapper` and `|Thread` is the next item processed. Processing results in a new association being created and inserted into the new component.

17. `|Thread` results in a new classifier template with the name `|Thread` being created. The `threadOp()` directive results in a new operation template being

created and inserted into the new classifier template. The new classifier template is then added to the new component.

18. The transformation schema association between |`ServiceWrapper` and |`Jinifeature` is the next item processed. Processing results in a new association being created and inserted into the new component.

19. The |`Jinifeature` transformation schema classifier results in a new classifier template with the name |`Jinifeature` being created. The |`initialize(|params1*)`, |`registerServer(|ser:|Server)` and |`configureServer(|ser:|Server, |params3*)` operation templates are copied into the new classifier template. The new classifier template is then added to the new component.

20. The transformation schema usage dependency between |`ServiceWrapper` and `ServiceRegistrar` is the next item processed. Processing results in a new usage dependency being created and inserted into the new component.

21. The `ServiceRegistrar` transformation schema interface is processed resulting in a new interface template with the name `ServiceRegistrar` being created. The embedded directives are processed as follows:

    - The `lookup(impl:ServiceTemplate) :Object{exception RemoteException}` directive results in the operation: `lookup(impl:ServiceTemplate):Object {exception RemoteException}` being created and added to the new interface.
    - The `register (item:ServiceItem, leaseDuration:long): ServerRegistration {exception RemoteException}` directive results in the operation: `register(item:ServiceItem,`

`leaseDuration:long):ServerRegistration {exception RemoteException}` being created and added to the new interface.

- The `notify(impl:ServiceTemplate, listener: RemoteEventListener, h:MarshalledObject, leaseDuration:long): ServerRegistration {exception RemoteException}` directive results in the operation: `notify(impl:ServiceTemplate, listener:RemoteEventListener, h:MarshalledObject, leaseDuration:long):ServerRegistration {exception RemoteException}` being created and added to the new interface. The new interface is added to the new component.

The `Account` class and the `manages` association are not bound to any source pattern model elements. These items are copied to the new component and the new component is added to the target model. The resulting Jini distribution class model is shown in Figure 5.6.

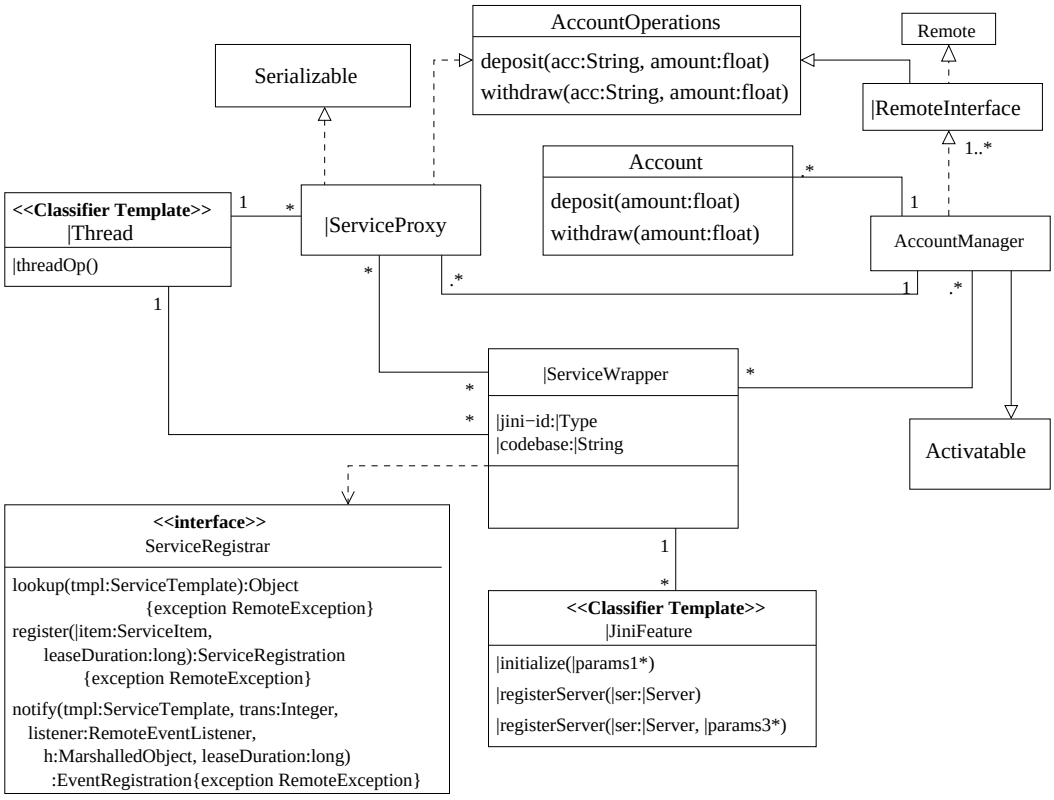

Figure 5.6: Jini Server Distribution Target Model.

5.3 Discussion

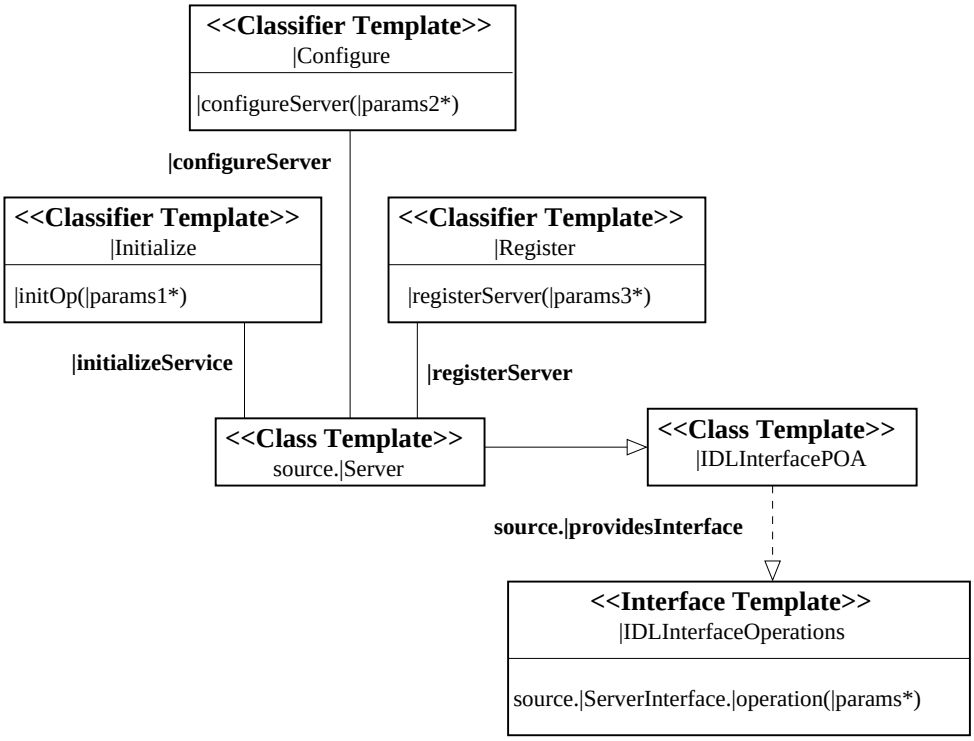

Figure 5.7: Target Class Pattern

In Chapter 4 the use of target patterns as transformation testing oracles was discussed. Figure 5.7 shows a target pattern for the model-to-model transformation of server distribution class models. The target pattern includes:

- The *source.|Server* class template represents the *|Server* class template of the source pattern.
- The *|IDLInterfacePOA* class template and the *|IDLInterfaceOperations* interface template represent model elements generated by the CORBA IDL compiler to support the distribution of servers. The *|IDLInterfaceOperations* interface template is created using operation templates defined in the

|*ServerInterface* interface template in the source pattern. This is specified using the *source.|ServerInterface.|operation(|params*)* directive.

