DISSERTATION

# A META-MODELING APPROACH TO SPECIFYING PATTERNS

Submitted by

Dae-Kyoo Kim

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2004

COLORADO STATE UNIVERSITY

June 21, 2004

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY DAE-KYOO KIM ENTITLED A META-MODELING APPROACH TO SPECIFYING PATTERNS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Committee Member: Dr. James M. Bieman

Committee Member: Dr. Sudipto Ghosh

Committee Member: Dr. Daniel E. Turk

Adviser: Dr. Robert B. France

Department Head: Dr. L. Darrell Whitley

ABSTRACT OF DISSERTATION

A META-MODELING APPROACH TO SPECIFYING PATTERNS

A major goal in software development is to produce quality products in less time and with less cost. Systematic reuse of software artifacts that encapsulate high-quality development experience can help one achieve the goal. Design patterns are a common form of reusable design experience that can help developers reduce development time. Prevalent design patterns are, however, described informally (e.g., [35]). This prevents systematic use of patterns.

The research documented in this dissertation is aimed at developing a practical pattern specification technique that supports the systematic use of patterns during design modeling. A pattern specification language called the Role-Based Metamodeling Language (RBML) was developed as part of this research. The RBML specifies a pattern as a specialization of the UML metamodel. The RBML uses the Unified Modeling Language (UML) as a syntactic base to enable the use of UML modeling tools for creating and evolving pattern specifications.

We used the RBML to develop specifications for design patterns in the Design Patterns book [35] including Abstract Factory, Bridge, Decorator, Observer, State, Iterator, and Visitor. We also used the RBML to define a large application domain pattern for checkin-checkout systems, and used the resulting specification to develop UML designs for a library system and a car rental system. In addition, we used the RBML to specify access control mechanisms as patterns including Role-Based

Access Control (RBAC), Mandatory Access Control (MAC), and a Hybrid Access Control (HAC) that is a composition of RBAC and MAC. The RBML is currently used at NASA for modeling pervasive requirements as patterns. NASA also uses the RBML in the development of Weather CTAS System that is a weather forecasting system. To determine the potential of the RBML to support the development of tools that enable systematic use of patterns, we developed a prototype tool called RBML-Pattern Instantiator (RBML-PI) that generates conforming UML models from RBML pattern specifications.

Dae-Kyoo Kim
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 2004

# ACKNOWLEDGEMENTS

I am greatly indebted to many people for the dissertation, which would not have been possible to complete without their support, help, and patience. It is my privilege to express my gratitude to these people for their part in completion of the dissertation.

To family and friends who have been with me since I started my journey towards scholarship in 1995, I thank you! My deepest love and respect go to my mother who dedicated her whole life to raise me and my sister ever since my father passed away when I was twelve years old. My heart goes to my wife, Jin-Hee who endured many years of hardship, but wonderfully made it. I specially thank Dr. Piatkowski and his family for constant love and support since I met them at WMU in 1996.

I thank my committee members - Dr. Robert France, Dr. Sudipto Ghosh, Dr. James Bieman, and Dr. Dan Turk. Special thanks go to my advisor, Dr. Robert France. He was the person who has been a tremendous source of inspiration to this dissertation. He's been always available when I needed him and painstakingly edited my coarse drafts to the point of perfection.

I am grateful to all those who have made me who I am today.

DEDICATION

*This dissertation is dedicated to my mother.*

TABLE OF CONTENTS

LIST OF FIGURES

# Chapter 1

# Introduction

## 1.1 Problem Statement

In recent years, software applications have dramatically grown in size and complexity. A major software development goal is to produce quality software in less time. Systematic reuse of software artifacts that encapsulate high-quality development experience can help developers reduce development time [71, 81].

Design patterns are a common form of reusable designs. A design pattern describes a family of solutions (henceforth referred to as pattern solutions) for a class of recurring design problems [35]. Prevalent descriptions of design pattern solutions (e.g., [16, 35, 41, 80, 92]) use typical examples to describe the structure of pattern solutions. These descriptions are supplemented by textual descriptions of other aspects such as the problems targeted by the patterns, participant collaborations, the consequences of applying the patterns, and implementation issues. Fig. 1.1 shows the diagram used in the Gamma *et al.* patterns book [35] to describe the structure of Abstract Factory pattern solutions. This diagram is a typical example of an Abstract Factory pattern solution, and not a specification of the family of solutions covered by the Abstract Factory pattern [66].

These structured, informal descriptions are useful for communicating design experience to developers [102]. However, the informal nature of pattern solution de-

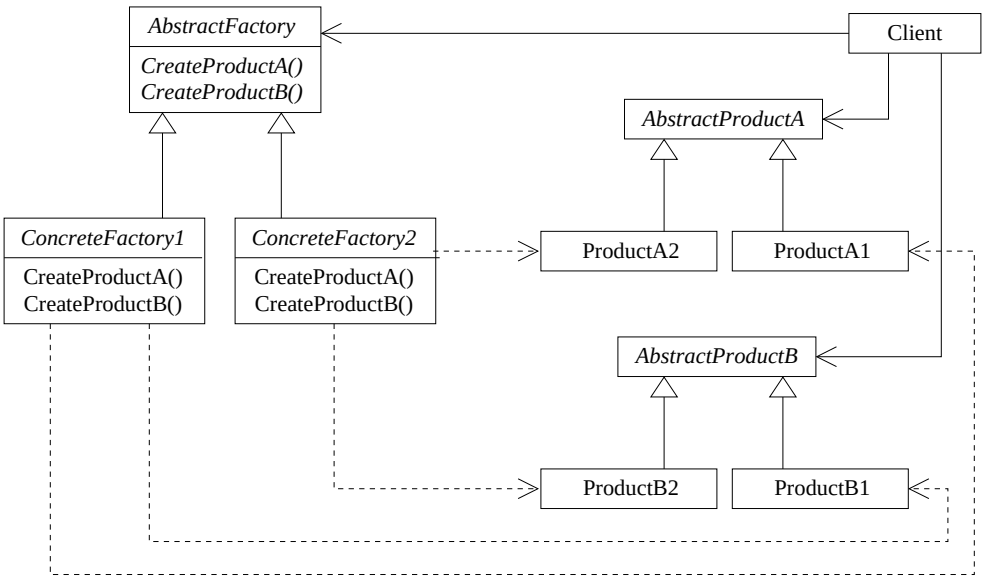



Figure 1.1: The Abstract Factory Pattern [35]

scriptions inhibits their use as a base for the development of tools that support the systematic use of patterns in software development. The problem with informal specifications of design patterns solutions is that they provide poor support for the following activities:

- Verifying conformance to a pattern: Conformance is concerned with establishing that a solution has the described pattern properties. Defining a rigorous notion of pattern conformance requires precise specification of pattern properties.

- Developing tools to support systematic application of patterns: Examples include tools that can (1) verify the presence of pattern solutions in designs (e.g., [47, 95]), (2) incorporate a pattern solution into a design (e.g., [55, 83]), and (3) generate solutions that conform to patterns (e.g., [77]). These tools require patterns to be rigorously specified.

- Communicating pattern properties: Rigorous pattern specifications enable non-ambiguous communication of pattern properties.

## 1.2 Research Overview

The goal of this research is to develop a rigorous and practical pattern specification technique that supports the systematic use of patterns during design modeling. "Practical" in this context means the technique is based on a widely known modeling language. To achieve the goal a language, the Role-Based Metamodeling Language (RBML), for precisely specifying pattern solutions was developed. The language has a syntax based on the Unified Modeling Language (UML) [100]. An RBML pattern specification defines a specialization of the UML metamodel that characterizes the set of UML solution models for the pattern. A model that conforms to a pattern specification satisfies the properties stated in the pattern specification.

The UML is used in this work for the following reasons:

- The UML is the *de facto* standard modeling language for object-oriented modeling, and there is a rapidly growing UML user base in industry. Using the UML as the syntactic base for the RBML makes it easier for UML modelers to create, understand, and evolve pattern specifications.

- The Object Management Group (OMG), the group that maintains the UML standard, is promoting an initiative called *Model Driven Architecture* (MDA) that supports the use of models as primary artifacts of development (see http://www.omg.org/mda). MDA raises the level of abstraction at which complex systems are developed. Technology that supports transformation of models is a key enabler of MDA. There is a growing interest in developing tools that support the transformation of models using design patterns (referred to as *model refactoring*). Such tools require precise descriptions of pattern solutions expressed in a widely used modeling notation such as the UML.

- The RBML uses UML syntax, and thus UML tools can be used to create and evolve RBML specifications and tools. This research uses IBM Rational Rose

as a base for an RBML specification tool.

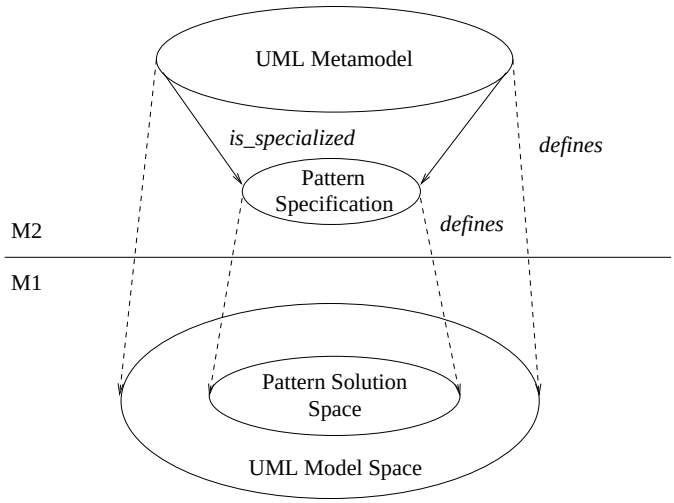



Figure 1.2: An Overview of the Approach

In the approach we developed, a specification of a pattern's solutions is obtained by specializing the UML metamodel so that it defines only models of the pattern solutions. This is illustrated in Fig. 1.2. M1 and M2 in Fig. 1.2 denote levels in the UML infrastructure where UML models are defined at the M1 level and the UML metamodel is defined at the M2 level.

An RBML specification consists of a set of role diagrams that describe pattern properties from different perspectives. A role diagram consists of roles where a role specifies properties of pattern participants. The role diagrams are notational variants of UML diagrams.

An RBML specification can consist of three types of specifications - Static Pattern Specifications (SPSs) present a UML class diagram view of pattern solution properties, Interaction Pattern Specifications (IPSs) present an UML interaction diagram view of pattern solution properties, State Machine Pattern Specifications (SMPSs) present a UML statechart view of pattern solution properties.

We demonstrate the utility of the RBML by using it to specify 1) Gamma *et al.*

design patterns [35]: Abstract Factory, Bridge, Iterator, Decorator, State, Visitor, and Observer, 2) a large domain pattern for checkin-checkout systems [57, 58], 3) and access control aspects (see [26, 91]). We developed a prototype tool for creating UML models from RBML specifications to demonstrate the feasibility of developing RBML-based tools that enable systematic use of patterns.

## 1.3 Scope

Existing descriptions of design patterns (e.g., [35]) often provide more information on their solution spaces than their problem spaces. For this reason, we only focus on specifying the solution spaces of patterns.

Pattern solution properties can be described from various perspectives. In this research pattern solutions are described in terms of the views that are supported by the UML. Specifically, we focus on UML class diagram, interaction diagram, and statechart diagram views of pattern solutions. Our work on the statechart diagram view of a pattern specification is not fully developed and we present only our preliminary results.

Pattern solutions have a functional dimension that consists of the functional properties of the pattern and non-functional dimensions that consist of descriptions of aspects such as performance, fault-tolerance, and capacity. In this research, we focus on specifying the functional properties (structural and behavioral properties) of pattern solutions.

Patterns can be classified into general-purpose patterns and domain specific patterns. General-purpose patterns describe solutions for problems that occur across different application domains. Examples of such patterns are the Gamma *et al.* design patterns [35]. Domain specific patterns describe a single application domain (e.g., telecommunication, banking systems). We show how the RBML can be used to specify both general-purpose and domain specific patterns.

## 1.4 Overview of the Dissertation

Chapter 2 presents the background and the state of the work related to the research documented in this dissertation. Chapter 3 describes the Role-Based Metamodeling Language (RBML). Chapter 4 describes how the RBML can be used to specify Gamma *et al.* design patterns. Chapter 5 describes how the RBML can be used to specify domain patterns using a pattern in the checkin-checkout domain where applications check in and check out items. Chapter 6 describes the use of the RBML to specify cross-cutting functionality using access control policies in a security domain. Chapter 7 demonstrates RBML tool support. Chapter 8 concludes the dissertation and describes future work. Appendix A collects the RBML specifications of design patterns presented in chapter 4, and describes specifications for Observer, Bridge, Decorator, and State patterns. Appendix B collects the RBML specifications of the domain pattern and the access control patterns presented in chapter 5 and 6. Appendix C shows a vehicle rental system generated from an RBML tool.

# Chapter 2

# Background

This chapter presents the technical background for the research. Section 2.1 gives an overview of of design patterns. Section 2.2 describes the infrastructure of the UML and gives an overview of the types of diagrams provided by the UML. Section 2.3 describes how the UML metamodel can be specialized. Section 2.4 presents related work on specifying design patterns.

## 2.1 Design Patterns

The notion of a software pattern is based on the notion of a pattern as defined in the field of architecture by Alexander [4, 5]. In 1987, Cunningham and Beck introduced a set of patterns for developing user interfaces in Smalltalk at an OOPSLA (Conference on Object-oriented Programming Systems, Languages and Applications) [21]. In 1995, Gamma, Helm, Johnson, and Vlissides published a book "Design Patterns: Elements of Reusable Object-Oriented Software", which is one of the most popular books on design patterns. There are many other publications on patterns (e.g., see [16, 80, 92]). Buschmann *et al.* [16] describes patterns that occur at several levels of abstraction ranging from architecture level to programming level. Pree [80] uses metapatterns to describe frameworks. Schmidt *et al.* [92] describes patterns in concurrency and networking.

A design pattern describes a generic solution to a recurring design problem. Typ-

ically, a pattern description includes a name, a description of the problems addressed by the pattern, diagrams and text describing the structure of the generic solution, and descriptions of the consequence of applying the pattern.

Design patterns facilitate communication of solutions among software developers by providing an explicit description of intent, participants, and consequences [1]. For example, use of a self-explanatory or well-known pattern name in a design description is often enough to communicate the details of the solution.

A typical description of a design pattern consists of two major parts [16, 35, 80, 92]:

- *Usage context*: This part consists of usage guidelines (including descriptions of the problems addressed by the pattern), descriptions of solution, quality attributes (e.g., performance, reliability, security), consequences of applying the pattern, and implementation concerns (e.g., see examples in Gamma *et al.* patterns [35]).

- *Solution description*: This part consists of descriptions of behavioral and structural aspects of the solutions characterized by the pattern (e.g., the Structure, Participants, and Collaborations parts of the Gamma *et al.* pattern descriptions [35]).

While it may not be possible (or desirable) to formally express all aspects of a pattern's usage context, it is possible to formally describe pattern solutions when their structural and behavioral properties are well-understood. The research documented in this dissertation focuses on specifications of pattern solutions. A pattern specification language based on the UML is used to describe pattern solutions.

## 2.2 Unified Modeling Language

The Unified Modeling Language (UML) [100] is a standard modeling language for object-oriented systems developed by the Object Management Group (OMG) (see

www.omg.org), a standards body for the object-oriented community. The UML started out as a modeling language developed collaboratively by Grady Booch, James Rumbaugh, and Ivar Jacobson. Booch and Rumbaugh unified their respective methods to produce the Unified Method v 0.8 in 1995. Jacobson joined the collaboration in 1996 to work on UML 0.9 [63].

UML 1.0 was proposed by the UML Partners, a consortium of several organizations, in 1997 in response to an OMG's request for proposals for a standard object-oriented analysis notation and semantic metamodel. Several revisions have been produced since the UML 1.0, and the most recent work, version 2.0, was recently approved by the OMG. The UML has gained prominence and is the industry de-facto standard modeling language in software development.

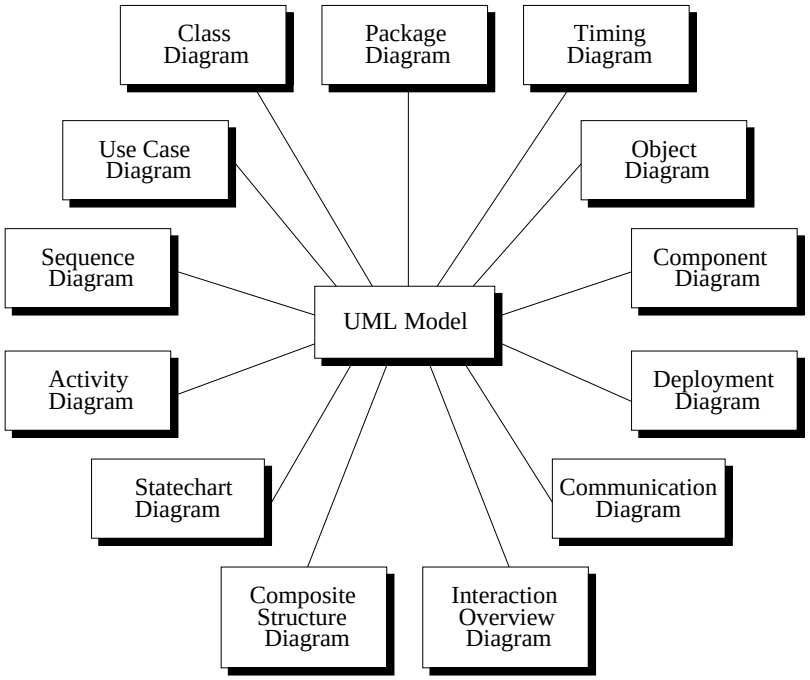



Figure 2.1: UML Diagrams

A UML model can consist of several diagrams as shown in Fig. 2.2. Each diagram describes a different view of the system being modeled. The following briefly describes each diagram supported by the UML [90, 100]:

- A Use Case Diagram describes required behavior of a system as it appears to an user. It partitions the system functionality into interactions called *use cases*.
- A Sequence Diagram describes how instances interact to accomplish a task.
- A Communication Diagram shows the configuration of objects in an interaction that accomplishes a task.
- A Statechart Diagram describes behavior of objects over time in terms of state transitions triggered by events.
- An Activity Diagram describes the flow of control (and optionally data) through steps of a computation.
- A Class Diagram describes classifiers and their relationships.
- An Object Diagram shows objects and their relationships at a point in time. An object diagram is a particular instance of a class diagram.
- A Component Diagram depicts the components of an application and their interfaces and relationships.
- A Deployment Diagram describes the implementation structure of a system in terms of nodes. A node is a run-time computational resource, such as a computer processor. A node can contain physical entities such as files.
- A Composite Structure Diagram depicts the internal structure of a classifier (such as a class, component, or use case) including the interaction points of the classifier to other parts of the system.
- An Interaction Overview Diagram is a variant of an activity diagram that gives an overview of the control flow within a system or business process. Each node/activity within the diagram can represent another interaction diagram.