- The middleware distribution service is represented by the *Initialize*, *Register* and *Configure* Classifier templates.

- The *Initialize* Classifier template represents the protocol for initializing the middleware infrastructure that supports distribution. Instances of the *initOp* operation template are used to prepare the middleware features that support distribution, for example, initialization of the CORBA ORB.

- The *Register* Classifier template represents the protocol for registering the server with the middleware distribution service. The *Server* is made available to clients by registering the server with the middleware using instances of the *registerServer* operation template.

- After a service has registered, it may be necessary to interact with the middleware service. For example to change a property associated with the server or to reset a property associated with the distribution service. This is accomplished using instances of the *configureServer* operation template of the *Configure* Classifier template.

The process of establishing conformance of the target model to this target pattern involves determining that there is a target model element that plays the role (or conforms) of each model element in the target pattern.

A binding specification for this target pattern and the CORBA target model shown in Figure 5.4 is shown in Table 5.1 and Table 5.2. The binding specification would be used in determining the conformance of the target model to the target pattern.

Table 5.2: Target Binding Specification.

Target Pattern Element	**Target Model Element**
|IDLInterfacePOA	|IDLInterfacePOA
|IDLInterfaceOperations	|IDLInterfaceOperations
|Initialize	ORB
|Initialize::|initOp	ORB::init
|Initialize::|initOp:: |params1*	ORB:: init::args:String[]
|Initialize::|initOp:: |params1*	ORB:: init::props:Properties
|Initialize::|initOp	ORB::run
|Register	POAManager
|Register::|registerServer	POAManager::activate
|Register	NamingContextExt
|Register::|registerServer	NamingContextExt::bind
|Register::|registerServer::|params3*	NamingContextExt:: bind::n:NameComponent
|Register::|registerServer::|params3*	NamingContextExt:: bind::obj:CORBA.Object
|Configure	Naming
|Configure::|configureServer	Naming::|list
|Configure::|configureServer::|params2*	Naming::|list::s:String
|Configure::|configureServer	Naming::|lookup
|Configure::|configureServer::|params2*	Naming::|lookup::s:String
|Configure::|configureServer	Naming::|unbind
|Configure::|configureServer::|params2*	Naming::|unbind::s:String
|Configure::|configureServer::|params2*	Naming::|unbind::obj:Remote
|Configure::|configureServer	Naming::|rebind
|Configure::|configureServer::|params2*	Naming::|rebind::s:String
|initializeService	initializeORB
|configureServer	manageNameComponent
|registerServer	registerServer

In general, items in the target pattern can be grouped into three sets. One set represents the primary business logic of the application, one set represents platform-specific items and the third set is the relationships defined between the first two sets of model elements. In this example, `source.|Server` and `source.|ServiceInterface.|operation(|params*)` represent the primary

business logic of the money transfer service application. |`Configure`, |`initialize`, |`Register`, |`IDLInterfacePOA` and |`IDLInterfaceOperations` represent CORBA-specific items.

Chapter 6

Pilot Studies: Transforming Distributed Transaction Models

This chapter presents two additional pilot studies to illustrate the application of the transformation algorithm to a platform independent transaction class model. The first pilot study uses the transformation language to transform a platform independent transaction class model into a CORBA transaction class model. The second pilot study uses the transformation language to transform the platform independent transaction class model into a Jini transaction class model. The transformation context is illustrated by the activity diagram shown in Figure 4.1.

6.1 Pilot Study 3: Transforming Transaction Class Model to CORBA Class Model

This section illustrates the transformation of a platform-independent server transaction class model into a CORBA transaction class model.

6.1.1 Overview of CORBA Transaction Service

A transaction is an indivisible collection of operations between servers and clients that remain atomic even if some clients and servers fail. An atomic operation is an operation that is free of interference from concurrent operations performed by

other threads in a system. Transactions are required to manifest the 'ACID' properties [1]. While different middleware may provide different transaction models, a generic transaction model that captures the essence of distributed transactions can be specified at the PIM level. The generic model can then be transformed based on the specific protocols for each middleware.

CORBA provides a transaction service that supports five types of application objects: *transactional clients, transactional objects, transactional servers, recoverable objects* and *recoverable servers*.

A transactional client is an application that creates a transaction and invokes operations on one or more transactional objects. An object that has some behavior that is invoked in the scope of a transaction is called a transactional object. A transactional server contains one of more transactional objects. Not all transactional objects need support transactional behavior. To support transactional behavior, an object must participate in the CORBA transaction protocols. Typically, objects participate in these protocols only if it manages persistent data. Transactional objects that directly manage persistent data are called recoverable objects. Recoverable objects participate in the transaction completion protocol by creating and registering resource objects with the transaction service. A resource object is one that implements the CORBA *Resource* interface. The transaction service manages the two phase commit protocol for all registered resources. A recoverable server contains one or more recoverable objects.

The CORBA transaction service provides interfaces to support distributed transactions. These interfaces include: *Current, Control, TransactionFactory, Coordinator, RecoveryCoordinator, Terminator* and *Resource*. The Current interface provides the primary protocol through which clients and servers interact with the transaction service. New transactions can be created using either the

Current interface or the TransactionFactory interface. Clients and servers can explicit propagate a transaction context using the Control interface. The Coordinator, RecoveryCoordinator and Terminator interfaces are used by participants in a transaction to manage the transaction. The RecoveryCoordinator is a coordinator that is used to manage the transaction recovery process while a Terminator object is used to end transactions.

The CORBA Object Request Broker (ORB) maintains a transaction context for each ORB-aware thread. The transaction context associated with an ORB-aware thread is null if there is no associated transaction, or it refers to a specific transaction. The transaction context is passed implicitly to each object that implements the *Resource* interface, however the transaction context may also be passed explicitly to transactional objects as a parameter of a transactional operation.

6.1.2 Source Class Pattern

The source class pattern for the generic distributed transactions is shown in Figure 6.1. The diagram has two class templates (`TransClient` and `Participant`, and one interface template (`ParticipantInterface`).

- The `TransClient` class template represents the set of classes that initiate the transaction. The TransClient performs one or more operations for the transaction.
- `Participant` represents the set of classes that provide some service required by transaction clients. These services are represented by the `ServiceInterface`.
- `ServiceInterface` represents a service provided by a participants in a transaction.

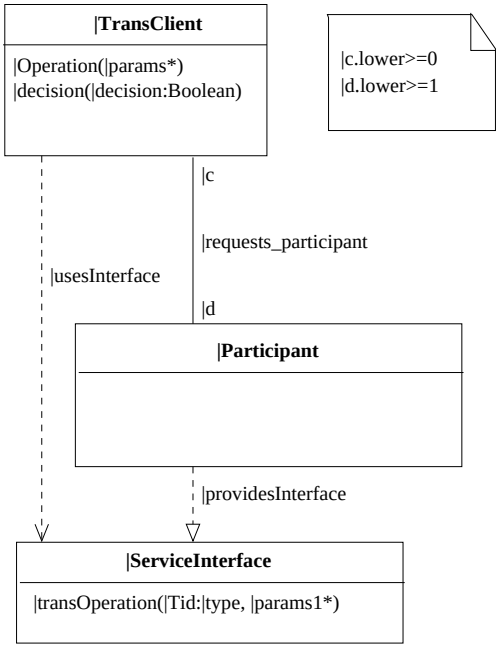

Figure 6.1: Source Class Pattern for Distributed Transactions.