- A Package Diagram shows how model elements are organized into packages, and the dependencies between the packages.

- A Timing Diagram depicts the change in state or condition of a instance or a role of a classifier over time. It is typically used to show the change in state of an object over time in response to external events.

In general, not all diagrams are needed to model an application. Diagrams are used depending on the aspects of the systems that are of interest. In this research, class diagrams, sequence diagrams, and statechart diagrams are used. These diagrams are described in more detail in the next subsection.

### 2.2.1 UML Diagrams

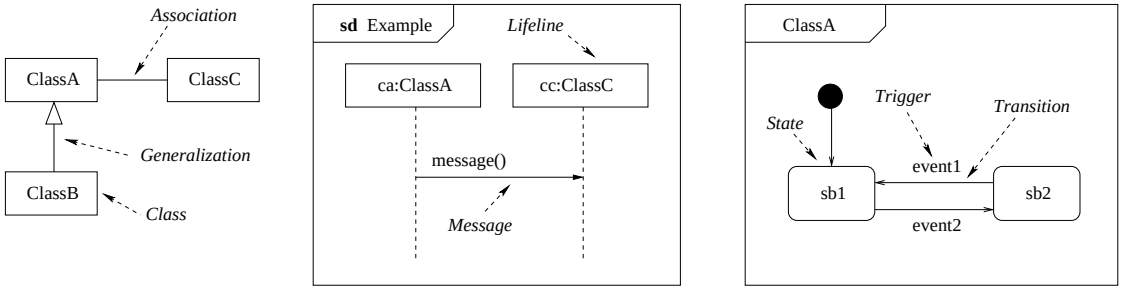



Figure 2.2: Class Diagram, Sequence Diagram, and Statechart Diagram

Fig. 2.2 shows examples of a class diagram, a sequence diagram, and a statechart diagram. A class diagram describes classifiers (e.g., classes, interfaces) and their relationships. A class is a classifier that characterizes a family of objects in terms of attributes and operations that are common to the objects. An operation can be specified using pre- and post-conditions expressed in the Object Constraint Language (OCL) [99]. Associations between classes specify links between class objects. The ends of associations, referred to as *association-ends*, have properties such as multiplicity and

navigability. The class diagram in Fig. 2.2 shows three classes *ClassA*, *ClassB*, and *ClassC*, and their relationships. *ClassA* has a generalization/specialization relationship with *ClassB*, which specifies that *ClassB* inherits the features of *ClassA*. The association between *ClassA* and *ClassB* declares that there can be links between the instances of *ClassA* and *ClassC*.

A sequence diagram describes how objects interact to accomplish a task. An interaction is expressed in terms of *lifelines* and *messages*. A lifeline is a participant in an interaction. In this research, participants are class objects. A message is a specification of a class of stimuli passed between two objects. A stimulus is a communication and can be a request to invoke a recipient's method or a signal that informs its recipient of the occurrence of an event. In this research, we focus on the messages that represent operation calls. The sequence diagram notation can be used to specify (1) alternative sets of interactions and (2) iterations over interactions. The sequence diagram in Fig. 2.2 shows that *ca*, a lifeline of *ClassA*, sends a message to *cc*, a lifeline of *ClassC*, to carry out a specific goal.

A statechart diagram describes behavior of objects over time in terms of state transitions triggered by events. A *state* is a set of object values for a given class that have the same qualitative response to events. A *trigger* specifies an event that represents the kind of changes that an object can recognize, for example the receipt of calls or explicit signals from other objects, a change in values, or the passage of time. A *transition* is triggered by the occurrence of an associated event. In this research, we focus on the triggers that are associated with operation calls (i.e., call triggers). The statechart diagram in Fig. 2.2 show a statemachine of an object of *ClassA* that has two states *sb1* and *sb2*, and two transitions; one from *sb2* to *sb1* triggered by *event1* and the other from *sb1* to *sb2* triggered by *event2*.

### 2.2.2 The UML Metamodel

The UML metamodel characterizes valid UML models. It consists of a class diagram and a set of well-formedness rules that define the abstract syntax of the UML. Informal descriptions of semantics are also included in the metamodel. The metamodel class diagram consists of classes whose instances are UML model elements. For example, instances of the metamodel class *Association* are UML associations. Well-formedness rules that are not expressible in the metamodel class diagram are expressed using the OCL where possible, and in natural language otherwise.

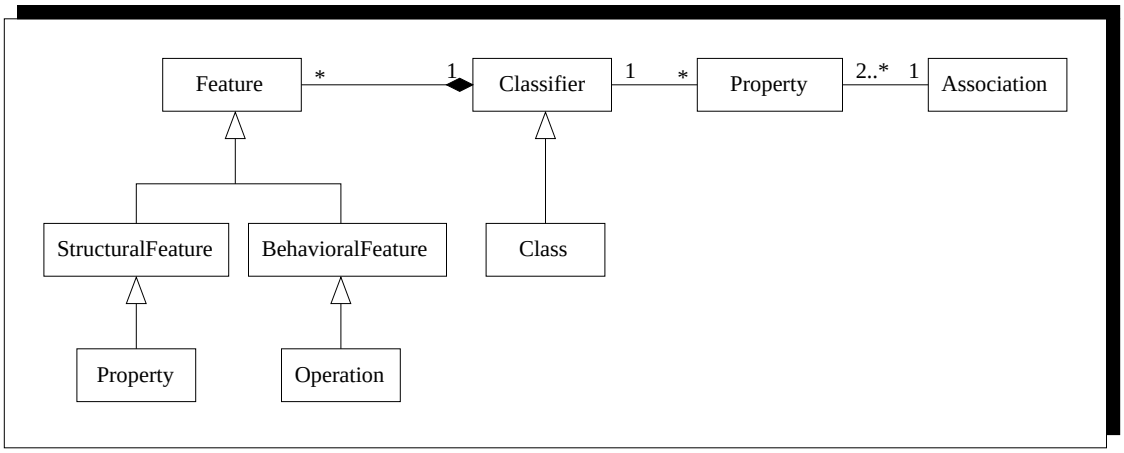



Figure 2.3: A Part of the UML Metamodel

Fig. 2.3 shows a part of the UML metamodel class diagram (class attributes are not shown) [100]. A classifier describes a set of instances that have features in common. A class is a kind of classifiers whose features are attributes and operations. Attributes of a class are represented by instances of *Property* that are owned by the class. Some of these attributes may represent the navigable ends of binary associations. In Fig. 2.3 the multiplicity *2..** at the *Property* end specifies that an association must have at least two association ends (properties).

Operations of a class are represented by instances of *Operation*. An operation is a behavioral feature of a classifier that specifies the name, type, parameters, and

constraints for an associated behavior.

UML metaclasses may have attributes. For example, *Classifier* has an *isAbstract* attribute that indicates whether the classifier is abstract or not.

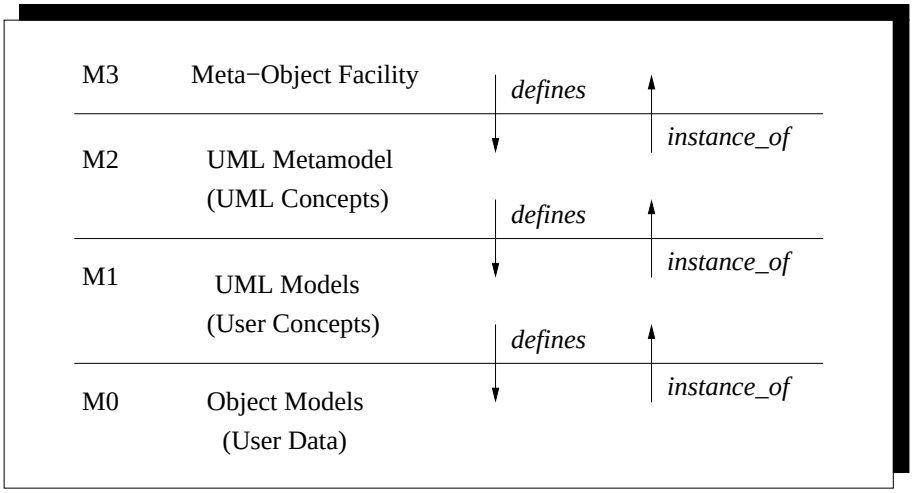



Figure 2.4: UML Four-Layer Infrastructure

The UML infrastructure is defined as a four-layer metamodel architecture as shown in Fig. 2.4: Level M3 defines a language for specifying metamodels, level M2 defines the UML metamodel, level M1 consists of UML models specified by the M2 metamodel, and level M0 consists of object configurations specified by the models at level M1.

The UML provides extension mechanisms in the form of stereotypes, tagged values, and constraints. A stereotype defines how an existing metaclass may be extended. For example, stereotypes can be used in the business modeling area to distinguish business objects and business processes as special kinds of modeling elements whose usage is distinct within a given development process [90].

## 2.3 Specializing the UML

The UML metamodel defines the syntax and semantics of the concepts (metaclasses) that are used for describing models. A model comprises application concepts and their relationships which are instances of the metaclasses defined in the metamodel. The UML metamodel can be constrained by specializing the metaclasses to define a subset of the UML models. For example, models in the telecommunication domain can be defined by specializing the UML metaclasses with telecommunication-specific properties [25]. In this research a pattern solution is specified by specializing the UML metamodel with pattern properties so that it defines only the UML models that describe pattern solutions. Tool vendors can use specialized metamodels that define pattern solutions for developing tools that allows one to build models of pattern solutions.

## 2.4 Related Work on Pattern Specification

The section describes prior work on pattern specifications. The related work is categorized as formal methods-based work and UML-based work.

### 2.4.1 Approaches Based on Formal Methods

There has been work on formalizing design patterns using formal specification techniques (e.g., [24, 65, 70]). Lano *et al.* treat design patterns as transformations. In their approach, a pattern consists of a "before" and "after" specifications whose structures are described in $VDM^{++}$ [23]. The semantics of a transformation from a "before" system to an "after" system is expressed using an extension of Object Calculus [27]. The extension was chosen because it allows the use of structured actions that corresponds to programming language statements. The semantics is used to prove that a design pattern as a transformation is a formal refinement. For example, the before specification of the Abstract Factory pattern defines a *Client* class that

specifies an action for creating products, and the after specification defines *Client* and *ConcreteFactory* classes where the *Client* specifies an action for creating factory. These classes are expressed in $VDM^{++}$ terms. A mapping between the terms in the before *Client* and the after *Client* is defined. The interpretation is used in the axiomatic transformation from the before specification to the after specification. A limitation of the technique is use of unfamiliar formalisms.

LePUS [24] was a formal language that is developed to support programming-level use of patterns. LePUS defines patterns in terms of object-oriented program properties. A program is said to implement a pattern if the program conforms to the pattern constraints. LePUS consists of basic "building blocks" such as hierarchy, totality, isomorphic, and clan. A hierarchy is a set of classes that contains an abstract class and other classes that inherit properties from the abstract class. Totality is a relation between two sets such that for every element in one set, there is a mapped element in the other set. Isomorphic is a restricted totality in which there is a one-to-one relationship or a bijection between two sets. A clan is a relationship between abstract methods and concrete classes that implement the abstract methods.

LePUS specifications are formulae expressed in higher order monadic logic [11]. A LePUS formula consists of variables and relations which correspond to the aforementioned building blocks. Variables (e.g., hierarchies, clans, tribes) represent program elements of classes and functions. Relations (e.g., totality, isomorphic) are predicates similar to PROLOG predicates, and define constraints on the variables. For example, the specification of the Factory Method pattern consists of 1) variables for *Creators* and *Products* hierarchies, 2) a clan for *FactoryMethod* in *Creators*, and 3) an isomorphic relation between *FactoryMethods* and *Products*. A program is said to be valid for a pattern specification if it satisfies the constraints defined on the variables in the specification. LePUS provides a visual notation that can represent the formulae as a diagram that is semantically equivalent to the formulae, but more abstract.

DisCo [70] is a pattern specification language that uses Temporal Logic of Actions [64] and UNITY [18] to formalize temporal aspects of design patterns. A DisCo specification consists of classes and their actions and relations. A class is defined with local variables that are used in action specifications. An action consists of a list of participants and parameters, a condition, and the definition of state changes. A relation defines an association between two classes. For example, the specification of the Observer pattern contains *Subject* and *Observer* classes, *Attached* and *Updated* relations between the classes, and *Attach*, *Detach*, *Notify*, and *Update* actions. DisCo specifications can be combined to produce a more complex pattern specification that can be used to develop specifications for complex systems. DisCo provides class refinement to support the combination of the specifications defined at a different level of abstraction. The combined specification is instantiated by binding concrete data to pattern properties. Such a template paradigm can only capture structural variations of pattern realizations. For example, in the Observer pattern there may be more than one observer that observes a subject, or there may be an observer that observes more than one subject. DisCo is not capable of specifying such variations.

### 2.4.2 UML-Based Approaches

Guennec *et al.* [44] use a UML metamodeling approach in which pattern properties are expressed in terms of *meta-collaborations*. A meta-collaboration consist of roles (e.g., classifier roles, feature roles) that are played by instances of UML metamodel classes. A classifier role with stereotype *hierarchy* represents a structure of concrete classes. Feature roles are stereotyped with *clan* representing a set of features that share the same signature, and *Tribe* representing a set of clans. Feature roles are associated with a classifier role through a composition relationship. For example, the meta-collaboration of the Visitor pattern defines *Element* and *Visitor* classifier roles and *Accept* and *Visit* feature roles. The *Element* role is associated with *Accept*

role, and the *Visitor* role is associated with the *Visit* role. Guennec *et al.* clearly point out deficiencies in the UML notion of role models, and provide an alternative representation in terms of meta-collaborations that utilize a family of recurring properties initially proposed by Eden [24]. However, Guennec *et al.* do not address how structural features of patterns can be specified. For example, *Subject* in the Observer pattern has a structural feature to store the current status of the subject, but is not clear how the feature can be expressed. A behavioral role is described using a class, but signatures of behavioral roles are not addressed. Patterns also describe interactions among participants. The authors mention several possible approaches to specify interactions of pattern roles, but no concrete solution is provided. The notion of conformance is coarse-grained. A model element is said to conform to a role if the model has a name matching to the role name.

Lauder and Kent [66] view pattern realization as a refinement process in which a high-level pattern description is refined to a model realization. They use graphical constraint diagrams to present patterns in three layers of models: *role-model*, *type-model*, and *class-model*. A role-model describes the essential aspects of a pattern in terms of highly abstract state and behavior elements. The notion of role is a placeholder that can be substituted by types in refinements. A type-model refines a role-model by adding domain-specific constraints. A class-model refines a type-model by deploying concrete classes. The three models are described using a combination of the UML and graphical constrain diagrams. For example, the role model of the Abstract Factory pattern consists of *AbstractFactory* role with a *dot* inside to denote an abstraction of operations, and *AbstractProduct* role with a *star* inside to denote creation of a product instance. The dot and star are linked by a line stereotyped with *create* representing that operations create products. In a type model, the dot is refined into a set of operations, and *AbstractProduct* is refined into a set of types (i.e., products) corresponding to the operations. A class model of the type model adds

hierarchies of concrete classes to *AbstractFactory* and *AbstractProduct*. Behavioral properties of a pattern are described using a variation of sequence diagram. Establishing that a model conforms to a pattern (as expressed by a role-model) involves establishing refinement relationships across the model levels. However, the graphical form of constraints is and is not currently integrated with the UML and it is not clear how tools can support the notation.

The DPML [69] is a visual modeling language that provides a set of modeling constructs (e.g., interface, dimension) to specify design pattern solutions. A pattern specification consists of interfaces, implementations, methods, operations, and attributes. These participants are linked by binary directed relations, and have constraints described in plain text. A pattern solution is realized by instantiating the specification, and binding the instantiated pattern elements to UML model elements. An instantiated diagram consists of "proxy" elements that are instantiated from the pattern participants, and "real" elements that are application-specific added during realization. A participant is played by more than one model element. This is specified by a notion called "dimension". For example, the Abstract Factory pattern is defined with *AbstractFactory* and *Products* interfaces, *concreteFactories* and *concreteProducts* implementations, and *createOps* and *concreteCreateOps* operations in *AbstractFactory* and *concreteFactories*, respectively. A prototype tool is developed. The tool can be used to build pattern specifications and UML class diagrams (what they call object diagrams), instantiate specifications and to check consistency between specifications and class diagrams. The pattern realization mechanism is similar to the templates in the UML in that pattern participants are instantiated and bound to application elements. Such a template paradigm is limited in instantiation in that they only allow uniform instantiation. Furthermore, it is not clear why a new notation had to be created instead of using the UML particularly when DPML is developed for UML models.

Albin-Amiot and Gueheneuc [3] propose a metamodel expressed in the UML to describe structural and behavioral aspects of design patterns for automatic code generation and pattern detection in code. The metamodel is specialized to define a particular pattern by adding specialized classes. The specialized metamodel is instantiated to produce a specification of the pattern. An instantiated specification is expressed in two equivalent forms; visualization using a graphical notation and text in Java. The specification is then bound to a concrete application. The resulting specification is used to generate Java code. For example, the Composite pattern metamodel (i.e., a specialization of the metamodel) defines *PClass*, *PInterface*, and *PDelegatingMethod* metaclasses. An instance of the metamodel (i.e., a Composite pattern specification) consists of *Component* class, *Leaf* and *Composite* interfaces, and *operation* operation. Concrete elements of a drawing application is then bound to the pattern participants: *Graphic* to *Component*, *Line* to *Leaf*, *draw* to *operation*. Pattern detection is performed in code by matching the type of concrete elements with the metaclasses in the metamodel. Based on the identified metaclasses, it is determined which pattern is used in the code. They use a graphical notation to express the instances of the metamodel. However, there is little description on the notation, and it is not clear why the metamodel and its instances would have to be expressed in two different notations. Like Lano *et al.*'s work and LePUS, the technique supports programming-level use of patterns, not model-level use.

### 2.4.3 Summary

Based on the survey of related work, the following issues have been identified:

- Pattern conformance for models: In Lano *et al.*'s work it is not clear how the transformation guarantees for pattern conformance. LePUS defines pattern conformance for programs, not for models. DisCo uses a template paradigm which automatically ensures conformance. In Guennec's work, conformance check is

simply done by matching the names between application elements and roles. In Lauder's approach, conformance is established by defining refinement relationships between the model levels. However, refinement rules are not defined. The DPML checks conformance by type matching between objects and instantiated pattern elements. This is coarse-grained in that properties (i.e., features) of pattern participants are not considered.

- Purpose of Pattern Specification: Pattern specifications can be used for 1) generating conforming artifacts (e.g., models, programs), 2) checking conformance of an artifact, and 3) incorporating pattern properties into artifacts. The first purpose is concerned with generating conforming artifacts from pattern specifications. The last purpose is, however, concerned with transforming designs using patterns. Lano *et al.*'s work addresses the last purpose. LePUS and Guennec's approach address the second purpose, where programs in LePUS and models in Guennec's work are checked for pattern conformance. DisCO concerns the first purpose where a system specification is generated by stamping out a pattern specification. DPML addresses the second and third purposes where for a given object model, the model is checked to determine whether it has pattern participants or not; if not, the missing participants are added. Albin-Amiot's work concerns the first and second purposes where a pattern specification is used to generate code and to detect patterns in code. DPML and Albin-Amiot's work are concerned with a broader purpose than the others, but still limited in that DPML is not concerned with generating conforming artifacts and Albin-Amiot's work is not concerned with incorporating pattern properties. There is a need for techniques that can address all the three purposes in order to facilitate the development of tools that support the three together.

- Notation: Lano *et al.*'s work, LePUS, and DisCo are based on formal notations.

While formal approaches enable rigorous reasoning, they can be difficult for users to read and use, not because of the complexity in the approaches, but because of the unfamiliar mathematical notations. LePUS provides a graphical notation for visualizing LePUS expressions, but the notation is not currently integrated with existing tools. DPML and Albin-Amiot's work also provides graphical notations for specifying patterns, but the syntax and semantics of the notations are not fully described. Techniques that use a standard notation as a base can be adopted easier, and can take advantage of existing tool support.

- Language features: Patterns involve structural and behavioral aspects. However, the previous work mainly focuses on specifying structural properties, and lacks features for specifying behavioral aspects. LePUS cannot specify the signature of behavioral properties. DPML and Guennec's work do not describe how the interactions of pattern participants are specified.

- Complexity of representation: Guennec's work and Albin-Amiot's work specify a pattern as a specialization of the UML metamodel. Every element in a pattern specification is represented as a class. For example, specifying *Subject* in the Observer pattern requires to have five metaclasses; *Subject*, *SetState*, *Attach*, *Detach*, *Notify*. Similarly, in DPML each type of pattern elements has its own graphical symbol. For example, interfaces are denoted by hexagons and operations are denoted by diamonds. Specifying the *Subject* also requires five constructs. The specifications of such approaches quickly become complicated even for small size of patterns.

- Pattern realization: In practice, a pattern can be realized in various forms depending on the problems that the pattern is applied to solve. Approaches like DisCo and Albin-Amiot's work that rely on template paradigm are rigid in that they only allow a uniform realization. The refinement approach in

Lauder's work is more flexible than template paradigm in that the form of refining elements can vary while it satisfies the refinement rules. However, the refinement rules are not clearly defined.

### 2.4.4 Contribution

This research addresses the aforementioned aspects by developing a pattern specification language called Role-Based Metamodeling Language (RBML). The RBML has the following characteristics:

- Model-level use of patterns: The RBML is designed to support systematic use of patterns at the model level. An RBML specification can be used 1) to support generation of conforming models, 2) as a base for conformance checking, and 3) for incorporating pattern properties into a model.

- Rigorous notion of pattern conformance: The RBML defines a set of conformance rules that can be used to check structural and behavioral conformance of models. Structural conformance rules are used to check the type of model elements and their features (i.e., attributes, operations) and relationship. Behavioral conformance rules are used to check the sequence of the interactions between model elements. A model conforms to a pattern if the model satisfies the rules.

- Use of familiar notation: The RBML uses the UML as its syntactic base to make it easier for UML modelers to create, understand, and evolve pattern specifications. This also enables the use of UML modeling tools for creating and evolving pattern specifications.

- Use of concise representation: RBML specifies a pattern as a specialization of the UML metamodel, and the specialized metamodel is represented concisely as a UML-like diagram.

- Flexible pattern instantiation: The RBML provides a special form of RBML specifications called RBML templates. RBML templates allow one to instantiate a pattern in various structures for applications through variation points where users can provide application-specific information that determines the structure of the application.

- Specification views: The RBML provides three types of pattern specifications; Static Pattern Specifications (SPSs), Interaction Pattern Specifications (IPSs), and Statemachine Pattern Specifications (SMPSs), each of which captures different views of pattern solutions. An SPS specifies the static structure of patterns including participants, their properties, and relationships. An IPS specifies interaction constraints among the participants. An SMPS specifies the internal behavior of the participants.

# Chapter 3

# RBML: Role-Based Metamodeling Language

This chapter describes the Role-Based Metamodeling Language (RBML). Section 3.1 describes the notion of role used in the research to define pattern properties. Section 3.2 describes the notation of the RBML, the types of RBML specifications, and conformance rules for each type. Section 3.3 gives a summary of the chapter. Section 3.4 discusses how the RBML can be used for the development of tools that support systematic use of patterns during modeling.

## 3.1 Using Roles to Specify Properties

A design pattern describes a family of solutions for a class of recurring design problems [35]. A model is said to be a member of the family if the model has elements that "play" the roles of participants in the pattern. In this sense, it is natural to consider use of roles to describe pattern properties, which motivates using roles in this research as a base concept. In this section, the notion of role used in the research is defined.

The prevailing notion of role in the object-oriented community is based on objects, that is, a role is played by objects (see [2, 22, 34, 85]). A role played by objects defines a set of properties that must be satisfied by objects playing the role. In this research, a different notion of role is used to support specification of design patterns at the

metamodel level where a role is played by model elements (e.g., classes, associations).

To define the notion of role used in this research, a general notion of role is defined by generalizing the characteristics of roles played by objects. The general role is then specialized to define the notion of role played by model elements.

### 3.1.1 Object Roles

The notion of role in the object-oriented community was introduced by Bachman and Daya [8]. They define a role to be a behavior pattern which may be assumed by objects of different kinds, and a particular object may concurrently play one or more roles. This concept has been a base for the work on roles in object-oriented data modeling [2, 22, 40, 78, 94, 97, 98, 105]. Roles can describe object behavior [78, 98], and extended to role hierarchy [40, 94] for inheritance of role properties. More recent work [22, 97] uses roles in modeling where roles are used to define an object-oriented modeling language [97] and a role model [22].

The notion of role in data modeling is used in object-oriented design modeling [34, 85, 87, 99] to describe views of design concepts (i.e., classes). Roles can describe a particular view of the structure and behavior of objects [99] or identify a behavior associated with an object [87]. The OOram method [85] defines a role as a set of properties that objects must possess in order to play the role.

The following characteristics of object roles are identified from the above work:

1. A role defines a subset of objects of a type [8, 40, 105]. In other words, a role defines constraints on a type. Only those objects that satisfy the constraints can play the role.

2. A role has structural and behavioral properties [22, 78, 98, 105]. Properties of a role may be inherited from other roles in a hierarchy [8, 40, 78, 94, 98, 105]. The hierarchy suggests that an object that plays a role can also play the superrole of

the role. For example, a person (an object) who is a graduate teaching assistant (a role) also plays a graduate assistant (the superrole).

3. An object may play multiple roles simultaneously [8, 78] or the same role several times with a different state of the object during its lifetime [40, 78]. For example, a person (an object) can be a student (a role) and/or an employee (a role).

4. An object may acquire and abandon roles several times during its lifetime [40, 78, 94]. The sequence of acquiring and abandoning roles may be restricted [78, 94]. For example, a person can become an advisor only after being a professor. In order for the object "person" to play an advisor role, the person must acquire a professor role first.

5. A role is context-specific. This means that a role is meaningful only in the context where the role is defined [2, 78, 94]. For example, if a person plays both the student role and employee role, in the context of the employment, the person cannot access a student's grade [2, 40, 94].

### 3.1.2 General Roles

In this section, the characteristics of object roles described in Section 3.1.1 are generalized to obtain a notion of general role. The following simple generalization rules are applied to the characteristics of object roles:

- Runtime-specific characteristics are excluded. Examples of such characteristics are simultaneity of objects that an object can play multiple roles simultaneously at the same time, and dynamicity that an object can change roles to play dynamically during its lifetime. This rules out item 3 and 4 in Section 3.1.1.

- Object is replaced by instance. The term instance is more general than object in that instance can be used at any level of modeling. For example, an object

of a class is an instance of the class, and a class is an instance of a meta-class. This results in substituting object in item 1, 2 and 5 to instance.

The following characteristics are obtained by applying the generalization rules:

1. A role defines a subset of instances of a type.
2. A role has structural and behavioral properties.
3. Properties of a role may be inherited from other roles in the hierarchy. Instances that play a role can also play the superrole of the role.
4. A role is context-specific.

Specializations of the general role inherit the above characteristics, but may also have their own characteristics. An object role is a specialization of the general role in that instances are objects (e.g., instances of classes). Object roles have additional characteristics that are runtime specific. The general notion of role can be specialized to define roles at any level (e.g., metametamodel level).

### 3.1.3 Model Roles

A model role is a role that is played by UML model elements (e.g., classes, associations), not by objects. Model roles inherit the characteristics of general roles, and have their own additional characteristics. The following describes the characteristics of model roles that include both inherited from the general role and newly added:

1. A model role has a base type which is a metaclass in the UML metamodel. Only the instances of the base type that possess the properties defined in the role can play the role. For example, a class role is a model role whose base is *Class* metaclass, and the role is played by instances (classes) of *Class* that satisfy the constraints specified in the role.

2. A model role defines a subset of instances of its base metaclass. For example, an association role specifies a subset of instances (i.e., associations) of *Association* metaclass. These instances are said to play the role.

3. A classifier role has structural and behavioral properties which are also roles whose bases are StructuralFeature and BehavioralFeature metaclasses, respectively. A structural feature role can be played by structural features such as attributes or query operations. A behavioral feature role can be played by operations.

4. Properties of a model role may be inherited from a model role called an *abstract role* which is not realizable. An abstract role is simply an organizational entity and is not meant to be played directly by an instance. Properties specified in an abstract role are inherited by concrete roles that can be realized. A role hierarchy describes the structure of the inheritance.

5. A model role may be played by several instances of its base metaclass. The number of model elements that can play the role may be constrained by multiplicities. For example, A role that has a multiplicity of 1..* can be played by one or more classes.

6. An instance of a metaclass may play several roles that have the metaclass as their bases. For example, if a role A and role B have the *Class* metaclass as their base, a class (an instance of *Class*) can play both roles.

7. Properties of a classifier role can only be accessed in the context of the role. For example, consider the Visitor pattern which uses the Composite pattern to describe the structure of elements [35]. In a solution model of the Visitor pattern, a class that plays the object structure role in the Visitor pattern may also play the composite role in the Composite pattern. In this case, properties

of the class that are specific to the context of the Composite pattern are not accessible in the context of the Visitor pattern.

The characteristics in items 2, 3, 4 and 7 are inherited from general roles; the rest are specific to model roles. The general role characteristic that states that "instances that play a role can also play the superrole of the role" is specialized in model roles; no model elements can play an abstract role (a superrole) in model role hierarchy.

## 3.2 Specifying Pattern Solutions

A pattern specification consists of a *Structural Pattern Specification* (SPS) that specifies the class diagram view of pattern solutions, *Interaction Pattern Specifications* (IPSs) that specify interactions in pattern solutions, and *Statemachine Pattern Specifications* (SMPSs) that specify the statechart diagram view of pattern solutions. The SPS is the core of a pattern specification. The IPSs are defined in terms of interacting participants specified in the SPS. The SMPSs are defined for the participants in the SPS that have state-based behaviors. A pattern specification consists of three components:

- *Role diagrams* that specify the structure of pattern solutions using an SPS, IPSs, and SMPSs.
- *Metamodel-level constraints* that are additional well-formedness rules for the UML metamodel. Role diagrams and metamodel-level constraints together define a specialization of the UML metamodel that describes the abstract syntax for models that conform to pattern solutions.
- *Constraint templates* that define the semantics of pattern solutions. Constraint templates are used to produce constraints that conforming models must incorporate.

A UML model is said to conform to a pattern specification if it possesses the properties defined the pattern specification.

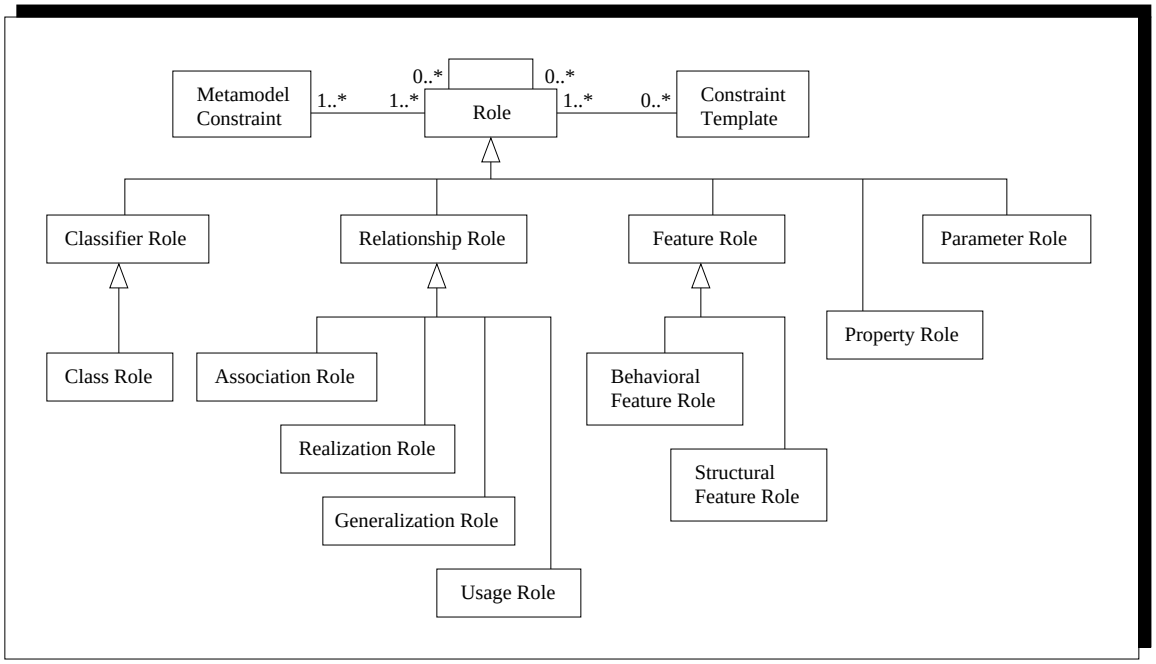



Figure 3.1: RBML Metamodel

Fig. 3.1 shows the RBML metamodel. It defines types of modeling elements for specifying pattern solutions. The top structure of the three classes *MetamodelConstraint*, *Role*, and *ConstraintTemplate* describes that a role must have at least one metamodel constraint and may be associated with constraint templates. The structure of the role specializations below shows role types whose instances define specializations of UML metaclasses. The instances are expressed in the UML, and thus follow the syntax of the UML. For example, instances of *Classifier Role* are expressed using *Classifier* notation, and instances of *Association Role* are expressed using *Association* notation. The UML syntax specifies that an association must be associated with at least two classifiers. Thus, an association role (an instance of *Association Role*) must be associated with at least two classifier roles (instances of *Classifier Role*). The specialization hierarchy in the figure may be extended if other types of

roles are required in specifying certain patterns that have not been attempted to specify. For example, *Enumeration Role* can be added to the hierarchy if the pattern being specified involves enumeration properties.

### 3.2.1 Static Pattern Specifications (SPSs)

An SPS defines the part of the pattern metamodel that characterizes class diagram views of pattern solutions. It defines subtypes of UML metamodel classes describing class diagram elements (e.g., UML metamodel classes *Class*, *Association*) and specifies semantic pattern properties using constraint templates.

An SPS consists of a structure of *pattern roles* [60] (henceforth referred to as roles), where a role specifies properties that a UML model element must have if it is to be part of a pattern solution model. Formally, a role defines a subtype of a UML metamodel class. The metamodel class is called the *base* of the role. A role with a base $B$ specifies a subset of instances of the UML metamodel class $B$. For example, a role that has the metamodel class *Association* as its base specifies a subset of UML associations. A UML model element *conforms to* (or *plays*) a role if it satisfies the properties defined in the role, that is, the element is an instance of the subtype defined by the role.

A role in an SPS can be classified as a *classifier* or a *relationship* role. A role that has the base $Classifier$ or a base that is a subtype of $Classifier$ (e.g., $Class$, $Interface$) is a classifier role. A relationship role is any role that has the base *Relationship* or a base that is a subtype of *Relationship* (e.g., *Association*, *Generalization*).

#### 3.2.1.1 SPS Notation

A classifier role is represented by a syntactic variant of the UML class symbol. The structure of a classifier role is shown in Fig. 3.2.

The top compartment of a classifier role consists of three parts:

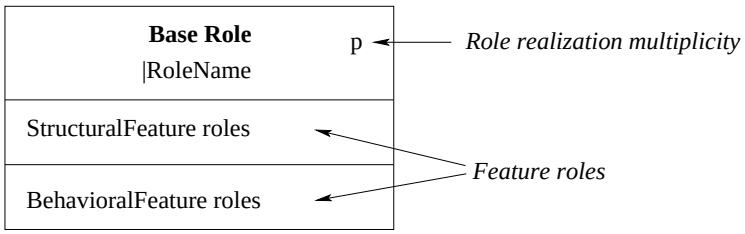



Figure 3.2: Structure of a Classifier Role

- A label of the form **Base Role**, where **Base** is the name of the role's base (i.e., the name of a metamodel class).