`TransClient` has an operation template named `Operation` representing operations that initiate transactional behavior. The `decision` operation represents functionality that records the outcome of a transaction. A transaction may be either committed or aborted. The `processCommit` and `processAbort` operations are carried out by the `TransClient` when a transaction is committed or aborted respectively.

`ServiceInterface` has an operation template named `transOperation` representing the `Participant` operations that are transactional. The instantiation of the `Operation` template makes the instantiations of `transOperation` transactional. For every instance of `transOperation` there should be a corresponding instantiation of `do_|transOperation`. For example, if the `transOperation` template in `Participant` is instantiated with a name *op1*, then `do_|transOperation` must be instantiated as *do_op1*. The `processCommit` and `processAbort` oper-

ations are carried out by the `Participant` when a transaction is committed or aborted respectively.

6.1.3 Specify Model Transformations

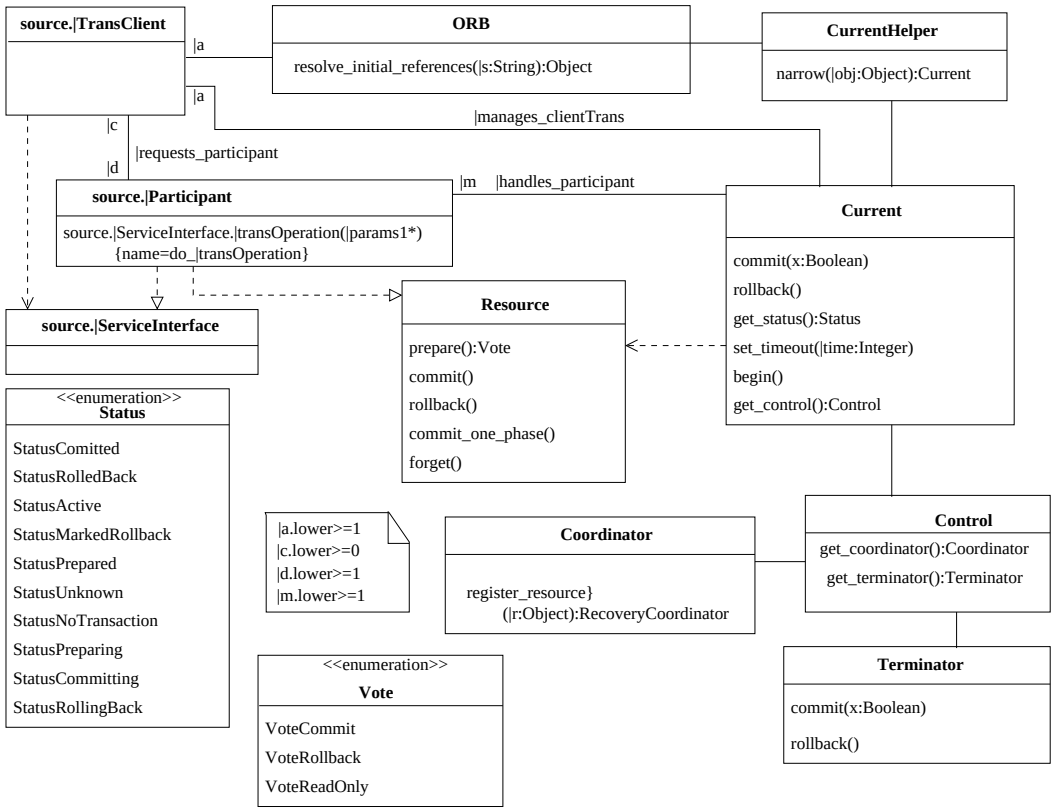

Figure 6.2: CORBA Class Diagram Transaction Transformation Schema.

The mappings to transform the source transaction class model into a target CORBA class model is specified as a transformation schema as shown in Figure 6.2. The transformation schema has the features described below:

1. CORBA requires all participants in a transaction to implement its `Resource` interface. This is represented as a transformation schema realization dependency between the `source.`|`Participant` transformation schema class and the `Resource` transformation schema interface.

2. CORBA provides a number of features to collectively perform the role of a transaction manager. These features are represented in the transformation schema by: `ORB, CurrentHelper, Current, Control, Coordinator` and `Terminator`.

3. `Source.|TransClient`, `source.|Participant` and `source.|ServiceInterface` represent model elements copied from the source model.

6.1.4 Source Class Model and Binding Specification

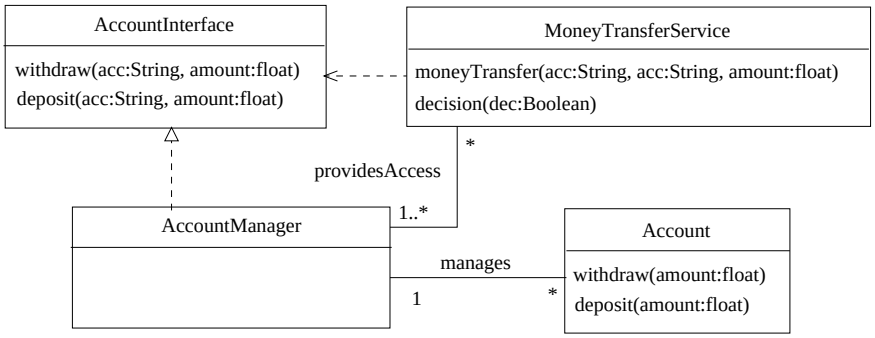

Figure 6.3: A Money Transfer Service Class Diagram.

Figure 6.3 shows a source class model of a money transfer service application. The model consists of three classes: `AccountManager, MoneyTransferService` and `Account`, and one interface: `AccountInterface`. Accounts are managed by `AccountManager` objects. The `AccountManager` class provides `AccountInterface` while the `Account` class has `withdraw` and `deposit` operations for the withdrawal of money from client accounts and deposit of money into client accounts respectively. The `MoneyTransferService` class has the `moneyTransfer` operation for transferring money between two accounts and the `lookup` operation for locating account manager objects. The binding specification for this source model is shown

in Table 6.1.

Table 6.1: Bindings for Money Transfer Transaction Source Model.

Aspect element name	**Model element name**
|TransClient	MoneyTransferService
|TransClient::|Operation	MoneyTransferService::moneyTranfer
|Operation::|params*	moneyTranfer::acc1:String, acc2:String, amount:Integer
|TransClient::|decision	MoneyTransferService::decision
|decision::|decision	moneyTranfer::decision:dec
|Participant	AccountManager
|do_transOperation::Tid:|type, |params1*	do_withdraw::Tid:String,acc:String, amount:float
|do_transOperation::Tid:|type, |params1*	do_deposit::Tid:String,acc:String, amount:float
|type	String
|ServiceInterface	AccountInterface
|ServiceInterface::|transOperation	AccountInterface::withdraw
|transOperation::Tid:|type, |params1*	withdraw::Tid:String,acc:String, amount:float
|ServiceInterface::|transOperation	AccountInterface::deposit
|transOperation::Tid:|type, |params1*	deposit::Tid:String,acc:String, amount:float
|requests_participant	requests_participant
|c	*
|d	1..*

6.1.5 Process Transformation Directives

The transformation schema has three connected components. The first includes the `source.|TransClient` transformation schema class and model elements connected to it. The other two are the enumeration templates: `Status` and `Vote`. Since there are three connected components, the loop in the `processAllConnectedComponents` operation in the class model transformation algorithm will be executed three times. A new component is created for each

iteration of the `while` loop and model elements in one component are processed and added to this new component. The composite model elements in the first connected component are processed as follows:

1. The `source.|TransClient` is the first composite transformation schema model element processed. The `|TransClient` class template is bound to the `MoneyTransferService` class, therefore, the `source.|TransClient` directive results in the `MoneyTransferService` class being copied and added to the new component.