- A declaration of the form $|RoleName$, where $RoleName$ is the name of the role. The symbol "|" is used to indicate that the following string is a role name.

- A *realization multiplicity*, $p$, that can restrict the number of classifiers playing the role in a conforming class diagram. The multiplicity can be omitted if the number of conforming classifiers is not constrained (i.e., the multiplicity is $*$).

The other compartments consist of *feature roles* that specify features associated with conforming classifiers. There are two types of feature roles:

- *StructuralFeature roles* specify properties represented by structural features of conforming classifiers. A StructuralFeature role can be played by an attribute or a query (i.e., a value-returning function with no side-effects). Structural feature roles may be labeled using strings of the form "s#" (e.g., s1, s2, ...), where "s" denotes structural feature roles. The labels are used to denote the roles that model elements play in class diagram using stereotypes [1].

[1]This is an informal use of UML stereotypes - the stereotype notation is used to simply label the model elements playing the roles.

- *BehavioralFeature roles* specify behavioral properties associated with conforming classifiers. A BehavioralFeature role can be played by an operation. Similar to structural feature roles, behavioral feature roles may be labeled using "b#" where "b" denotes behavioral feature roles.

Each feature role is associated with a realization multiplicity that can constrain the number of features (e.g., attributes or operations) in a conforming classifier playing the feature role. A realization multiplicity with a lower bound of 0 (e.g., $*$) indicates that the feature may or may not be present in a conforming classifier (i.e., it is an optional feature).

#### 3.2.1.2 An Example of Class Role

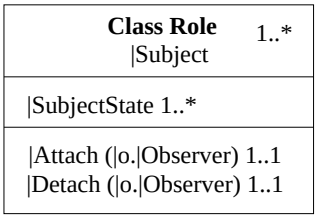



Figure 3.3: A Class Role

Fig. 3.3 shows a class role whose base is the *Class* metaclass. The role specifies that one or more classes can play the role. A class playing the role must one or more structural features that play the *SubjectState* role, and exactly one feature that play each of the *Attach* and *Detach* roles.

#### 3.2.1.3 Roles and the UML Metamodel

Fig. 3.4 describes the relationship of a role to the UML infrastructure [99]. *MyRole* is a role whose base type is the *Class* metaclass. *MyRole* defines a subset of instances (classes) of the *Class* metaclass. *ClassA* is an instance of *Class* and it is a member of the subset defined by *MyRole*. Thus, *ClassA* is said to play the *MyRole*. The *RoleA*

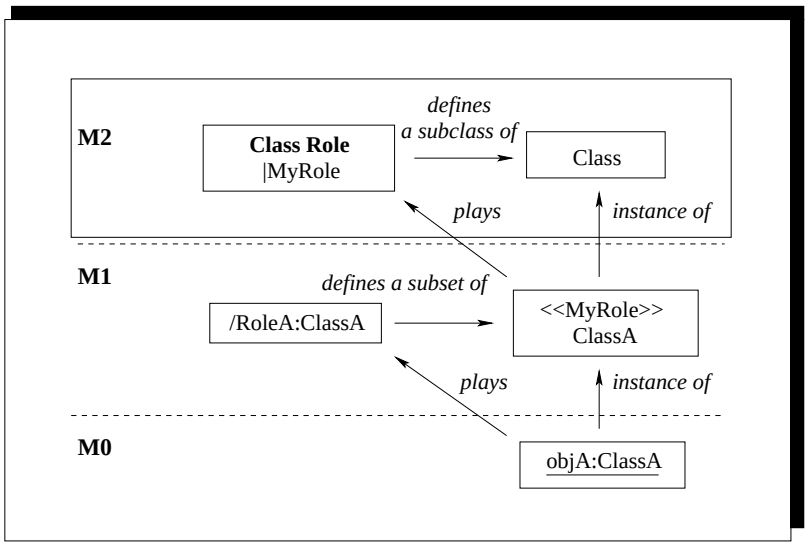



Figure 3.4: Relationship between Role and UML Infrastructure

is an UML role that defines a subset of instances (objects) of the *ClassA*. The *RoleA* is played by an object *objA* which is an instance of the *ClassA*.

#### 3.2.1.4 Role Hierarchies

Properties of a classifier role can be inherited from an abstract classifier role as shown in Fig. 3.5(a). In the hierarchy, the *MyRole* is an abstract role that is simply an organizational entity and is not meant to be played directly. Conforming classifiers must play at least one of its role specialization - *AbstractMyRole* and *ConcreteMyRole*. If there is a classifier that plays the *AbstractMyRole*, then there must be at least one relationship that plays either the *MyRoleRealization* role or *MyRoleGeneralization* role in a conforming structure. A role hierarchy can be concisely represented in the RBML by a single construct called the folded hierarchy role. An example of a folded hierarchy role for the *MyRole* hierarchy is shown in Fig. 3.5 (b).

The following are the well-formedness rules associated with the role hierarchy:

- A conforming classifier of *AbstractMyRole* must either be an interface or an abstract class:

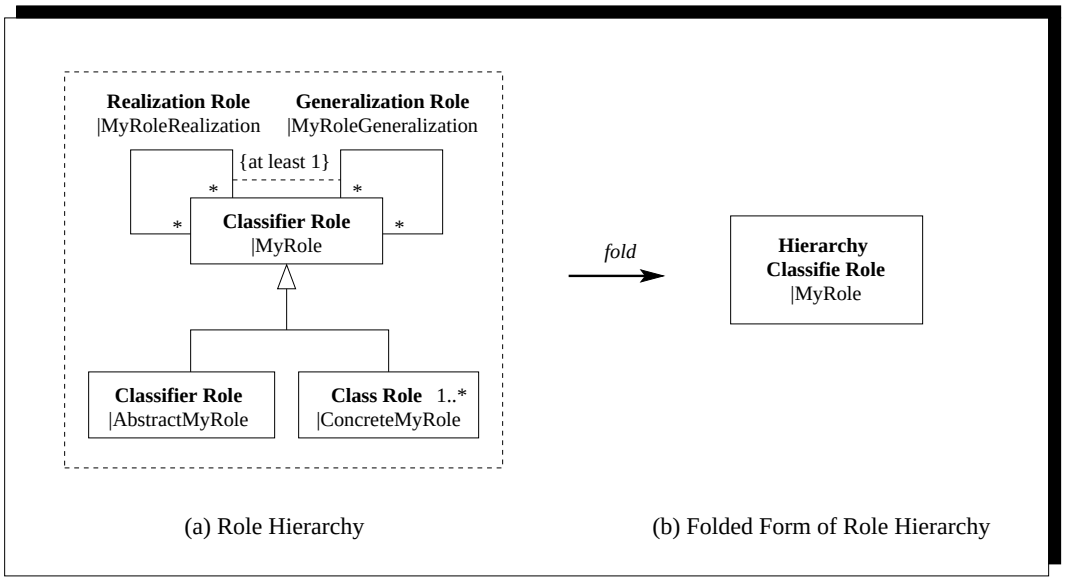



Figure 3.5: Role Hierarchy

**context** |AbstractMyRole **inv**

self.oclIsTypeOf(Interface) or

(self.oclIsTypeOf(Class) and self.isAbstract = true)

- A conforming class of *ConcreteMyRole* must be a concrete class:

  **context** |ConcreteMyRole **inv**: self.isAbstract = false

- A relationship that conforms to *MyRoleRealization* must have an interface or a type at its supplier end and a concrete class at its client end.

  **context** |MyRoleRealization **inv**:

  (self.supplier.oclIsTypeOf(Interface) or

  (self.supplier.oclIsTypeOf(Class) and self.supplier.isAbstract = true) and

  self.client.oclIsTypeOf(Class)

- A relationship that conforms to *MyRoleGeneralization* must have its parent and child to be the same type.

**context** |MyRoleGeneralization **inv**:

self.parent.evaluationType() = self.child.evaluationType()

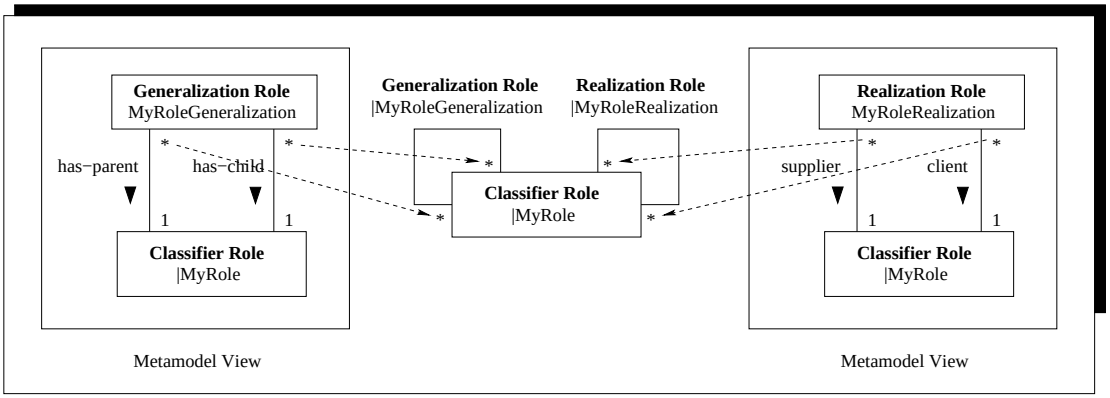



Figure 3.6: UML Metamodel View of MyRoleGeneralization and MyRoleRealization Roles

Fig. 3.6 shows the UML metamodel views of *MyRoleGeneralization* and *MyRole-Realization* roles.

A classifier in a conforming *MyRole* hierarchy can be an abstract class or an interface (i.e., a classifier plays *AbstractMyRole*) or it can be a concrete class (i.e., a class plays *ConcreteMyRole*).

Fig. 3.7(a) shows a structure that conforms to the role hierarchy model:

- *InterfaceA* is an interface that plays the *AbstractMyRole* role,
- *ClassA*, *ClassB*, and *ClassC* play the *ConcreteMyRole* role,
- the realization relationship between *InterfaceA* and its class realizations plays the *MyRoleRealization* role in the hierarchy, and
- the generalization relationship between *ClassB* and *classC* plays the *MyRole-Generalization* role.

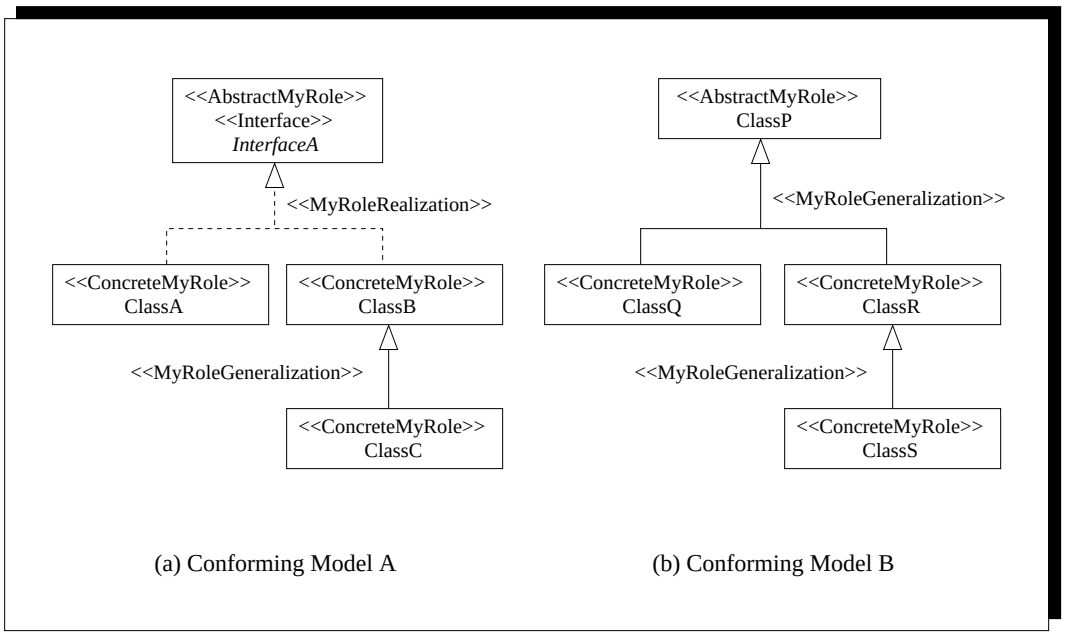



Figure 3.7: Conforming Models of the Role Hierarchy in Fig. 3.5

Fig. 3.7(b) is another conforming model that has a similar structure except that the generalization relationship between *ClassP* and its subclasses play the *MyRole-Generalization* role.

A relationship role is represented by a syntactic variant of the UML association symbol. Like classifier roles, each relationship role is associated with a label that indicates the base of the role. Association roles also have association-end roles that define subtypes of the UML metamodel *Property* class (see Section 2.2.2). Association-end roles specify multiplicity, navigability, and other properties associated with conforming association-ends. An association-end role is also associated with a realization multiplicity that can constrain the number of association-ends playing the role in a conforming model. The realization multiplicity for an association role can be inferred from the realization multiplicities of its association-end roles, and thus they are not shown in the SPSs presented in this documentation. An example of an association role is shown in Fig. 3.8.

Roles with realization multiplicities that have lower limits greater than 0 (e.g., 1..*) are referred to as *mandatory* roles. A conforming model must have models elements that conform to these roles. Both *Subject* and *Observer* in Fig. 3.8 are mandatory classifier roles. Roles that have realization multiplicities with lower limits that are 0 are referred to as *optional* roles. An SPS must have at least one mandatory role. If all SPS roles are optional then the SPS metamodel characterizes all valid UML class diagrams and thus is not a good discriminator.

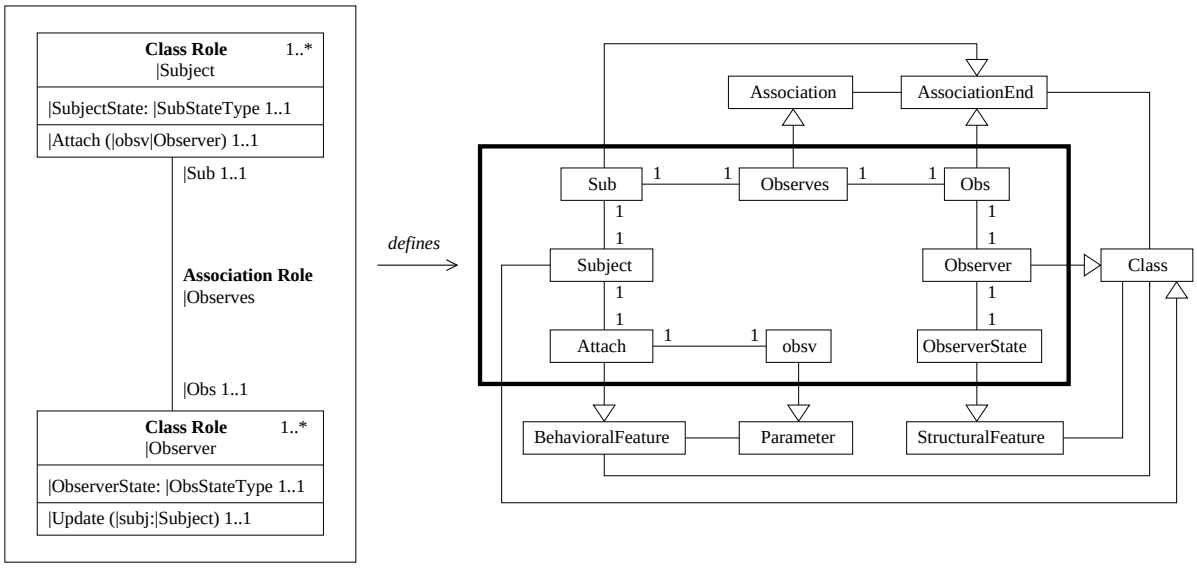
Well-formedness rules for the pattern metamodel that cannot be expressed in an SPS's role structure are expressed in the OCL. These constraints are called *metamodel-level constraints*. Examples of metamodel-level constraints are given in Section 3.2.1.5.

Semantic pattern properties are expressed as constraint templates in an SPS. For example, constraint templates are used to constrain the form of specifications for operations that conform to BehavioralFeature roles. Constraint templates are described in more detail in Section 3.2.1.6. Metamodel-level constraints and constraint templates are defined separately from the SPS role structure to avoid cluttering the diagram.

#### 3.2.1.5 An SPS Example

Fig. 3.8(a) shows a partial SPS that specifies solutions of a restrictive variant of the *Observer* pattern [35] (metamodel-level constraints, constraint templates, and some feature roles are not shown). In this variant of the pattern, there can be one or more observer classes and one or more subject classes. An observer class must have only one *Observes* association with a subject class and a subject class must have only one *Observes* association with an observer class.

The SPS in Fig. 3.8(a) consists of two class roles, *Subject* and *Observer*, and an association role, *Observes*. The roles define subtypes (specializations) of classes in




(a) Example of an SPS

(b) A Partial View of the Specialized UML Metamodel

Figure 3.8: Partial Views of an Observer Pattern SPS and its Metamodel

the UML metamodel, as shown in Fig. 3.8(b) (not all specializations are shown). For example, the *Observer* role defines a subtype of *Class* called *Observer* in the metamodel (see Fig. 3.8).

The class roles shown in Fig. 3.8 indicate that conforming class diagrams must have at least one class that conforms to the *Subject* role (as indicated by the $1..*$ realization multiplicity in the role), and at least one class that conforms to the *Observer* role. A class that conforms to the *Subject* role (referred to as a *Subject* class) must have exactly one structural feature (e.g., an attribute or query) that conforms to the *SubjectState* role and exactly one operation that conforms to the *Attach* role. A class that conforms to the *Observer* role must have exactly one structural feature that conforms to the *ObserverState* structural feature role, and one operation that plays the *Update* BehavioralFeature role.