2. The `ORB` transformation schema class is processed. Processing results in a new class with the name `ORB` being created. The `resolve_initial_references(s:String):Object)` transformation schema operation results in a new operation being created and added to the new class. The new class is then inserted into the new component.

3. The `CurrentHelper` transformation schema class is processed. Processing results in a new class with the name `CurrentHelper` being created. The `narrow(obj:Object):Current` transformation schema operation results in a new operation being created and added to the new class. The new class is then inserted into the new component.

4. The `Current` transformation schema class is processed resulting in a new class with the name `Current` being created. The transformation schema operations in `Current` result in new operations being created and added to the new class. The new class is then inserted into the new component.

5. The `Control` transformation schema class is the next item processed. This item results in a new class with the name `Control` being created. Two new operations: `get_coordinator():Coordinator` and

`get_terminator():Terminator` are created and added to the new class. The class is then inserted into the new component.

6. The `Coordinator` transformation schema class is the next item processed. This item results in a new class with the name `Coordinator` being created. A new operation: `register_resource (|r:Object):RecoveryCoordinator` is created and added to the new class and the class is inserted into the new component.

7. The `Terminator` transformation schema class is the next item processed. This item results in a new class with the name `Terminator` being created. Two new operations: `commit(x:Boolean)` and `rollback()` are created and added to the new `Terminator` class. The class is then inserted into the new component.

8. The `source.|Participant` is the next composite transformation schema model element processed. Processing results in the `AccountManager` class from the source model being copied and added to the new component. The `deposit` and `withdraw` operations in AccountInterface are bound to the `|transOperation` operation template in `|ServiceInterface`. As a result, the `source.|ServiceInterface.|transOperation(|Tid:|type, |params1*){name=do_|transOperation}` directive results in two new operations: `do_deposit(acc:String, amount:float)` and `do_withdraw(acc:String, amount:float)` being added to the new `AccountManager` class. The new class is then inserted into the new component.

9. The `Resource` transformation schema interface is processed resulting in a new interface with the name `Resource` being created. Five new operations:

`prepare():Vote`, `commit()`, `rollback()`, `commit_one_phase()` and `forget` are created and inserted into the new `Resource` interface. The interface is then inserted into the new component.

10. The `source.|ServiceInterface` is the next composite transformation schema model element processed. Processing results in the `AccountInterface` from the source model being copied and added to the new component.

The first component is added to the target model and the other two connected components are processed as follows:

1. The `Status` transformation schema enumeration results in an enumeration being created with the name `Status` and the following enumeration literals: `StatusCommitted, StatusRolledBack, StatusActive, StatusMarkedRollback, StatusPrepared, StstusUnknown, StatusNoTransaction, StatusPreparing, StatusCommiting` and `StatusRollingBack`. The enumeration is then adeed to a new component and the new component is added to the target model.

2. The `Vote` transformation schema enumeration results in an enumeration being created with the name `Vote` and the following enumeration literals: `VoteCommit, VoteRollback` and `VoteReadOnly`. The enumeration is then adeed to a new component and the new component is added to the target model.

All relationships in the transformation schema are also processed. Finally, the `Account` class and the `manages` association are copied to the target model. The CORBA transaction target class model that results when all the transformation directives are processed is shown in Figure 6.4.

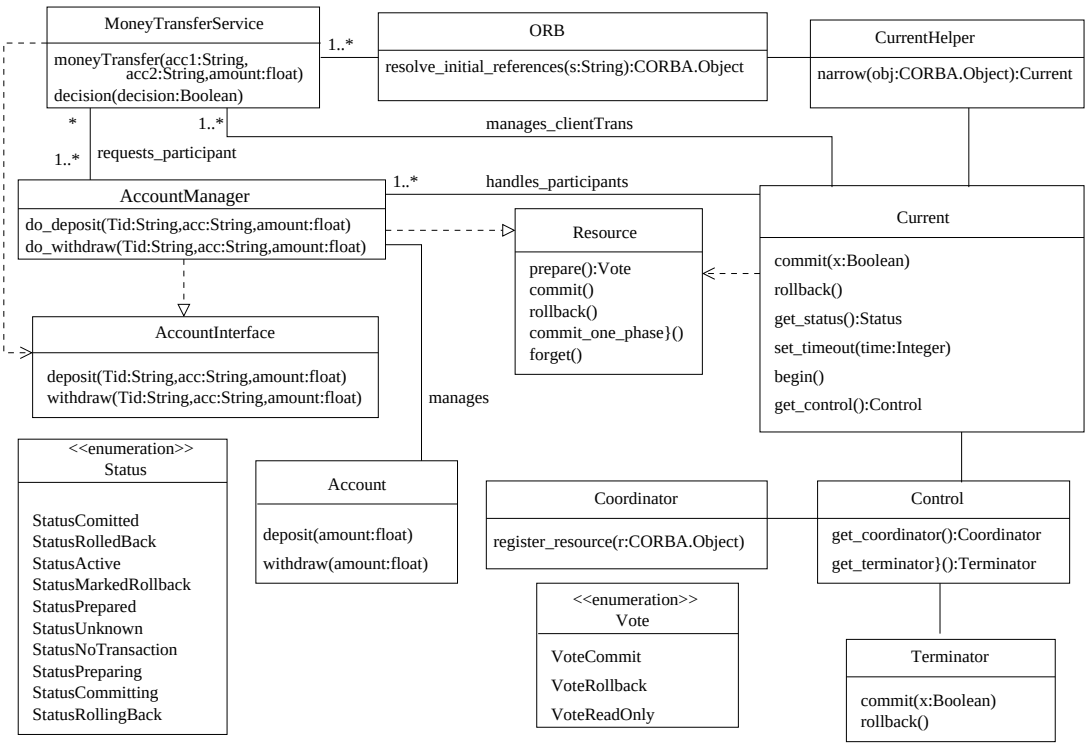

Figure 6.4: Target CORBA Transaction Class Model.

6.2 Pilot Study 4: Transforming Transaction Class Model to Jini Class Model

This section uses the model-to-model transformation technique to transform the platform independent source model shown in Figure 6.3 into a Jini transaction class model. The source pattern for the transformation is shown in Figure 6.1 and the binding specification for the source model is shown in Table 6.1.

6.2.1 Overview of Jini Transaction Service

In Jini, a distributed transaction model has three main parts: a `Transaction Manager`, `Transactional Clients` and `Participants`. The Jini transaction manager is specified as a `TransactionManager` interface. The interface supports distributed transactions and manages the two-phase commit protocol for all par-

ticipats that have joined a transaction. The interface has the following operations:

- The operations `void abort(long id)` and `void abort(long id, long waitFor)`, are used by clients and participants to request that a transaction be aborted. The second operation waits for all participants to be notified of the decision. The `TimeoutExpiredException` is thrown if the transaction manager is unable to notify all participants of the decision before the specified `waitFor` timeout expires.

- The `void commit(long id)` and `void commit(long id, long waitFor)` operations are used by clients and participants to request that a transaction be committed. The second operation waits for all participants to be notified of the decision. The `TimeoutExpiredException` is thrown if the transaction manager is unable to notify all participants of the decision before the specified `waitFor` timeout expires. However, in cases where the `waitFor` timeout expires before the transaction manager reaches a decision, the `TimeoutExpiredException` is not thrown until the transaction manager reaches a decision.

- The `TransactionManager.Created create(long lease)` operation is used to create a new top-level transaction. Jini also provides a TransactionFactory interface for creating transactions.

- The `getState(long id)` operation returns the current state of the given transaction. The value returned can be any one of `ABORTED, ACTIVE, COMMITTED, NOTCHANGED, PREPARED` or `VOTING`. These returned values are of integer type.

- Clients and participants use the `join(long id, TransactionParticipant part, long crashCount)` operation to inform the `TransactionManager` of

their desire to become participants in a transaction, i.e to 'join' the transaction.

A `transactional client` is an application that creates a transaction and invokes operations on one or more services participating in the transaction. A transactional client initiates a transaction by asking the Jini *Lookup service* if a transaction manager is available. If one is available, the client creates a object reference to the transaction manager and a new transaction by invoking the `create` operation of the `TransactionManager` or `TransactionFactory` class. The `create` operation returns a `TransactionManager.Created` object consisting of an identifier for the transaction `tid` and a `Lease` object. The client then joins the transaction by invoking the `TransactionManager`'s `join()` operation, passing the transaction identifier as an argument. Typically, the client will then perform one or more transactional operations associated with one or more transaction participants.

Objects become `participants` in a transaction by implementing the Jini `TransactionParticipant` interface and joining the transaction. The TransactionParticipant interface has four operations: `prepare, commit, abort` and `prepareAndCommit`. Each operation has the same signature *operation(TransactionManager mgr, Long transactionID)*. The `prepare, commit` and `abort` operations are used by the transaction manager to determine, if a participant is ready to commit resources, to inform a participant that it should commit its resources and to inform a participant that it should abort its resources respectively. When all but one participant have been asked to prepare its resources, the `prepareAndCommit` operation is used to inform the last participant to both prepare and commit its resources. A participant is either a client or a `proxy` for a service. A service proxy is an application that known the location of the actual

service. Clients invoke operations on the proxy which in tern directs all requests to the actual service.

6.2.2 Specify Model Transformations

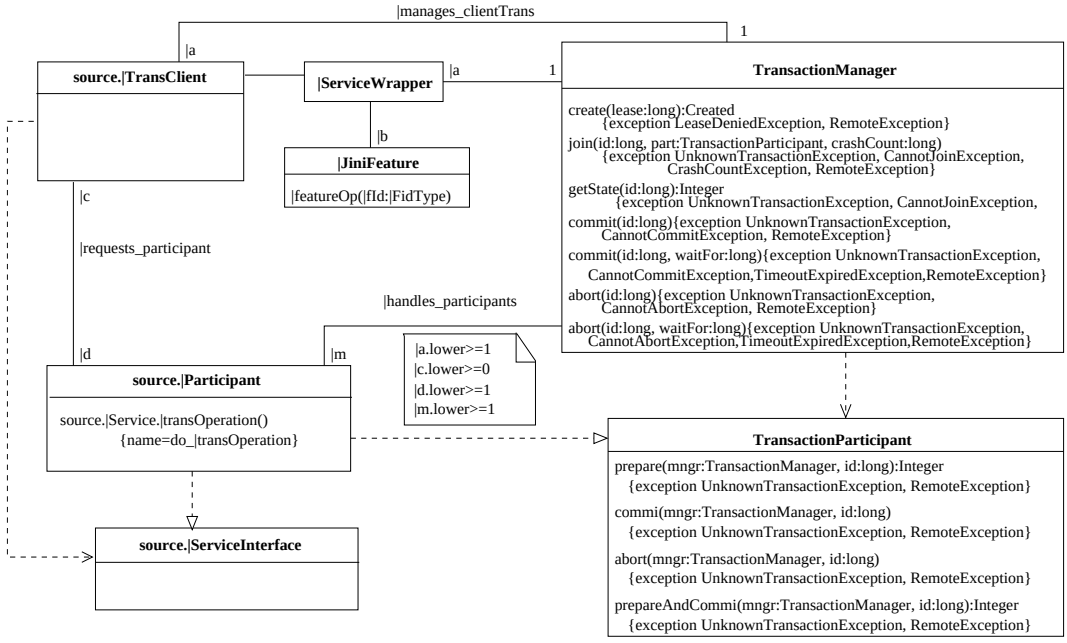

Figure 6.5: Jini Class Diagram Transaction Transformation Schema.

The mappings to transform the source transaction aspect class model into a Jini model is specified as a transformation schema as shown in Figure 6.5. The `|JiniFeature` transformation schema class represents Jini features other than the Jini transaction manager, that are used to support distributed transactions, for example features of the Jini event service.

6.2.3 Process Transformation Directives

The transformation schema has a single connected component. The composite model elements in the connected component are processed as follows:

- The `source.|TransClient` transformation schema class is the first composite transformation schema model element processed. The `|TransClient` class template is bound to the `MoneyTransferService` class. The `source.|TransClient` directive is processed resulting in the `MoneyTransferService` class being copied and added to the new component.

- The `|ServiceWrapper` transformation schema class is the next item processed. This item results in a new class with the name `|ServiceWrapper` being created and inserted into the new component.

- The `|JiniFeature` transformation schema class is the next item processed. This item results in a new class template with the name `|JiniFeature` being created. The `|featureOp(|id:|IdType)` directive results in a new operation template being created and inserted into the new |JiniFeature class template. The new class template is inserted into the new component.

- The `source.|Participant` is the next composite transformation schema model element processed. Processing results in the `AccountManager` class from the source model being copied and added to the new component. The `deposit` and `withdraw` operations in AccountInterface are bound to the `|transOperation` operation template in `|ServiceInterface`. As a result, the `source.|ServiceInterface.|transOperation(|Tid:|type, |params1*){name=do_|transOperation}` directive results in two new operations: `do_deposit(acc:String, amount:float)` and `do_withdraw(acc:String, amount:float)` being added to the new `AccountManager` class. The new class is then inserted into the new component.

- The `TransactionParticipant` transformation schema interface is processed resulting in a new interface with the name `TransactionParticipant` being created. The new directives in `TransactionParticipant` results in the following operations being added to the new interface:

  - `prepare}(mngr:TransactionManager, id:long):Integer{ exception UnknownTransactionException, RemoteExcpetion}`.
  - `commit(mngr:TransactionManager, id:long){exception UnknownTransactionException, RemoteExcpetion}`.
  - `abort(mngr:TransactionManager, id:long){exception UnknownTransactionException, RemoteExcpetion}`.
  - `prepareAndCommit(mngr:TransactionManager, id:long){exception UnknownTransactionException, RemoteExcpetion}`.

- The `TransactionManager` transformation schema class is processed resulting in a new class with the name `TransactionManager` being created. The new directives in `TransactionManager` results in the following operations being added to the new class:

  - `create(lease:long):Created{exception LeaseDeniedException, RemoteExcpetion}`. The operation is inserted into the newly created `TransactionManager` class.
  - `join(lease:long, part:TransactionParticipant, crashCount:long):Created{exception UnknownTransactionException, CannotJoinException, CrashCountException, RemoteExcpetion}`.