The association role *Observes* specifies associations between *Subject* and *Observer* classes. The association role is expressed in an abbreviated form of the UML meta-

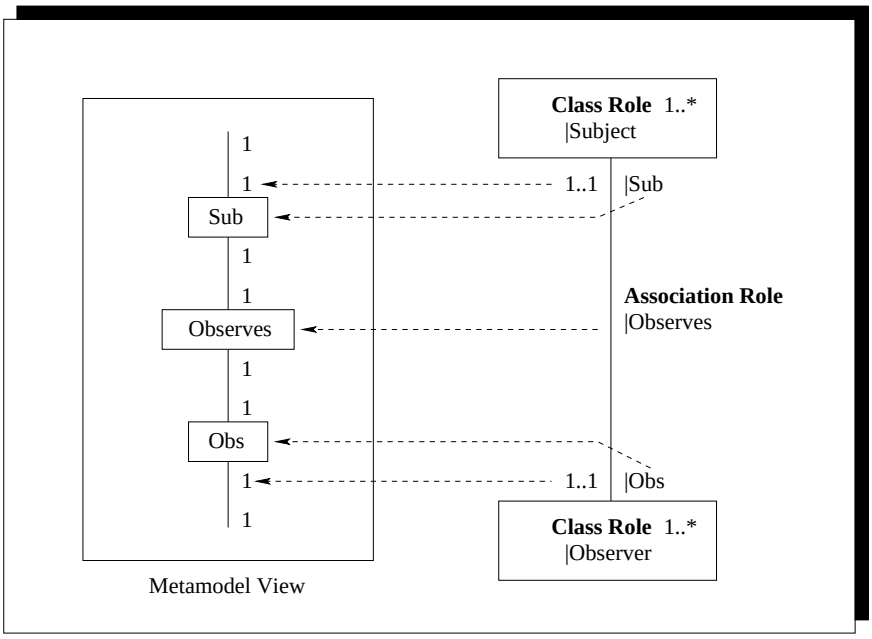



Figure 3.9: Association Role

model view as shown in Fig. 3.9. Each conforming association must have one association-end connected to a *Subject* class and the other association-end connected to an *Observer* class. In a conforming class diagram, the association-end connected to a *Subject* class must conform to the *Sub* role and the association-end connected to an *Observer* class must conform to the *Obs* role. The realization multiplicity on the *Sub* role specifies that a *Subject* class must be part of only one *Observes* association. Similarly, an *Observer* class must be part of only one *Observes* association.

Additional constraints on model elements that can play roles are expressed as metamodel-level constraints. For example, a constraint that restricts *Subject* classes to concrete classes is expressed in the OCL as follows:

**context** Subject **inv**: self.isAbstract = false

In the above, *Subject* is the *Class* subtype (subclass) defined by the role, *isAbstract* is an attribute inherited from the metamodel class *Class* and *self* refers to an instance

of the *Subject* subtype (i.e., a *Subject* class). A similar constraint is associated with the *Observer* role.

Relationship roles and association-end roles can also be associated with metamodel-level constraints. The following are some of the constraints associated with the *Sub* and *Obs* association-end roles in the *Observer* pattern:

- An association-end that conforms to *Sub* must have a multiplicity of 1..1:

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

- An association-end that conforms to *Obs* must have a multiplicity of 0 or more (∗):

  **context** |Obs **inv**: self.lowerBound() = 0 and self.upperBound() = *

Class diagrams that conform to the above constraints describe an observer system in which subjects can attach themselves to zero or more observers, and an observer is restricted to monitoring only one subject.

#### 3.2.1.6 Specifying Semantic Pattern Properties in an SPS

The role structure and metamodel-level constraints of an SPS determine the syntactic structure of conforming class diagrams. A pattern also describes semantic properties. For example, an operation that plays the *Attach* feature role must have a behavior in which the observer passed in as an argument to the operation is linked to the subject. These semantic properties are specified by constraint templates in a pattern specification. A constraint template is an OCL constraint expressed in terms of roles.

Constraint templates that are associated with BehavioralFeature roles constrain the contents of specifications associated with conforming operations. These templates are called *operation templates*. An operation template for the *Attach* BehavioralFeature role is given below:

**context** |Subject::|Attach(|obsv:|Observer)

**pre**: self.|Obs → excludes(|obsv)

**post**: self.|Obs = self.|Obs@pre → including(|obsv)

The *Attach* operation template specifies behaviors that attach observer objects to subject objects. The postcondition states that the observer object is attached. The expression *x@pre* in a postcondition refers to the value of $x$ before execution of the operation, and thus *self.|Obs@pre → including(|obsv)* states that the observer parameter playing the *obsv* role is added to the set of observers associated with the subject.

The *Subject* role is also associated with the following BehavioralFeature roles (these roles are not shown in Fig. 3.8(a)):

- **Detach** specifies behaviors that remove observers from subjects.
- **SetState** specifies behaviors that set the subject state.
- **Notify** specifies behaviors that notify observers whenever a change in the subject state occurs.
- **GetState** specifies behaviors that return the subject state.

The operation templates for the *Detach*, *GetState*, and *SetState* roles are given below. The $Notify$ feature role is not associated with an operation template (i.e., it does not restrict the form of pre- and postconditions associated with conforming operations). The operation template associated with the *Update* feature role in *Observer* is given below:

**context** |Observer::|Update(|subj:|Subject)

**pre**: true

**post**: |ObserverState = |Function (|subj.|SubjectState)

The above template specifies behaviors in which the state attribute of an observer is

updated with a value that is a function of a subject state attribute. The function is defined by the developer and plays the *Function* role. The identity function is used in the cases where the subject state is assigned to the observer state.

Constraint templates can also be used to specify invariant properties in a UML model. These templates are referred to as *property templates*. For example, a property template that specifies a semantic relationship between structural features playing the *SubjectState* and the *ObserverState* roles is given below:

**context** |Subject **inv**:

|Obs → forAll(|ObserverState = |Function (|SubjectState))

The presence of the above template in an *Observer* SPS requires that conforming class diagrams have a constraint stating that each observer attached to a subject must have a state value that is a function of the subject's state value. If the observer state must be the same as the subject state then the identity function plays the role of *Function*.

### 3.2.2 Establishing Structural Conformance to an SPS

A class diagram *structurally conforms to* an SPS, with respect to a binding of model elements to roles, if it satisfies (1) the structural constraints specified by the SPS role structure and (2) the metamodel-level constraints. The following checks are carried out when establishing that a class diagram structurally conforms to an SPS with respect to a given binding:

- Realization multiplicity check: Check that the number of classifiers bound to a classifier role satisfy the realization multiplicities associated with the role, and check that mandatory roles have classifiers bound to them.

- Structural conformance check: For each classifier bound to a classifier role this requires establishing that (1) the classifier satisfies the metamodel-level con-

straints associated with the classifier role, (2) the features bound to feature roles in the classifier role satisfy the realization multiplicities of the feature roles, and that (3) the mandatory feature roles have features bound to them.

- Relationship conformance check: Check that relationships bound to relationship roles satisfy metamodel-level constraints associated with the roles and that the relationships have ends attached to classifiers that conform to the roles at the ends of the relationship roles. For an association role, bound associations must have association-ends that conform to the association-end roles and to metamodel-level constraints associated with the association-end roles.

A class diagram that structurally conforms to the *Observer* pattern SPS, with respect to a binding, is shown in Fig. 3.10(a). The bindings are indicated by the

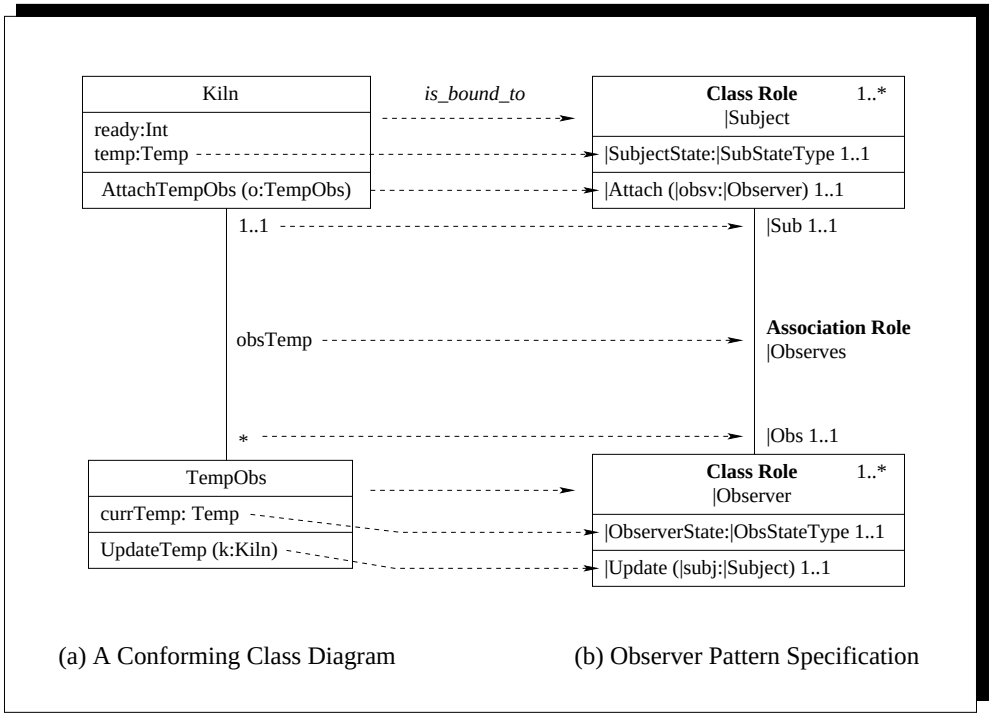



Figure 3.10: A Structurally Conforming Observer Class Diagram

dashed lines between the class diagram and the SPS in Fig. 3.10 (e.g., *Kiln* is bound

to the *Subject* role). The class *Kiln* describes kiln objects whose temperatures are monitored by *TempObs* objects.

A partial view of a less restrictive variant of the *Observer* pattern and a conforming class diagram are shown in Fig. 3.11. The SPS shown in Fig. 3.11(b) specifies

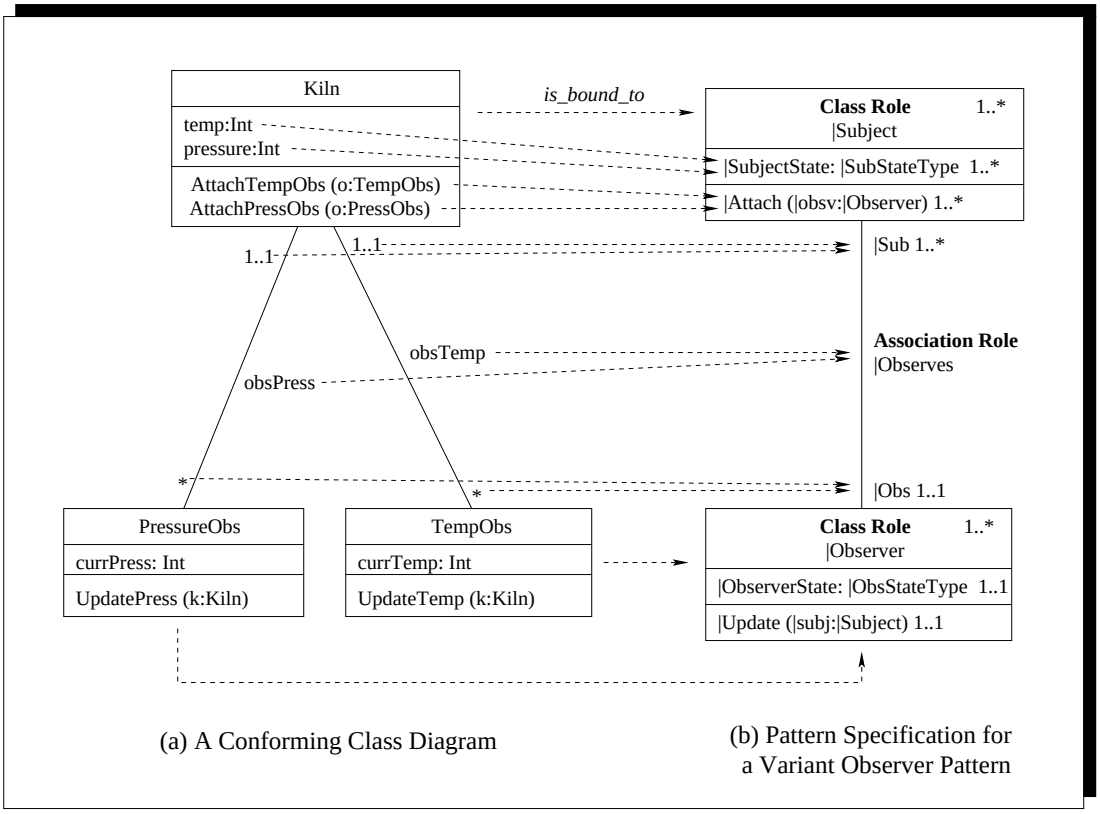



Figure 3.11: A Partial SPS for a Variant of the Observer Pattern and a Conforming Class Diagram

class diagrams in which *Subject* classes can have one or more structural features that can be monitored and can be part of one or more associations connected to *Observer* classes. The *Observer* pattern variant shown in Fig. 3.8 is a specialization of this less restrictive pattern variant, that is, the set of class diagrams characterized by the SPS in Fig. 3.8 is a proper subset of the set of class diagrams characterized by the SPS shown in Fig. 3.11(b).

### 3.2.3 Establishing Full Conformance to an SPS

A class diagram fully conforms to an SPS, with respect to a binding of model elements to roles, if (1) it structurally conforms to the SPS, and (2) the semantic properties expressed by constraints in the class diagrams (e.g., operation specifications and class invariants) conform to the constraint templates in the SPS. Establishing that the semantic properties expressed in a class diagram conform to constraint templates in an SPS involves (1) instantiating the constraint templates using the role bindings, and (2) establishing that the constraints given in the class diagram refine the instantiations of the constraint templates.

The result of instantiating a constraint template is an application-specific OCL expression of the properties described by the constraint template. For example, instantiating the property template given in Section 3.2.1.6 using the binding shown in Fig. 3.10 results in the following constraint:

**context** Kiln **inv**:

obsTemp → forAll(currTemp = temp)

The identity function plays the role of *Function* in the property template. The class diagram shown in Fig. 3.10 must have a constraint that implies the above instantiation if it is to fully conform to the *Observer* SPS. In general, a class diagram that fully conforms to an SPS containing property templates must have constraints that imply instantiations of the property templates.

Instantiating the *Attach* operation template using the binding shown in Fig. 3.10 produces the following:

**context** Kiln::AttachTempObs(tobs: TempObs)

**pre**: self.TempObs → excludes(tobs)

**post**: self.TempObs = self.TempObs@pre → including(tobs)

Establishing that an operation specification conforms to an RBML operation template involves proving that the operation specification refines the operation template instantiation. Given an operation *Op* with pre- and postconditions

**context** Op(...): **pre**: preR; **post**: postR,

and an instantiated operation of an RBML operation template for a feature role *ROp*

**context** Op(...): **pre**: preM; **post**: postM,

*Op* is said to fully conform to *ROp* (with respect to the binding used to produce the instantiation) if (1) $preM \Rightarrow preR$, and (2) ($preM$ and $postR$) $\Rightarrow postM$.

These proof obligations must be discharged before one can assert that an operation fully conforms to a BehavioralFeature role.

As an example, consider the following pre- and postcondition for the *AttachTempObs* operation shown in Fig. 3.10:

**context** Kiln::AttachTempObs(tobs: TempObs)
**pre**: self.TempObs → excludes(tobs)
**post**: self.TempObs = self.TempObs@pre → including(tobs) and
ready = ready@pre + 1

The preconditions for *AttachTempObs* and the instantiation of the *Attach* constraint template are equivalent so only the second operation proof obligation needs to be discharged:

self.TempObs → excludes(tobs) and
self.TempObs = self.TempObs@pre → including(tobs) and
ready = ready@pre + 1 ⇒ self.TempObs = self.TempObs@pre → including(tobs)

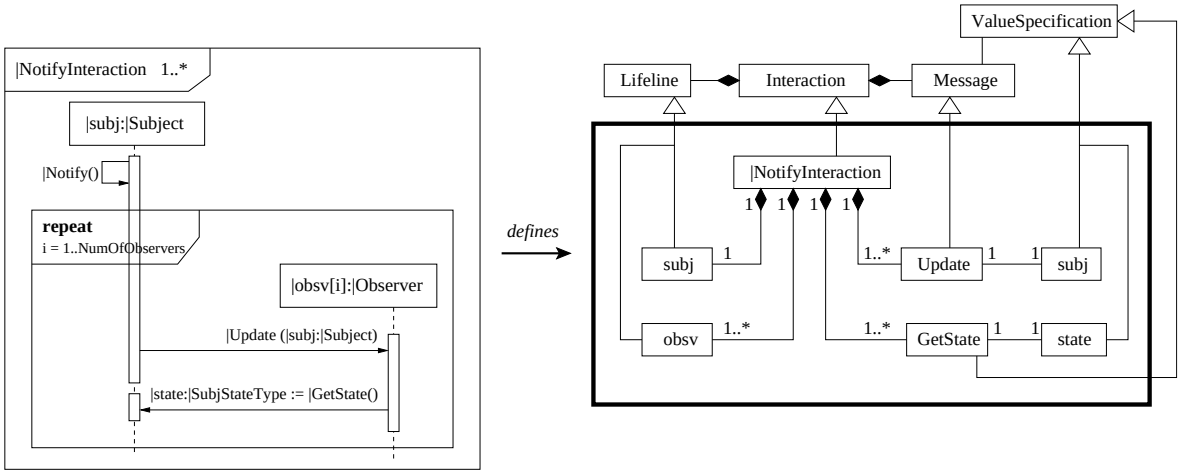
Automated support for structural conformance checking is possible: mechanisms that check conformance of UML models to the abstract syntax defined by the UML meta-

model can be extended to support well-formedness checks for patterns as defined by SPSs. Tools that can mechanically discharge most proof obligations are not likely to appear in the near future, but it is possible to build a tool that generates proof obligations that can then be discharged by humans.

### 3.2.4 Interaction Pattern Specifications (IPSs)

An Interaction Pattern Specification (IPS) describes a pattern of interactions and is defined in terms of roles defined in an SPS. The SPS roles are used to specify participants in an interaction pattern. Formally, an IPS defines a part of the pattern metamodel that specifies conforming interaction diagrams.

Fig. 3.12(a) shows an IPS that describes the pattern of interactions between a subject and its observers initiated by the invocation of the subject's $Notify$ operation.




(a) Example of an IPS

(b) A Partial View of the Specialized UML Metamodel

Figure 3.12: An IPS for the Observer Pattern and a Partial View of its Specialized UML Metamodel

The expression $|subj : |Subject$ indicates that the lifeline role $subj$ is played by an instance of a $Subject$ class (i.e., a class that conforms to the $Subject$ role defined in the $Observer$ SPS). The lifeline role $obsv[i]$ is played by the $i^{th}$ observer in the set

of observers attached to the subject playing the *subj* role. The **repeat** fragment in the IPS indicates that the enclosed interaction is repeated for each observer attached to the subject playing the *subj* role. $NumOfObservers$ is the number of observers attached to the subject. The **repeat** fragment is used to concisely represent parts of conforming interaction diagrams that have a common structure.

The IPS describes the following interaction pattern:

- Invocation of a subject's $Notify$ operation (i.e., an operation that conforms to the $Notify$ feature role) results in calls to the $Update$ operation in each observer linked to the subject.
- Each $Update$ operation calls the $GetState$ operation in the subject.

An IPS consists of an *interaction role* that defines a specialization of the UML metamodel class $Interaction$. In the UML 2.0 an interaction is a structure of lifelines and messages. Consequently, an interaction role is a structure of *lifeline* and *message* roles. Each lifeline role is associated with a classifier role: a participant that plays a lifeline role is an instance of a classifier that conforms to the classifier role.

In this research, the messages that represent operation calls are focused. A message role is associated with a BehavioralFeature role: a conforming message specifies a call to an operation that conforms to the BehavioralFeature role. For example, the $Update$ message role is associated with the feature role $Update$.

Part of the metamodel defined by the $NotifyInteraction$ IPS is given in Fig. 3.12(b). Lifeline roles define specializations of the $Lifeline$ class and message roles define specializations of $Message$.

### 3.2.5 Establishing Interaction Conformance to an IPS

Roles in an IPS refer to roles in an SPS. For example, a lifeline role and a message role in an IPS refer to a classifier role and behavioral role in an SPS, respectively.

Therefore, before checking conformance of a sequence diagram to an IPS, there must be a class diagram that conforms to an SPS whose roles are referred in the IPS. The following checks are carried out when establishing that a sequence diagram conforms to an IPS with respect to a binding:

- Lifeline conformance check:
  - Check that for each object $o$ bound to a lifeline role $o{:}C$, the object must be an instance of the class bound to $C$.
  - Check that for each object bound to a lifeline role, constraints associated with the object must satisfy the constraint templates associated with the role.
- Message conformance check:
  - Check that for each message $m$ bound to a message role $M$ directed towards a lifeline role $o{:}C$, the class bound to $C$ must have the operation $m$.
  - Check that for each message bound to a message role, the message satisfies the metamodel-level constraints defined for the message role.
  - Check that for each message bound to a message role, constraints associated with the message must satisfy the constraint templates associated with the message role.
  - Check that the relative sequence of the bound messages in the sequence diagram respect the order specified in the IPS.

In a conforming sequence diagram, additional application-specific participants and messages may exist.

Fig. 3.13 shows an example of a conforming sequence diagram to the $NotifyInteraction$ IPS. The sequence diagram has the following binding: ($s{:}Kiln \mapsto$

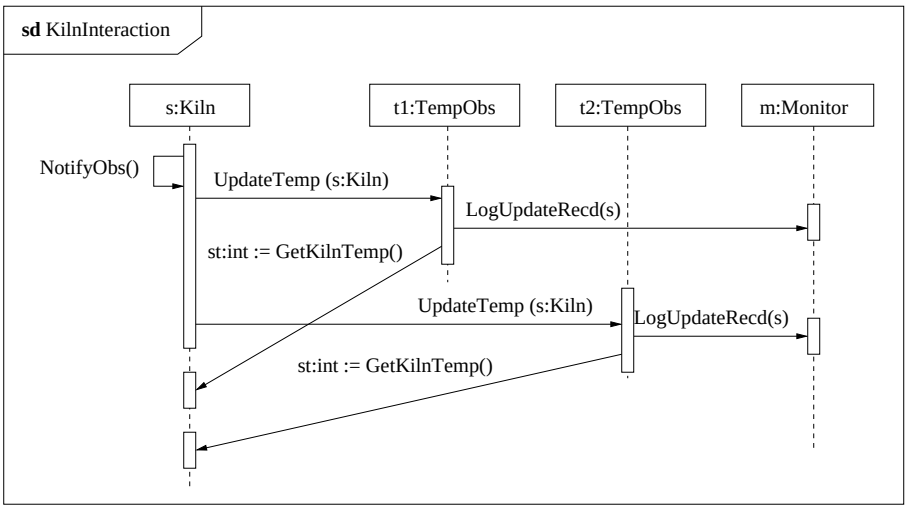



Figure 3.13: A Sequence Diagram that conforms to the Observer IPS

*subj:Subject*), (*t1:TempObs* $\mapsto$ *obsv[1]:Observer*), (*t2:TempObs* $\mapsto$ *obsv[2]:Observer*), ($NotifyObs \mapsto Notify$), ($UpdateTemp \mapsto Update$), ($GetKilnTemp \mapsto GetState$). The *m:Monitor* and *LogUpdateRecd* are application-specific elements. Given the binding, the sequence diagram satisfies the conformance rules as follows:

- The *s*, *t1*, and *t2* are instances of *Kiln* and *TempObs* in the class diagram in Fig. 3.10 (a) that is a base class diagram of the sequence diagram.

- The operations *NotifyObs* and *GetKilnTemp* are defined in the *Kiln* class and the *UpdateTemp* is defined in the *TempObs* Fig. 3.10 (a) (*NotifyObs* and *GetKilnTemp* are not shown).

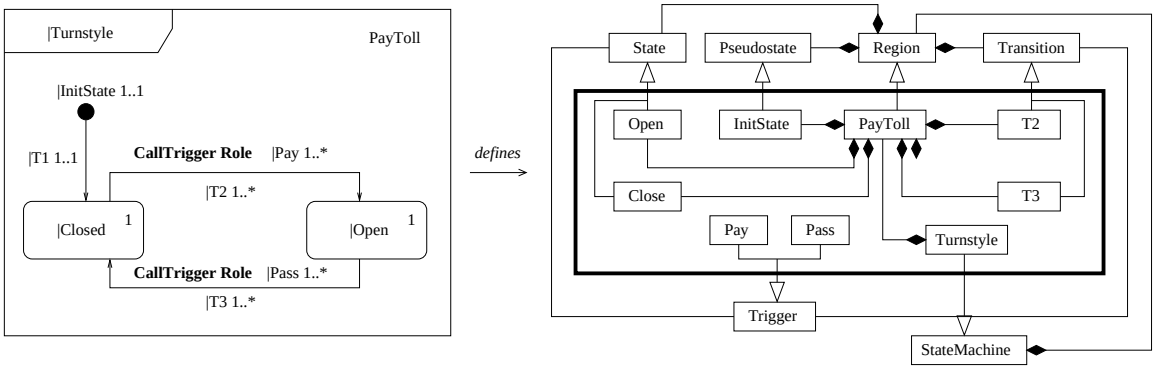
- The message *NotifyObs*, *UpdateTemp*, and *GetKilnTemp* are synchronous, which satisfies the "synchronous" metamodel-level constraints (denoted by filled arrow head) for the messages roles in Fig. 3.12. The repeated message groups of *UdpateTemp* and *GetKilnTemp* satisfy the *repeat* metamodel-level constraint with the variable *NumOfObservers* set 2.

- The relative order of the conforming messages, $NotifyObs$, $UpdateTemp$, and

*GetKilnTemp* is the same as the relative order specified in the IPS.

### 3.2.6 Statemachine Pattern Specifications (SMPSs)

SMPSs specify a UML statemchart diagram view of pattern solutions describing local behavior of participants. An SMPS consists of *state*, *transition*, and *trigger* roles whose bases are State, Transition, and Trigger metaclasses, respectively. The work on SMPSs is immature. In this section, only initial work is presented.




(a) Example of an SMPS

(b) The Specialized UML Metamodel

Figure 3.14: An SMPS and a Partial View of its Specialized UML Metamodel

Fig. 3.14(a) shows an SMPS for turnstyle systems, for example subway or library turnstyles. The SMPS in Fig. 3.14(a) describes the following:

- When an object of a conforming classifier of *Turnstyle* role is created the state of the object moves from an initial state to a state of *Closed*.
- When the object in a state of *Closed* receives a call trigger of *Pay*, the state of the object changes to a state of *Open*.
- The object in *Open* state moves back to a *Closed* state when a *Pass* event is triggered.

- The multiplicities on *Pay* and *Pass* trigger roles and *T2* and *T3* transition roles constrain that there should be at least one or more model elements that play the roles.

The multiplicities specify the number of model elements that can play the roles. For example, the multiplicy "1" on *Closed* constraints that a conforming statechart must have exactly one state that plays the role. The metamodel-level constraints for the turnstyle SMPS are defined as follows:

- Conforming state machines of *Turnstyle* role may be extended:

  **context** |Turnstyle **inv**:

  self.isFinal = false

- Conforming trantisions of *T1* role may be extended:

  **context** |T1 **inv**:

  self.isFinal = false

  Similar constraints are defined for *T2* and *T3* roles.

- States playing *Closed* and *Open* must be simple states and may be extended:

  **context** |Closed **inv**:

  isSimple = true and self.isFinal = false

  Similar constraints are defined for *Open* role.

- States playing *InitState* must be initial states:

  **context** |Inistate **inv**:

  self.kind = #initial

Part of the metamodel defined by the *Turnstyle* SMPS is given in Fig. 3.14(b). In the diagram, state roles define specializations of the *State* class and transition roles define specializations of *Transition*.

Trigger roles in an SMPS refer to operations roles in an SPS. In Fig. 3.14 (a), triggers that play *Pay* role must be call triggers activated by a *MakePayment* operation call. This is specified in the following constraint template:

**context** |Pay **inv**:

self.operation.oclIsKindOf(|MakePayment)

A similar constraint template is defined for *Pass* role.

### 3.2.7 Establishing Statechart Conformance to an SMPS

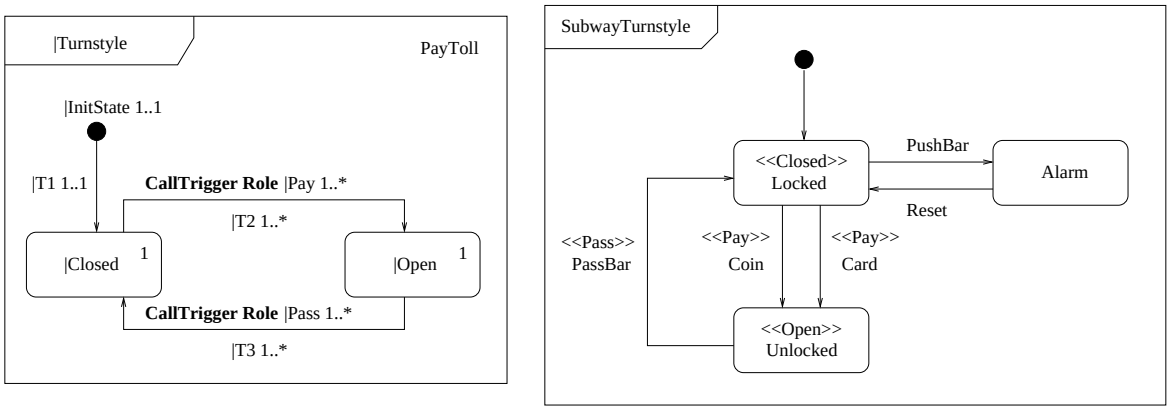
Like IPSs, SMPSs refer to SPS roles, for example, a trigger role in an SMPS refer to a behavioral feature role in the classifier role in an SPS for which the SMPS is defined. Therefore, before checking conformance of a statechart diagram to an SMPS, there must be a class diagram that conforms to the SPS whose roles are referred in the SMPS. The following checks are carried out to establish conformance of a statechart diagram to an SMPS with respect to a given binding:

- Realization multiplicity check: Check that model elements bound to roles (state, trigger, transition roles) satisfy the realization multiplicities defined in the roles.
- State conformance check: For each state bound to a state role, check that the state satisfies the metamodel-level constraints associated with the state role.
- Transition conformance check: For each transition bound to a transition role,
  - Check that the transition satisfies the metamodel-level constraints defined for the transition role, and
  - Check that the transition has source and target states that are bound to the source and target state roles of the transition role.
- Trigger conformance check: For each trigger bound to a trigger role,

– Check that the trigger satisfies the metamodel-level constraints defined for the trigger role, and

– Check that the trigger constraints for the trigger role imply the constraint instantiated from the trigger constraint template associated with the trigger role.

In a conforming statechart, application-specific elements may exist as long as the conformance rules are satisfied.




(a) An SMPS

(b) A Conforming Statechart Diagram

Figure 3.15: An SMPS and a Conforming Statechart Diagram

Fig. 3.15(b) shows a conforming statechart of the turnstyle SMPS. The statechart describes a turnstyle that takes coins and cards for payment and triggers an alarm for illegal passing. The following constraints are defined for the *Coin*, *Card*, and *PassBar* triggers:

- The *Coin* trigger is activated by a *MakePaymentByCoin* operation call:

  **context** Coin **inv**:

  self.operation.oclIsKindOf(MakePaymentByCoin)

- The *Card* trigger is activated by a *MakePaymentByCard* operation call:

**context** Card **inv**:

self.operation.oclIsKindOf(MakePaymentByCard)

- The *PassBar* trigger is activated by a *PassBar* operation call:

  **context** PassBar **inv**:

  self.operation.oclIsKindOf(PassBar)

The statechart has the following binding: $(Locked \mapsto Closed)$, $(Unlocked \mapsto Open)$, $(Coin \mapsto Pay)$, $(Card \mapsto Pay)$, $(PassBar \mapsto Pass)$. The *Alarm*, *PassBar*, and *Reset* are application-specific elements. Given the binding, the statechart conforms to the turnstyle SMPS as follows:

- The given binding satisfies the realization multiplicities in the SMPS.
- The states *Locked* and *Unlocked* are simple states and can be extended, which satisfy the metamodel-level constraints *isSimple = true* and *isFinal = false* [2].
- The transitions from the *Locked* to *Unlocked* satisfy the metamodel-level constraint *isFinal = false* defined for *T2*. Similar interpretations are made for the transition from the *Unlocked* to *Locked* and the transition from the initial state to *Lockeded*.
- The transitions bound to the *T2* role have the source state *Locked* and target state *Unlocked* that are bound to the source state role *Closed* and target state role *Open* of the *T2* role. Similar interpretations are made for the transitions bound to the *T1* and *T3*.

---

[2]The version of the UML 2.0 used in this work does not have notations for states and transitions whether isFinal is true or false, and the default is not mentioned. In this work, the default of isFinal is assumed to be false unless noted.

- An instantiation of the constraint template associated with the *Pay* role is given below:

  **context** Coin **inv**:

  self.operation.oclIsKindOf(MakePaymentByCoin)

  The instantiation is identical to the constraint associated with the *Coin* trigger (i.e., the constraint satisfies the constraint template). Similar interpretations are made for the *Card* and *PassBar* triggers.

An SMPS can be viewed as determining a constrained form of the UML metamodel. Fig. 3.16 describes the relationship between an SMPS and the UML metamodel. In the figure, an SMPS *MySMPS* to which a state machine *MyGeneralStateMachine* conforms, defines a subset of instances of UML *StateMachine* metamodel class whose instances are *GeneralStateMachine*, *MyGeneralStateMachine*, and *MySpecialStateMachine*.

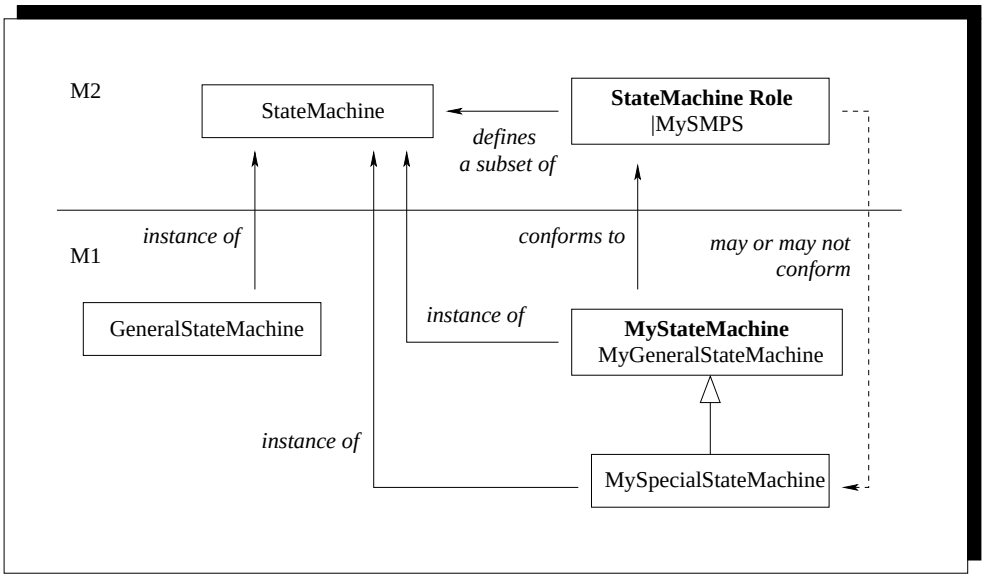



Figure 3.16: An SMPS Role and the UML metamodel

*MySpecialStateMachine* is a specialization of *MyGeneralStateMachine*. A state machine can be specialized by 1) adding regions, states, and transitions, 2) extending

regions and states, or 3) redefining (replacing) transitions [100]. An extension of a region and a state occurs when new states and/or transitions are added (e.g., a simple state becomes a composite state). A redefinition of a transition occurs when the target state of the transition is changed.

Note that in Fig. 3.16, the *MySpecialStateMachine may* or *may not* conform to *MySMPS* role. If *MySpecialStateMachine* has a transition that is redefined to have a different target than in *MyGeneralStateMachine*, then *MySpecialStateMachine* does not conform to *MySMPS*. Therefore, given a statemachine that conforms to an SMPS, it is possible to have non-conforming specializations of the state machine to the SMPS.