- `getState(id:long):Integer{exception UnknownTransactionException, RemoteExcpetion}`.
- `commit(id:long):Integer{exception UnknownTransactionException, CannotCommitException, RemoteExcpetion}`.
- `commit(id:long, waitfor:long):Integer{exception UnknownTransactionException, CannotCommitException, TimeoutExpiredException, RemoteExcpetion}`.
- `abort(id:long):Integer{exception UnknownTransactionException, CannotAbortException, RemoteExcpetion}`.
- `abort(id:long, waitfor:long):Integer{exception UnknownTransactionException, CannotAbortException, TimeoutExpiredException, RemoteExcpetion}`.

- The `source.|ServiceInterface` is the next composite transformation schema model element processed. Processing results in the `AccountInterface` from the source model being copied and added to the new component.

All relationships in the transformation schema are also processed and the `Account` class and the `manages` association are copied to the target model. The resulting target Jini transaction class model is shown in Figure 6.6.

6.3 Discussion

A target class pattern for the model-to-model transformation of distributed transaction class models is shown in Figure 6.7. The target pattern includes the fol-

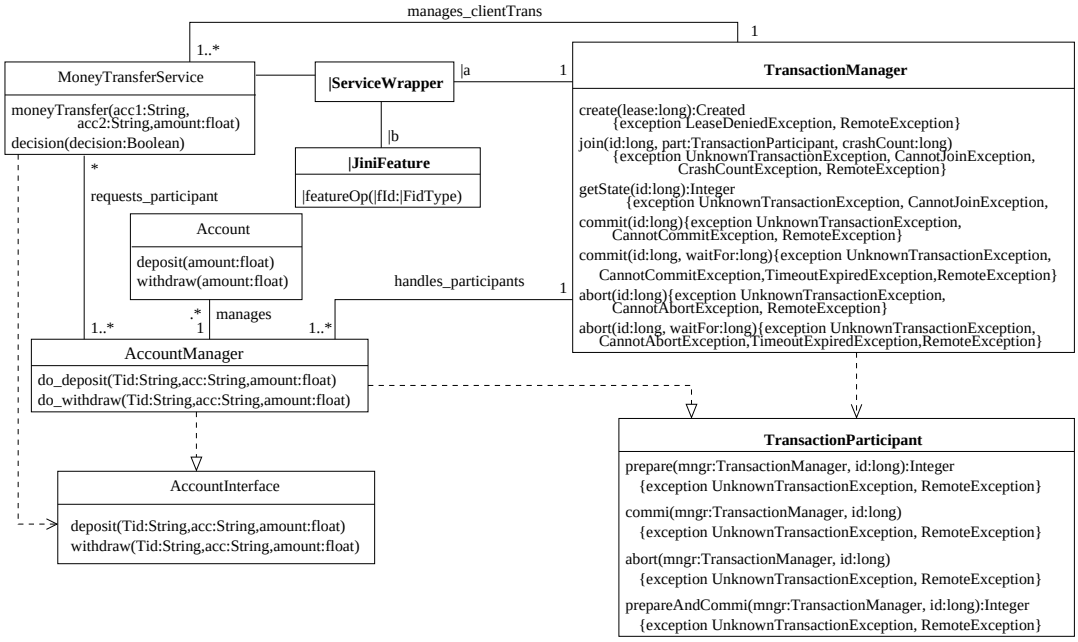

Figure 6.6: Target Jini Transaction Class Model.

lowing model elements:

- The `source.|Transclient, source.|Participant` and `source.|ServiceInterface` represent model elements defined in the source pattern.

- `|TransactionManager` represents a protocol for managing distributed transactions. The `|create`, `|join`, `|commit`, and `|abort` operation templates represent operations used for creating a transaction, registering to become a participant in a transaction, committing a transaction and aborting a transaction respectively. The `|getTransState` operation template returns the state of a transaction as specified by the `|State` enumeration template. A transaction may be in a number of different states for example, a transaction may be in a `|COMMITTED` or `|ABORTED` state which indicate that the transaction has been committed or aborted respectively.

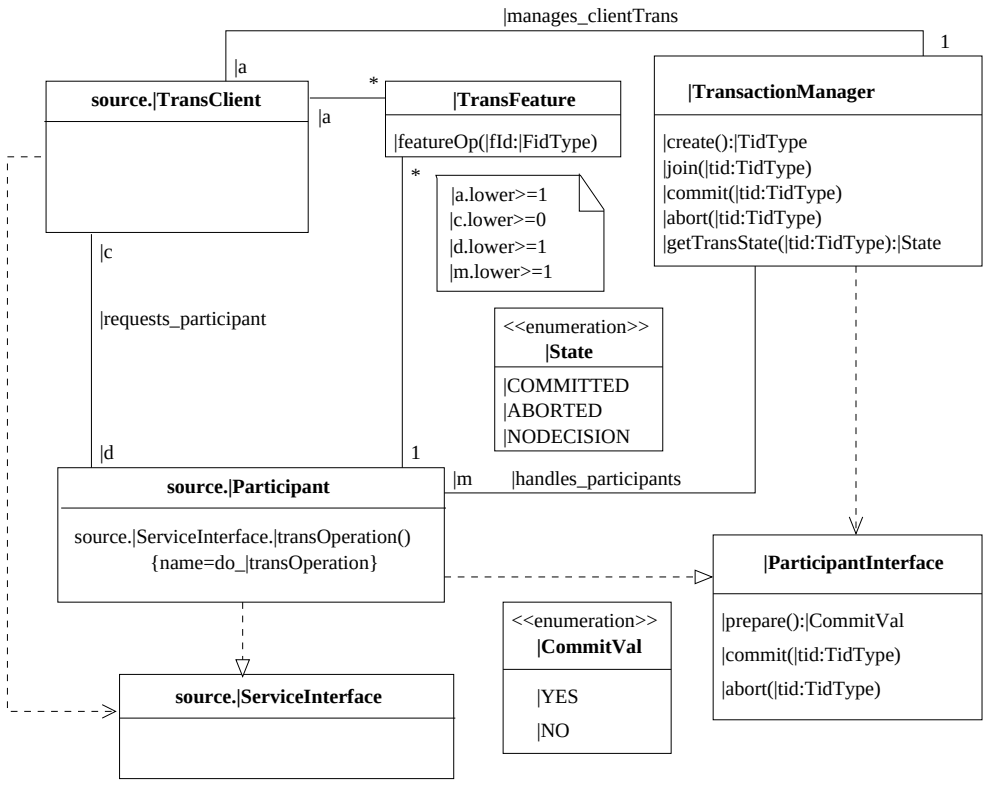

Figure 6.7: Target Pattern for Distributed Transactions.

- `ParticipantInterface` represents the protocol through which a transaction manager communicates with Participant instances.

- The `CommitVal` enumeration template represents values returned by instances of the `canCommit` operation template to indicate the decision of a participant in a transaction to commit ('YES') or abort ('NO') a transaction.

- `|TransFeature` represents any other platform-specific transactional feature other than a transaction manager or a participant interface.

The process of establishing conformance of the target model to this target pattern involves determining that there is a target model element that plays the

role (or conforms) of each model element in the target pattern.

Items in the target pattern can be grouped into sets. `TransClient, Participant, ServiceInterface` and `|request_participant` represent the business logic of the application. In contrast, `|TransactionManager, |ParticipantInterface, |TransFeature, |CommitVal` and `|State` represent platform-specific items. Model elements in the CORBA target model shown in Figure 6.4 are bound to target pattern model elements as follows:

- `TransClient, Participant, ServiceInterface` and `|request_participant` are bound as described in Table 6.1.
- The other model elements are bound as shown in Table 6.2.