In order for *MySpecialStateMachine* to be a conforming state machine of *MySMPS*, the *isFinal* meta attribute in UML meta classes must be set to true for all roles in MySMPS whose conforming model elements are generalizable (e.g., states, transitions, events). This constraint disallows the cases of specialization described above. In spite of this restriction, state machines can still be extended in a limited, but conforming way (e.g., by adding new transitions) if *isFinal* is set to false for MySMPS itself.

## 3.3 Summary

We have developed the RBML, a pattern specification language, that specifies patterns as specializations of the UML metamodel to support the use of patterns in UML system modeling. The RBML uses the UML as the syntactic base for the pattern specification language to make it easier for UML modelers to create, understand, and evolve pattern specifications, and to enable the use of UML modeling tools for creating and evolving pattern specifications.

An RBML pattern specification comprises abstract syntax, metamodel-level constraints, and constraint templates. Abstract syntax defines the structure of pattern solutions using the notion of roles. A role whose its base is a UML metaclass defines a subclass of the base metaclass by defining additional constraints to the metaclass

called metamodel-level constraints expressed in the OCL. Constraint templates expressed in parameterized OCL define semantics of pattern solutions and can be used to generate model constraints for conforming models. Three types of pattern specification are developed:

- Static Pattern Specifications (SPSs): An SPS is a class diagram view of pattern solution. An SPS consists of classifier roles and relationship roles. A classifier role defines structural and behavioral properties of pattern participants and a relationship role defines a relationship between these participants. If an instance of the metaclass on which a role is based has properties as defined in the role, then the instance is said to conform to the role.

- Interaction Pattern Specifications (IPSs): An IPS is an interaction diagram view of pattern solution. An IPS comprises lifeline roles and message roles describing the sequence of interactions between pattern participants. A sequence diagram is said to conform to an IPS if the sequence diagram preserves the relative order of the message sequence as defined in the IPS.

- Statemachine Pattern Specifications (SMPSs): An SMPS is a statemachine diagram view of pattern solutions. An SMPS consists of state roles and transition roles describing the sequence of transitions between states.

## 3.4 Discussion

The pattern specification technique described in this chapter can be used as a base for the development of tools that support creation and evolution of patterns, and rigorous application of design patterns to UML models. The tool-independent UML-based notation makes it easier to share design patterns across UML modeling tools. The work presented in this chapter has been published [32, 33, 59, 60]. SPSs, IPSs,

and their conformance rules are described [32, 33], SMPSs are described [59], and the notion of pattern role is described [60].

SMPSs need to be further developed to expand their applicability. The work presented in this chapter only considers state, trigger, and transition roles. The notion of activity roles needs to be defined to support specifying actions, for example states with actions or triggers with actions. Notation and conformance rules for composite state roles need to be developed. For example, composite state roles can be used for patterns in the domain of component-based systems to capture state hierarchies [46]. Checking consistency between SMPSs and IPSs also needs to be investigated. For example, composite state roles can be used for patterns in the domain of component-based systems to capture state hierarchies [46]. Checking consistency between SMPSs and IPSs also needs to be investigated. For example, a trigger role and an activity role in an SMPS can respectively refer to an incoming message role on a lifeline role and an outgoing message role in an IPS. Consistency checking involves ensuring that the referred roles exist in an IPS. So far we have used SMPSs to specify the Iterator pattern only, and more study is needed to improve applicability of SMPSs.

Given an RBML pattern specification, one can loosen the constraints to include more models as conforming models (generalization of an RBML specification) or tighten the constraints to exclude some models as conforming models (specialization of an RBML specification). An RBML specification is a specialization (child) of another (parent) RBML specification if it further restricts the properties specified in the parent specification. A specialization of an RBML specification characterizes a subset of conforming models of its parent. An SPS can be specialized by:

- specializing SPS roles,
- further restricting the multiplicities on role associations,
- reducing the number of alternatives by removing alternative structures and

tightening the constraints (e.g., association and realization multiplicities) to allow only the remaining alternatives (e.g., a SPS that has generalization and $\ll realize \gg$ relationship roles as alternative relationships can be specialized by removing the alternative generalization relationship role and allowing only $\ll realize \gg$ relationships), and

- by adding new roles and associations to the SPS that must be realized (i.e., requiring additional structure in realizations).

Examples of specializations of an Abstract Factory SPS are shown in Section 4.2. IPS and SMPSs depend on an SPS. Thus, specializing an SPS would result in specializing associated IPSs and SMPSs. For example, specializing all classifier roles to interface roles in an SPS results in interacting participants in associated IPSs to be interfaces. This can be useful in Cheesman and Daniels approach [19] for component-based software development (CBSD) where interactions between components are specified using their interfaces. Specializing a classifier role in terms of behavioral feature roles would result in specializing associated SMPSs in terms of trigger roles activated by the behavioral roles. Specializing IPSs and SMPSs needs further investiation.

It is possible in some cases to specialize a pattern specification to the point that variations in the designs can be expressed as parameters. Then the pattern specification becomes a template. The resulting templates pave the way for automated generation of initial designs from patterns. In this research RBML templates are used to specify access control aspects in Chapter 6.

Specifying pattern solutions at the UML metamodel level allows tool developers to build support for creating patterns and for checking conformance to pattern specifications. Pattern solutions can be specified at the UML metamodel level through interfaces that allow developers to access and specialize a tool's internal representation of the UML metamodel. Using interfaces does not have to require direct modification

of the internal metamodel: the specializations can be created and managed by a layer that sits on top of the UML metamodel layer in the tool. A new generation of UML tools that allow software developers to specialize the UML metamodel in limited ways are emerging (e.g., IBM Rational XDE). These tools are expected to mature to the point where users can define pattern by specializing the metamodel as described in this chapter.

We have developed a prototype tool, RBML Pattern Instantiator (RBML-PI), that generates conforming class diagrams and sequence diagrams from SPSs and IPSs. RBML-PI generates various structures of conforming class diagrams and sequence diagrams based on the input (e.g., association multiplicities) from developer, A demonstration of the tool is presented in Chapter 7.1.

The popularity of the UML and the heightened interest in model-driven approaches to software development has raised interest of researchers in model transformations. Techniques and tools that support systematic and rigorous application of design patterns through model transformations can ease access to and reuse of design experience during software development. The Software Engineering group at Colorado State University uses the RBML to support pattern-based model transformation techniques [31, 52, 96].

# Chapter 4

# Using the RBML to Specifying Design Patterns

This chapter describes specifications of design patterns in Gamma *et al.* [35] using the RBML. The specifications presented in this chapter are based on our interpretation of the descriptions given in the GoF patterns. The patterns specified in this chapter are Visitor, Abstract Factory, and Iterator patterns. These patterns are chosen because the Visitor pattern has significant interaction behavior, the Abstract Factory pattern possesses significant structural properties, and the Iterator pattern has localized behavior to specify. Specifications for other patterns - Observer, Composite, Bridge, Decorator, State, and Adapter patterns - are presented in Appendix A. The work presented in this chapter has been published [31, 32, 59].

This chapter is organized as follows. Section 4.1 shows specifications for the Visitor pattern solutions. Section 4.2 shows specifications for the Abstract Factory pattern solutions. Section 4.3 shows specifications for the Iterator pattern solutions. Section 4.4 describes a validation of the RBML based on the feedback from the experience of the RBML in a software engineering course.

## 4.1 Specifying the Visitor Pattern

The Visitor design pattern improves understandability and maintainability of a system where operations are distributed over the structure of classes (elements) in the system by separating related operations from the elements and packaging them into an object called *visitor*. When an element wants to perform the operation defined in a visitor, the element sends a request to the visitor, and then the visitor sends a request back to the element for acceptance of the visitor to perform the requested operation on the element. Such a separation facilitates defining new operations over the structure when the classes defining the structure rarely change. The Visitor pattern enables a separation of abstraction from implementation for visitors and elements to make it easy to change them without changing clients.

### 4.1.1 Pattern Description by Gamma *et al.*

A class diagram and a sequence diagram describing a typical Visitor pattern solution are respectively shown in Fig. 4.1 and Fig. 4.2. This solution is used by Gamma *et al.* [35] to describe the structure and behavior of Visitor pattern solutions. The model describes a solution consisting of two types of visitors, $ConcreteVisitor1$ and $ConcreteVisitor2$, whose instances visit elements in an element collection (instances of *ObjectStructure*) consisting of two types of elements, $ConcreteElementA$ and $ConcreteElementB$.

The sequence diagram shown in Fig. 4.2 describes a typical interaction in which the *anObjectStructure* object (an instance of *ObjectStructure*) calls the *Accept* operation for each of its elements. The element collection consists of two elements - $aConcreteElementA$ is an instance of $ConcreteElementA$ and $aConcreteElementB$ is an instance of $ConcreteElementB$. Execution of the *Accept* operation in an element results in an operation call to the visitor passed in as an argument of the *Accept* operation. The visitor then performs an operation on the element.

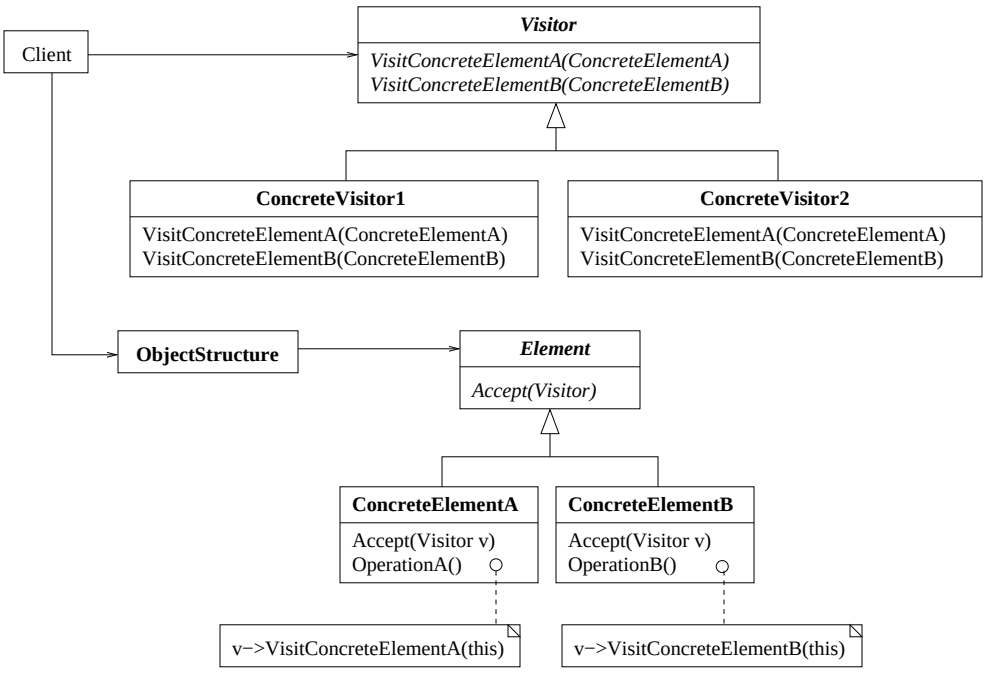



Figure 4.1: A Visitor Pattern Solution: Class Diagram

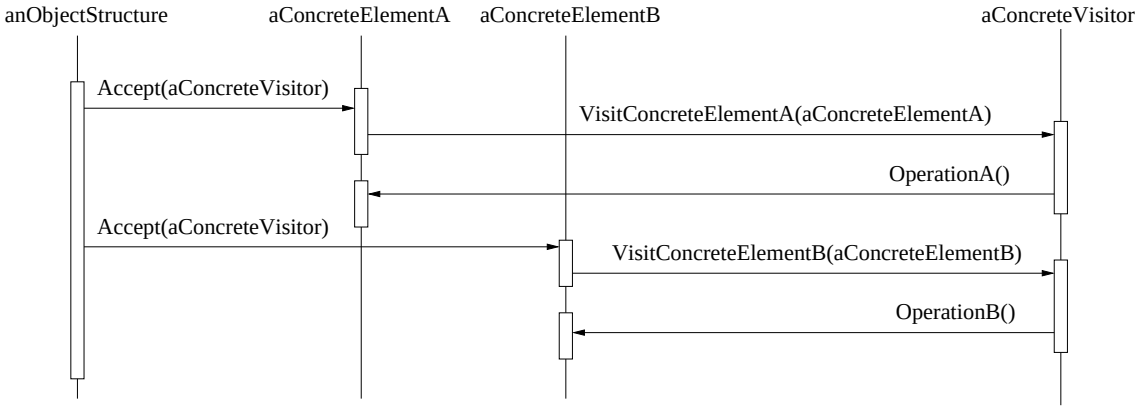



Figure 4.2: A Visitor Pattern Solution: A Sequence Diagram

A pattern specification for a variant of the Visitor pattern described by Gamma *et al.* [35] is presented in this subsection. It characterizes solution models including those involving flat sets of elements such as the one described above, and more complex solutions that involve composite element structures.

### 4.1.2 The Visitor SPS

The class diagrams characterized by a Visitor SPS possess visitor and element classifiers that are abstract or concrete, and an object structure class associated with *Element* classifiers. Among the classifiers, there are at least one concrete visitor class, at least one concrete element class, and at least one object structure class.

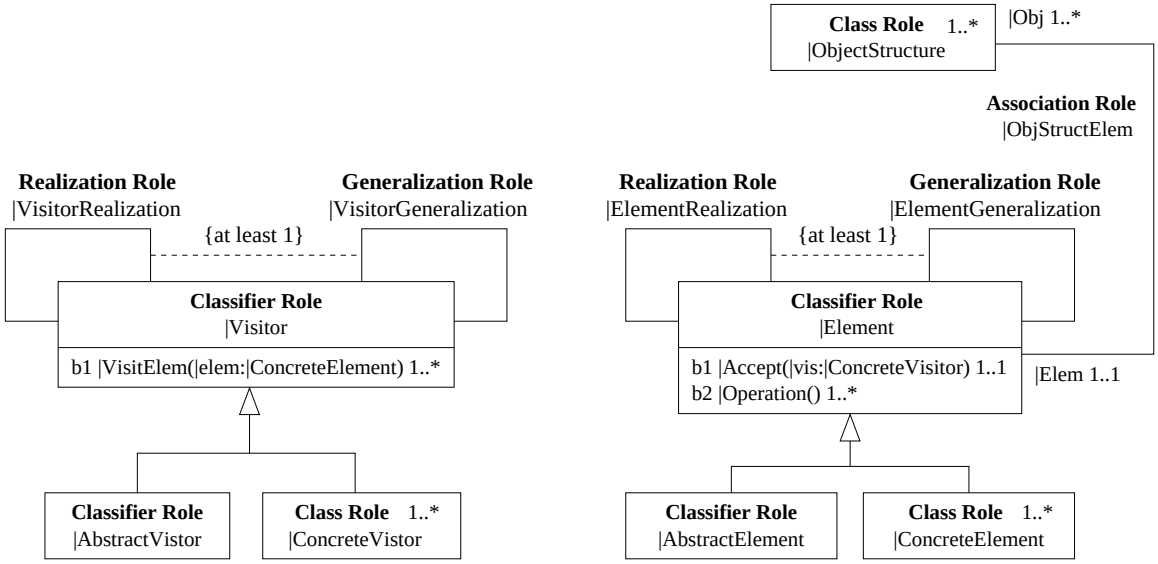



Figure 4.3: A Visitor SPS

Fig. 4.3 shows an SPS for the Visitor pattern. The SPS consists of $Visitor$ and *Element* role hierarchies and *ObjectStructure* class role. The $Visitor$ hierarchy specifies that there must be one or more concrete classes (specified by the multiplicity 1..*) that play $ConcreteVisitor$ role and may be abstract visitor classifiers (specified by the default multiplicity 0..* which is not shown). Similarly, the *Element* hierarchy specifies that there must one or more concrete classes that play $ConcreteElement$ role and may be abstract concrete elements. $ObjectStructure$ role specifies that there must be one ore more classes that play *ObjectStructure* role.

#### 4.1.2.1 Well-formedness Rules

The following are some of the metamodel-level constraints for the Visitor SPS:

- A classifier that conforms to *AbstractVisitor* must be an interface or an abstract class:

  **context** |AbstractVisitor **inv**:

  self.oclIsTypeOf(Interface) or

  (self.oclIsTypeOf(Class) and self.isAbstract = true)

  A similar constraint is defined for *AbstractElement*.

- A classifier that conforms to *ConcreteVisitor* must be a concrete class:

  **context** |ConcreteVisitor **inv**: self.isAbstract = false

  A similar constraint is defined for *ConcreteElement*.

- A relationship that conforms to *VisitorRealization* must have an interface or a type at its supplier end and a concrete class at its client end:

  **context** |VisitorRealization **inv**:

  (self.supplier.oclIsTypeOf(Interface) or

  (self.supplier.oclIsTypeOf(Class) and self.supplier.isAbstract = true) and

  self.client.oclIsTypeOf(Class)

  A similar constraint is defined for *ElementRealization*.

- A relationship that conforms to *VisitorGeneralization* must have its parent and child to be the same type:

  **context** |VisitorGeneralization **inv**:

  self.parent.evaluationType() = self.child.evaluationType()

  A similar constraint is defined for *ElementGeneralization*.

- An association-end that conforms to *Obj* must have a multiplicity of 0..1:

  **context** |Obj **inv**: self.lowerBound() = 0 and self.upperBound() = 1

- An association-end that conforms to *Elem* must have a multiplicity of 1..*:

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

Metamodel-level constraints for the hierarchies in the other pattern specifications described in this dissertation can be defined similarly as in the Visitor pattern specification.

#### 4.1.2.2 Constraint Templates

Constraint templates for the *VisitElem* and *Accept* behavioral feature roles are given below:

- An *Accept* operation invokes a *VisitElem* operation call:

  **context** |Element:: |Accept (|vis:|ConcreteVisitor)

  **pre** : true

  **post**: **let** elemMessage: OclMessage =

  |ConcreteVisitor^^|VisitElem(|elem:ConcreteElement) → notEmpty()

- A *VisitElem* operation invokes an *Operation* operation call:

  **context** |Visitor:: |VisitElem(|elem : |ConcreteElement)

  **pre** : true

  **post**: **let** visitorMessage: OclMessage =

  |ConcreteElement^^|Operation() → notEmpty()

There are no pre- and post-conditions defined for the *Operation* role, which means any operation can play the role.

#### 4.1.2.3 Folded Form of SPS

A role hierarchy may be folded for a high level view of abstraction (see Section 3.2.1.4). A folded form of the SPS in Fig. 4.3 is shown in Fig. 4.4.