Model elements in the Jini target model shown in Figure 6.6 are bound to target pattern model elements as follows:

- `TransClient, Participant, ServiceInterface` and `|request_participant` are bound as described in Table 6.1.
- The other model elements are bound as shown in Table 6.3.

The binding specifications would be used in determining the conformance of the target model to the target pattern.

Table 6.2: Target CORBA Binding Specification.

Target Pattern Element	**Target Model Element**
|**TransactionManager**	Current, CurrentHelper, Control, Coordinator, Terminator
|**TransactionManager::**|**create**	CurrentHelper::narrow
|**TransactionManager::**|**join**	Coordinator::register_resource
|**TransactionManager::**|**commit**	Current::commit
|**TransactionManager::**|**commit**	Terminator::commit
|**TransactionManager::**|**abort**	Current::rollback
|**TransactionManager::**|**abort**	Terminator::rollback
|**TransactionManager::**|**getTransState**	Current::get_status
|**ParticipantInterface**	Resource
|**ParticipantInterface::**|**prepare**	Resource::prepare
|**ParticipantInterface::**|**commit**	Resource::commit
|**ParticipantInterface::**|**abort**	Resource::rollback
|**State**	Status
|**State::**|**COMMITTED**	Status::StatusCommitted
|**State::**|**ABORTED**	Status::StatusRolledBack
|**State::**|**NODECISION**	Status::StatusUnknown
|**CommitVal**	Vote
|**CommitVal::**|**YES**	Vote::VoteCommit
|**CommitVal::**|**NO**	Vote::VoteRollback
|**manages_clientTrans**	manages_clientTrans
|**handles_participants**	handles_participants
|**TransFeature**	ORB
|**TransFeature::**|**featureOp**	ORB::resolve_initial_references
|a	1..*
|m	1..*

Table 6.3: Target Jini Binding Specification.

Target Pattern Element	**Target Model Element**
|**TransactionManager**	TransactionManager
|**TransactionManager::**|**create**	TransactionManager::create
|**TransactionManager::**|**join**	TransactionManager::join
|**TransactionManager::**|**commit**	TransactionManager::commit
|**TransactionManager::**|**abort**	TransactionManager::abort
|**TransactionManager::**|**getTransState**	TransactionManager::getState
|**ParticipantInterface**	Resource
|**ParticipantInterface::**|**prepare**	TransactionParticipant::prepare
|**ParticipantInterface::**|**commit**	TransactionParticipant::commit
|**ParticipantInterface::**|**abort**	TransactionParticipant::rollback
|**State**	Status
|**State::**|**COMMITTED**	Integer
|**State::**|**ABORTED**	Integer
|**State::**|**NODECISION**	Integer
|**CommitVal**	Integer
|**CommitVal::**|**YES**	Integer
|**CommitVal::**|**NO**	Integer
|**manages_clientTrans**	manages_clientTrans
|**handles_participants**	handles_participants
|a	1..*
|m	1..*

Chapter 7

Conclusion and Future Work

A technique for transforming class models has been presented. The technique includes a graphical model transformation language and a notation for specifying transformations. A transformation model is called a transformation schema. Transformation schemas contain imperative statements called directives that stipulate how a source model is obtained from a target model. A source model is transformed into a target model by processing the directives in the transformation schema using a transformation algorithm. A grammar for the language, a transformation algorithm and a transformation metamodel were developed to support class diagram transformation.

Specifying transformations at the level currently supported by QVT can be tedious because QVT transformation specifications are expressed as fine-grained object (an instance of a metamodel class) models. The new language leverages the UML class model syntax to specify transformations. This raises the level of abstraction at which transformations are specified above level of instances of metamodel classes. In the new language a transformation specification is a transformation schema that consists of one or more directives. Each transformation schema (and each directive) implicitly refers to one or more objects. For example, a transformation schema with one operation directive describes one object for the

associated class, one object for the associated operation, one object for the result type of the associated operation, and two objects for each associated parameter (one for the parameter, one for its type). As such, each transformation schema defines a relationship among a set of objects and is therefore at a higher level of abstraction than instances of metamodel classes.

The transformation technique was illustrated by transforming platform-independent transaction and distribution class models into CORBA and Jini transaction and distribution class models respectively.

7.1 Lessons Learned

The first two pilot studies (presented in Chapter 5) applied the transformation algorithm to the transformation of a platform independent distribution class model into a CORBA distribution class model and a Jini distribution class model. Separate transformation schemas were created for the CORBA transformation and the Jini transformation. The two pilot studies presented in Chapter 6 applied the transformation algorithm to the transformation of a platform independent transaction class model into a CORBA transaction class model and a Jini transaction class model. Separate transformation schemas were created for each of these transformations as well.

For all four pilot studies, a need for new directives was not encountered as the transformation directives currently defined were sufficient to specify the transformations.

While a need for new directives was not encountered, a number of scenarios arose that may require the extension of the syntax of one or more of the current transformation directives. These extensions are anticipated to provide modelers with alternate ways of specifying transformations. For example, consider

the following scenario: `a modeler wants to specify that a copy of each class bound to |ClassA should be present in the target model, with the attributes from the class eliminated`.

Using the current notation, the goal of the modeler is realized by: (1) copying the attributes and operations using a source directive in the name-directive compartment of a transformation schema class, and (2) eliminating the class attributes using one or more exclude directives in the attribute-directive compartment of the transformation schema class. One exclude directive is required for each attribute template in `|ClassA`.

Since all the class attributes are to be excluded, it may be convenient to extend the syntax of the exclude directive to allow a modeler the ability to use a single exclude directive to eliminate all the class attributes. This may be possible using a notation such as: *exclude All*, which would stipulate the exclusion of all class attributes associated with the transformation schema attribute-directive compartment in which the directive is specified. The use of extended forms of the current transformation directives is the subject of future research.

7.2 Future Work

The model transformation research presented in this dissertation may be extended in several ways.

1. Extending the transformation language to behavioral models such as sequence diagrams. Different diagram types (e.g. class diagrams, sequence diagrams) are used to describe different views of the same application and as such these views need to be consistent. Extending the research to include other diagram types such as sequence diagrams, should also include how the consistency of different diagram types can be effected.

2. Examining how target patterns may be generated from source patterns and transformation schemas.

3. Examining extensions to the current transformation directives.

4. Examining how transformation schemas may be generated from source and target patterns, for example, QVT source and target patterns.

5. Examining the use of aspect contracts for the run time validation of conditions and properties necessary for the proper application of a transformation.

6. Applying the transformation technique to other software features.

7. Defining mappings between QVT and new language.

The transformation language presented in this dissertation facilitates the specification of model transformations at a higher level of abstraction than object diagrams. More work, however, needs to be done to extend the research beyond UML class models and to create tool support to automate the model transformation process.

REFERENCES

[1] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design (3rd Ed.)*. International Computer Science Series. Addison-Wesley/Pearson Education, USA, 2001.

[2] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proc. Workshop on Generative Techniques in the Context of Model-Driven Architecture,OOPSLA'03*, Anaheim, California, USA, October 2003.

[3] eclipse.org. Eclipse - an open development platform. *URL* `http://eclipse.org/`.

[4] W. Keith Edwards. *Core Jini (2nd Ed.)*. Java Series. Prentice Hall, USA, 2001.

[5] Robert France, James M. Bieman, and Ray Trask. Extending the UML to Support Evolution Management. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM.01)*, Florence, ITALY, 2001.

[6] Robert France, Sudipto Ghosh, Eunjee Song, and Dae-Kyoo Kim. A Metamodeling Approach to Pattern-based Model Refactoring. *IEEE Software Special Issue on Model-Driven Development*, 20(5):52–58, September 2003.

[7] Robert France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, March 2004.

[8] Robert France, Dae-Kyoo Kim, and Eunjee Song. Patterns as precise characterizations of designs. Technical Report TR-02-101, Computer Science Department, Colorado State University, Fort Collins, Colorado, 2002.

[9] Robert France, Dae-Kyoo Kim, Eunjee Song, and S. Ghosh. Using roles to characterize model families. In *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics: Back to the Basics*, Tampa, Florida, USA, October 2001.

[10] Robert B. France and James M. Bieman. Multi-view software evolution: A UML-based framework for evolving object-oriented software. In *ICSM*, pages 386–, 2001.

[11] Jack Greenfield and Keith Short. *Models, Frameworks and Tools.* Wiley Publishing, Inc., Chapter 7: Generating Implementations, 2003.

[12] Reaz Hoque. *CORBA for real programmers.* IDG Books, USA, 1998.

[13] Frederic Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proc. Model Transformations in Practice Workshop at Models 2005*, Montego Bay, Jamaica, October 2005.

[14] Sheena Judson, Doris Carver, and Robert France. A MetaModeling Approach to Model Refactoring. In *submitted to UML 2003*, San Francisco, California, USA, 2003.

[15] Audris Kalnins, Janis Barzdins, and Edgars Celms. Basics of Model Transformation Language MOLA. In *Proc. Workshop on Model Driven Development (WMDD 2004) at ECOOP 2004*, Oslo, Norway, June 2004.

[16] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model Transformation Language MOLA. *Presentation made at Model-Driven Architecture: Foundations and Applications (MDAFA) 2004*, June 2004.

[17] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model Transformation Language MOLA. In *Proc. of Model-Driven Architecture: Foundations and Applications (MDAFA) 2004*, Linkoping, Norway, June 2004.

[18] Audris Kalnins, Edgars Celms, and Agris Sostaks. Model Transformation Approach Based on MOLA. In *Proc. Model Transformations in Practice Workshop at Models 2005*, Montego Bay, Jamaica, October 2005.

[19] Dae-Kyoo Kim. A Meta-Modeling Approach To Specifying Patterns, Ph.D. Dissertation, Department of Computer Science, Colorado State University. 2004.

[20] Dae-Kyoo Kim, Robert France, and Sudipto Ghosh. A UML-based language for specifying domain-specific patterns. *Journal of Visual Languages and Compuiting*, 15:265–289, January 2004.

[21] Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. A Role-Based Metamodeling Approach to Specifying Design Patterns. In *27th Annual International Computer Software and Applications Conference, COMPSAC 2003*, Dallas, USA, November 2003.

[22] Michael Lawley and Jim Steel. Practical Declarative Model Transformation With Tefkat. In *Proc. Model Transformations in Practice Workshop at Models 2005*, Montego Bay, Jamaica, October 2005.

[23] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes In Theoretical Computer Science*, 152:125 – 142, 2006.

[24] Object Managent Group (OMG). MOF 2.0 Query/Views/Transformations Final Adopted Specification (ptc/05-11-01).

[25] Object Managent Group (OMG). MOF 2.0 Query/Views/Transformations Final Adopted Specification (ptc/05-11-01) Figure 7.10.

[26] Object Managent Group (OMG). MOF 2.0 Query/Views/Transformations Final Adopted Specification (ptc/05-11-01) Figure 7.8.

[27] Object Managent Group (OMG). Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10), QVT-Merge Group 1.8, OMG document ad/2004-10-04.

[28] OMG Adopted Specification ptc/03-10-04. The Meta Object Facility (MOF) Core Specification. Version 2.0, OMG, http://www.omg.org.

[29] Karlis Podnieks. Mda: Correctness of model transformations. which models are schemas? *Frontiers in Artificial Intelligence and Applications, Selected papers from 6th International Baltic Conference on Databases and Information Systems*, 118:185–197, 2005.

[30] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J.M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development 1*, 3880:75–105, Febraury 2006.

[31] Richard Soley. MDA, An Introduction. *URL* `http://omg.org/mda/presentations.htm/`, 2002.

[32] Douglas C. Schmidt, Aniruddha Gokhale, Balachandran Natarajan, Sandeep Neema, Ted Bapty, Jeff Parsons, Andrey Nechipurenko, Jeff Gray, and Nanbor Wang. CoSMIC: A MDA tool for Component Middleware-based Distributed Real-time and Embedded Applications. In *Proc. OOPSLA Workshop on Generative Techniques for Model- Driven Architecture*, Seattle, WA USA, November 2002.

[33] Bran Selic. The pragmatics of model-driven development. *IEEE Softw.*, 20(5):19–25, 2003.

[34] S. Sendall, G. Perrouin, N. Guelfi, and O. Biberstein. Supporting Model-to-Model Transformations: The VMT Approach. In *Proc. Workshop on Model Driven Architecture: Foundations and Applications. Proceedings published in Technical Report TR-CTIT-03-27*, University of Twente, 2003.

[35] Shane Sendall. Source Element Selection In Model Transformation. In *UML 03 Workshop in Software Model Engineering (WiSME). Proceedings to be published in Technical Report, University of Bremen*, San Francisco, USA, 2003.

[36] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software, Special Issue on Model Driven Software Development*, 20(5):42–45, 2003.

[37] Raul Silaghi, F. Fondement, and Alfred Strohmeier. Towards an MDA-Oriented UML Profile for Distribution. In *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference, EDOC*, Monterey, CA, USA, September 2004.

[38] Sun Microsystems. Jini Technology Architectural Overview. *URL: http://wwws.sun.com/software/jini/whitepapers/architecture.html*, January 1999.

[39] Sun Microsystems. Jini Network Technology: An Executive Overview. *URL: http://wwws.sun.com/software/jini/*, 2001.

[40] The Object Management Group. CORBA IDL Language Mappings Specifications. *URL* `http://omg.org/corba/`, 2004.

[41] The Object Management Group. The Common Object Request Broker Architecture CORBA/IIOP 2.6. *URL* `http://omg.org/corba/`, 2004.

[42] The Object Management Group. MDA Success Stories. *URL* `http://www.omg.org/mda/products-success.htm/`, 2006.

[43] The Object Management Group. The Model Driven Architecture. *URL* `http://www.omg.org/mda/`, 2006.

[44] The Object Management Group (OMG). The MDA Technical Architecture ormsc/01-07-01. *URL http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01*, 2001.

[45] The Object Management Group (OMG). MDA Guide Version 1.0. *URL* `http://www.omg.org/mda/`, 2003.

[46] The Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0, Final Adopted Specification, OMG, http://www.omg.org, August 2003.

[47] The Object Management Group (OMG). The OMG Web Page. *URL* `http://omg.org/`, 2006.

[48] The Software Engineering and Systems Software Group at Freie Universitt Berlin and Xtradyne Technologies AG. JacORB: The free Java implementation of the OMG's CORBA standard. *URL http://jacorb.inf.fu-berlin.de/*, 2006.

[49] TRISKELL. The KerMeta Project Home Page. *URL* `http://www.kermeta.org`, 2005.

[50] TRISKELL. The KerMeta Manual. *URL* `http://www.kermeta.org`, 2006.

[51] Jim Waldo. Alive and Well: Jini Technology Today. *IEEE Computer*, 33(6):107–109, June 2000.

[52] Jos Warmer and Anneke Kleppe. *The Object Constraint Language Second Edition: Getting Your Models Ready For MDA*. Addison-Wesley.