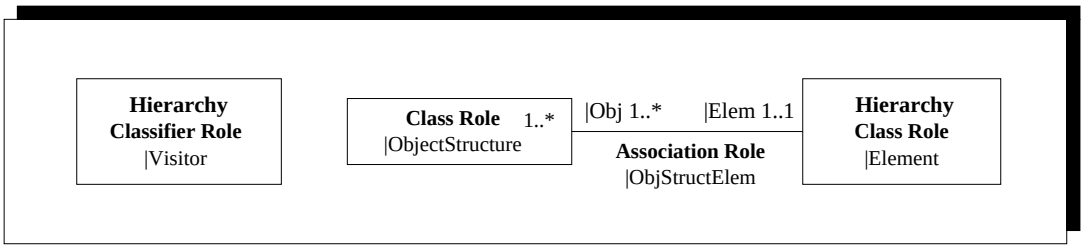



Figure 4.4: A Folded Form of the Visitor SPS

### 4.1.3 Example of a Conforming Class Diagram

The class diagram shown in Fig. 4.1 structurally conforms to the Visitor SPS with respect to the bindings shown in Fig. 4.5. A more complex class diagram that conforms to the Visitor SPS is shown in Fig. 4.6. This diagram includes an element class structure that describes composite elements.

An instance of *CompositeEquipment* is a composite element structure that can also be an element in a larger element structure (i.e., it can be visited by an instance of the visitor class *PricingVisitor*). The *CompositeEquipment* thus plays two roles: *Element* and *ObjectStructure*.

The semantic properties expressed in the visitor pattern concern the interactions that take place in the context of the $VisitElem$, *Accept* and *Operation* behaviors. These properties are described by the pattern's IPS (see Section 4.1.5).

### 4.1.4 Example of a Non-conforming Class Diagram

Fig. 4.7 shows a class diagram that does not conform to the Visitor SPS for the following violations. First, *PricingVisitor* class playing *ConcreteVisitor* role does not have a generalization or realization required by the Visitor SPS. The Visitor SPS requires

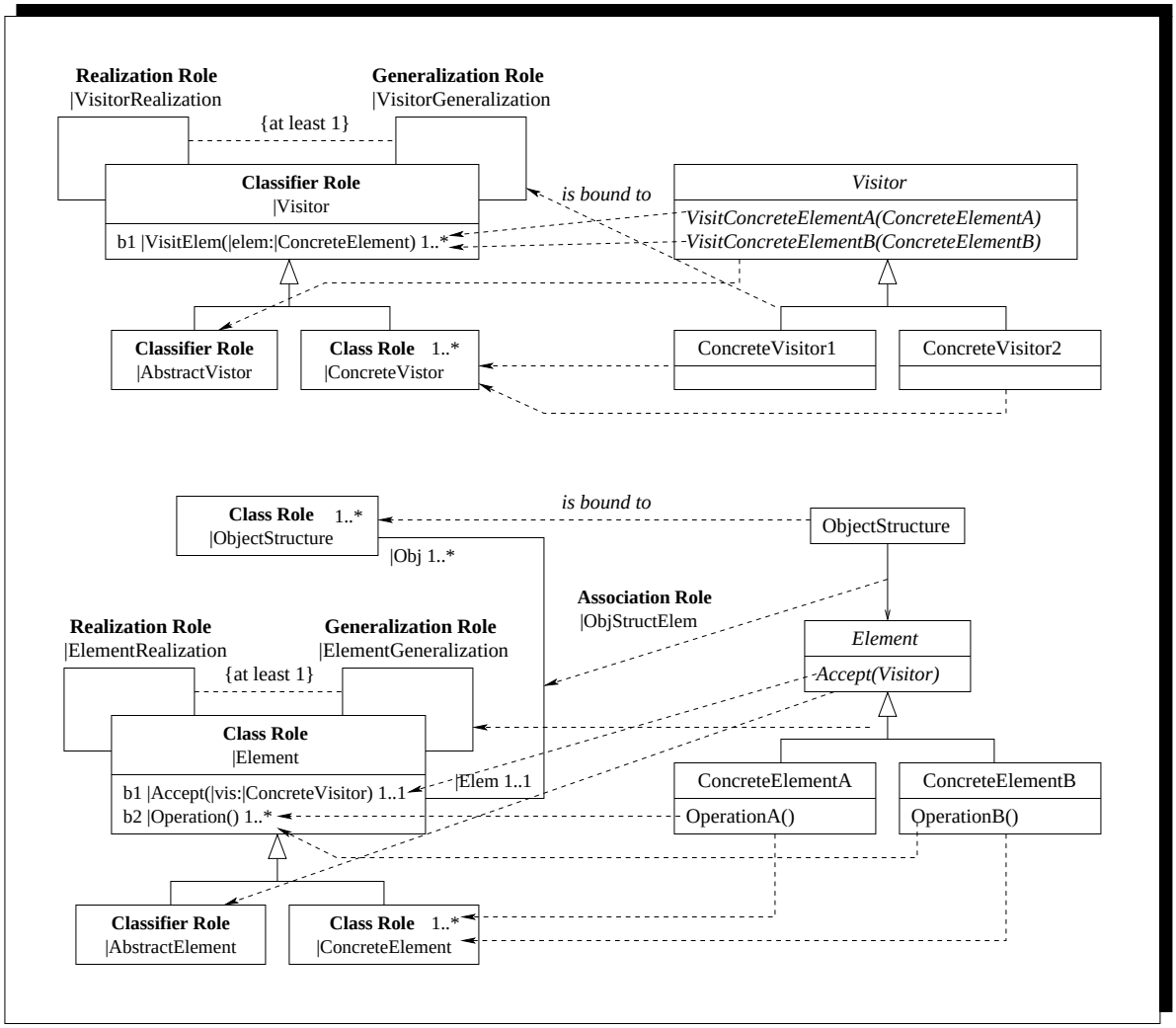



Figure 4.5: A Structurally Conforming Visitor Class Diagram

a conforming class model to have at least one generalization or realization for visitor classes. Second, there are no operations that play *Operation* role in elements classes. The Visitor SPS specifies that all classes that play elements must have operations playing *Accept* and *Operation* roles. Third, the multiplicities at the ends of *composed-of* association does not satisfy the constraints specified on *Obj* and *Elem* roles. The constraints specify that an association end playing *Obj* must have a multiplicity of 0..1 and an association end playing *Elem* must have a multiplicity of 1..*.

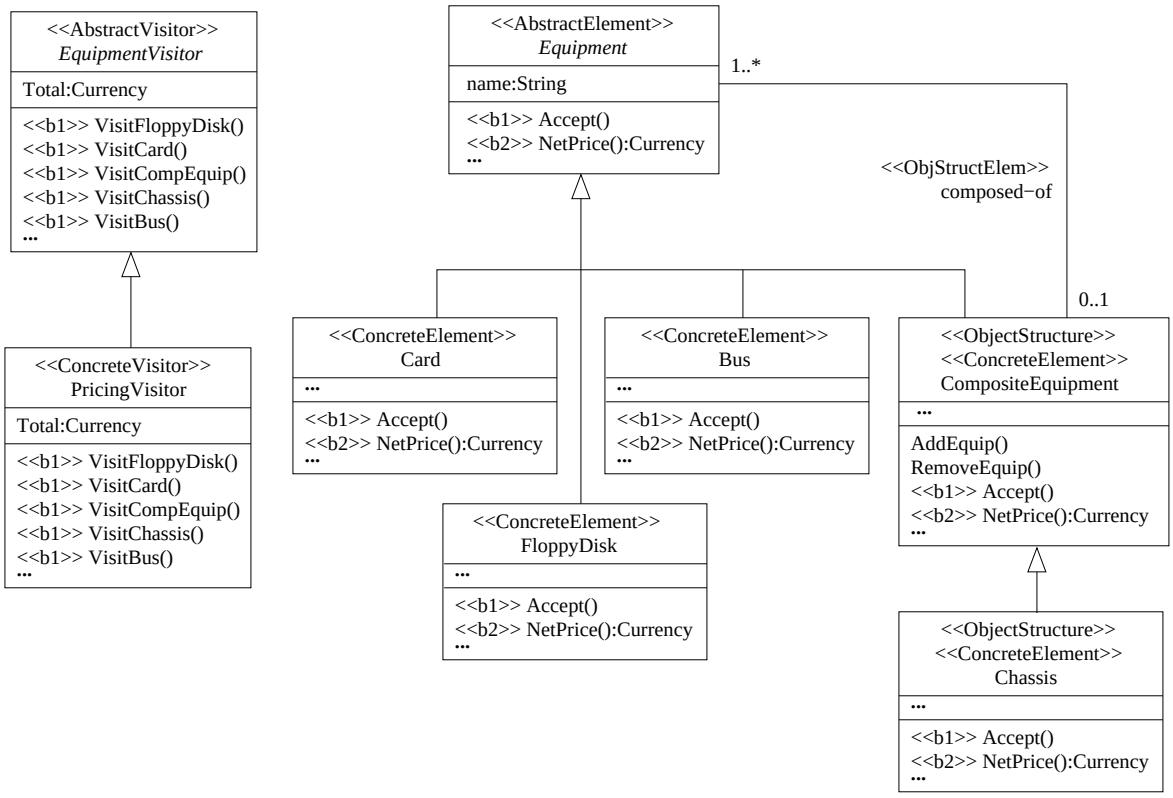



Figure 4.6: A More Complex Conforming Visitor Class Diagram

### 4.1.5 The Iterator IPS

Fig. 4.8 shows an IPS named *CompositeInteraction* that describes the interactions that take place when accessing a composite element structure with a visitor.

An instance of an *ObjectStructure* class plays the role *obj*. The $i^{th}$ element of the object structure plays the role $elem[i]$. The interaction structure enclosed in the **repeat** fragment is repeated for each element in the object structure that plays the role *obj*. $NumOfElements$ is the number or elements associated with the object structure.

An *Accept* message is sent to each element, $elem[i]$, in the object structure. If the element, $elem[i]$, is a composite element then the interaction structure defined in *CompositeInteraction* is recursively applied with $elem[i]$ becoming the

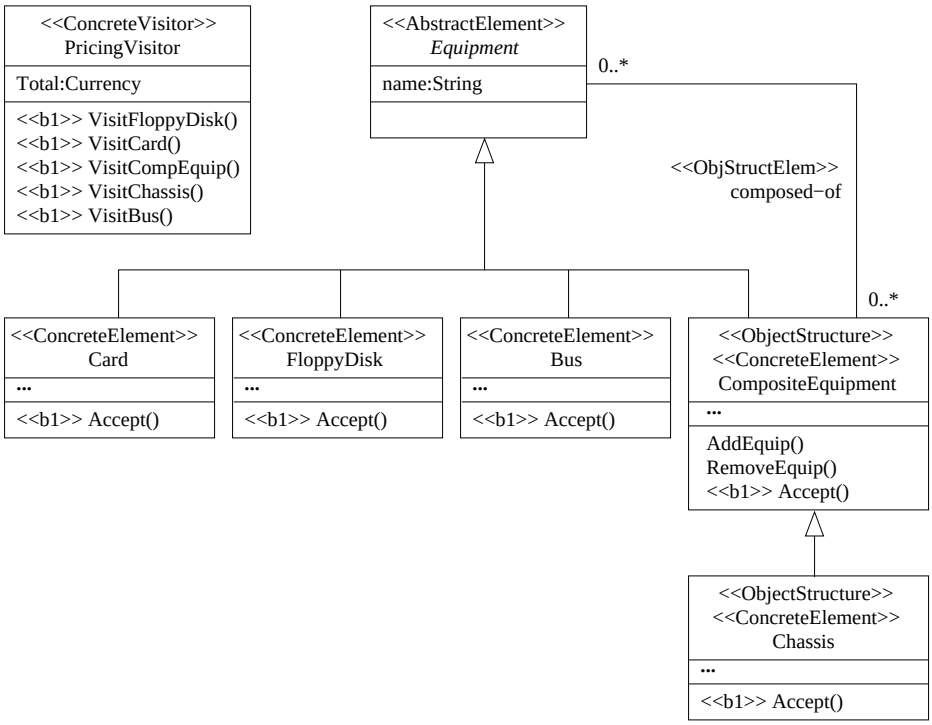



Figure 4.7: A Non-conforming Visitor Class Diagram

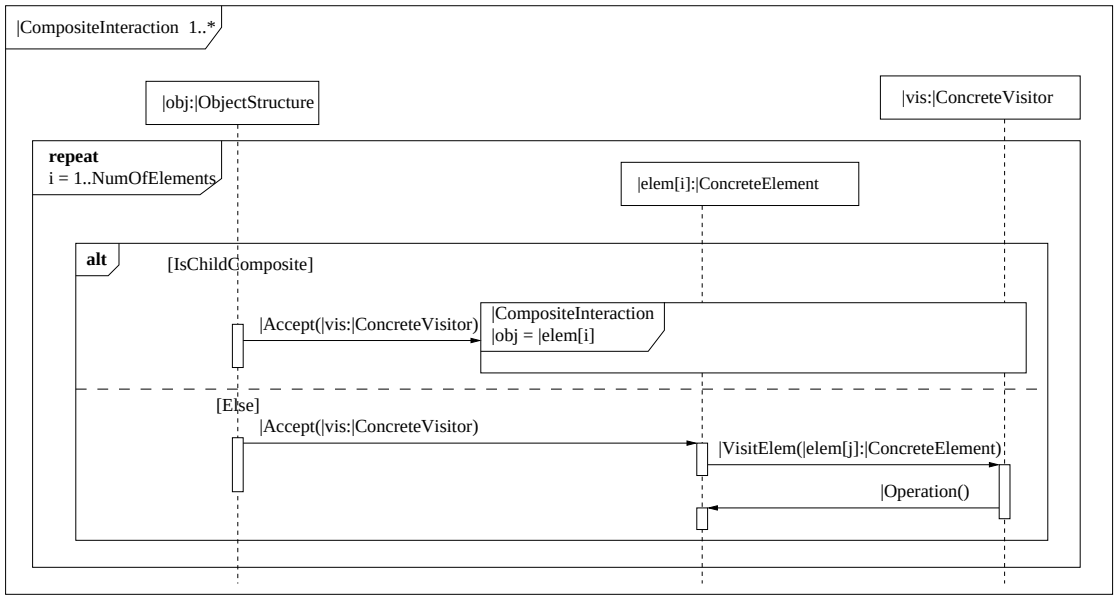



Figure 4.8: A Visitor IPS

*ObjectStructure* participant (i.e., $|obj = |elem[i]$). If the element is not a composite element (i.e., it is a primitive element) then the element calls the $VisitElem$ operation in the visitor. This results in the visitor invoking an operation on the element. The choice between interaction structures for primitive and composite elements is represented by the fragment labeled **alt**. This fragment is divided into two regions describing alternative interaction structures. A *guard condition* determines the region of an **alt** fragment that is selected in a particular situation. The guard condition for the top region is $[IsChildComposite]$ which is true if the element is composite (i.e., the element is an object structure) and false otherwise. The bottom region of the **alt** fragment has a guard $[Else]$ which is true when $IsChildComposite$ is false, and false otherwise.

The simple interaction diagram shown in Fig. 4.2 conforms to the Visitor IPS:

- The $anObjectStructure$ lifeline conforms to the lifeline role $obj$,
- Lifelines for $aConcreteElementA$ and $aConcreteElementB$ conform to the $elem[i]$ lifeline role.
- The $aConcreteVisitor$ lifeline conforms to the lifeline role $vis$.
- The relative order of interactions conforms to the order specified in the IPS. The calls to the *Accept* operations and the ensuing interactions are described by interaction structures obtained by applying the $Else$ part of the **alt** fragment twice.

An example of a composite element structure described by the class diagram given in Fig. 4.6 is shown in Fig. 4.9. The composite element $EquipStructure$ consists of three elements: a primitive element $FloppyDisk1$, a primitive element $Bus2$, and a composite element $Chassis1$. The composite element $Chassis1$ consists of a primitive element $Bus1$ and a composite element $Chassis2$. Fig. 4.10 shows a Visitor sequence diagram that is based on the composite structure shown in Fig. 4.9.

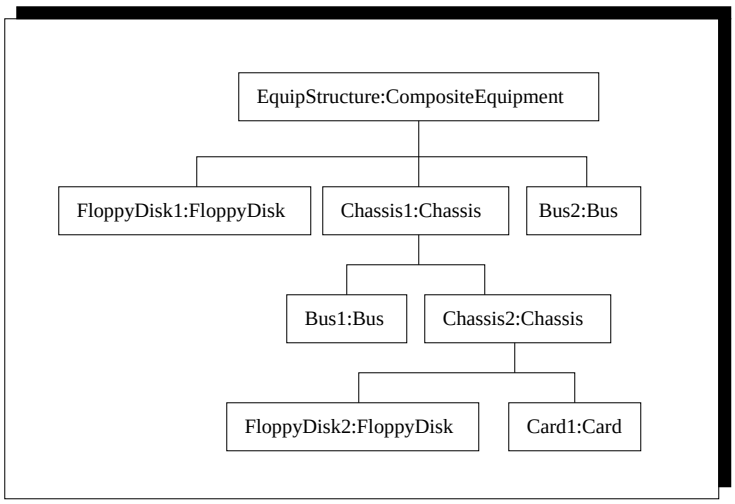



Figure 4.9: A Composite Part Structure

### 4.1.6 Example of a Conforming Sequence Diagram

The interaction sequence involving $FloppyDisk1$ and the sequence involving $Bus2$ have the structure specified by the *Else* part of the **alt** fragment. The interaction sequences involving $Chassis1$ has the structure specified by the $IsChildComposite$ region of the **alt** fragment. Establishing this involves recursively applying the $CompositeInteraction$ structure: $Chassis1$ becomes the $ObjectStructure$ lifeline, $Card1$ becomes the primitive element involved in the interactions described by the *Else* region, and $Chassis2$ becomes the composite element involved in the interactions described by the $IsChildComposite$ region.

The two examples of conforming sequence diagrams given in this section demonstrate the wide range of interaction structures characterized by the concisely stated Visitor IPS.

### 4.1.7 Example of a Non-conforming Sequence Diagram

Fig. 4.11 shows a sequence diagram that does not conform to the Visitor IPS for the following violations. First, the sequence diagram does not include elements of *Flop-*

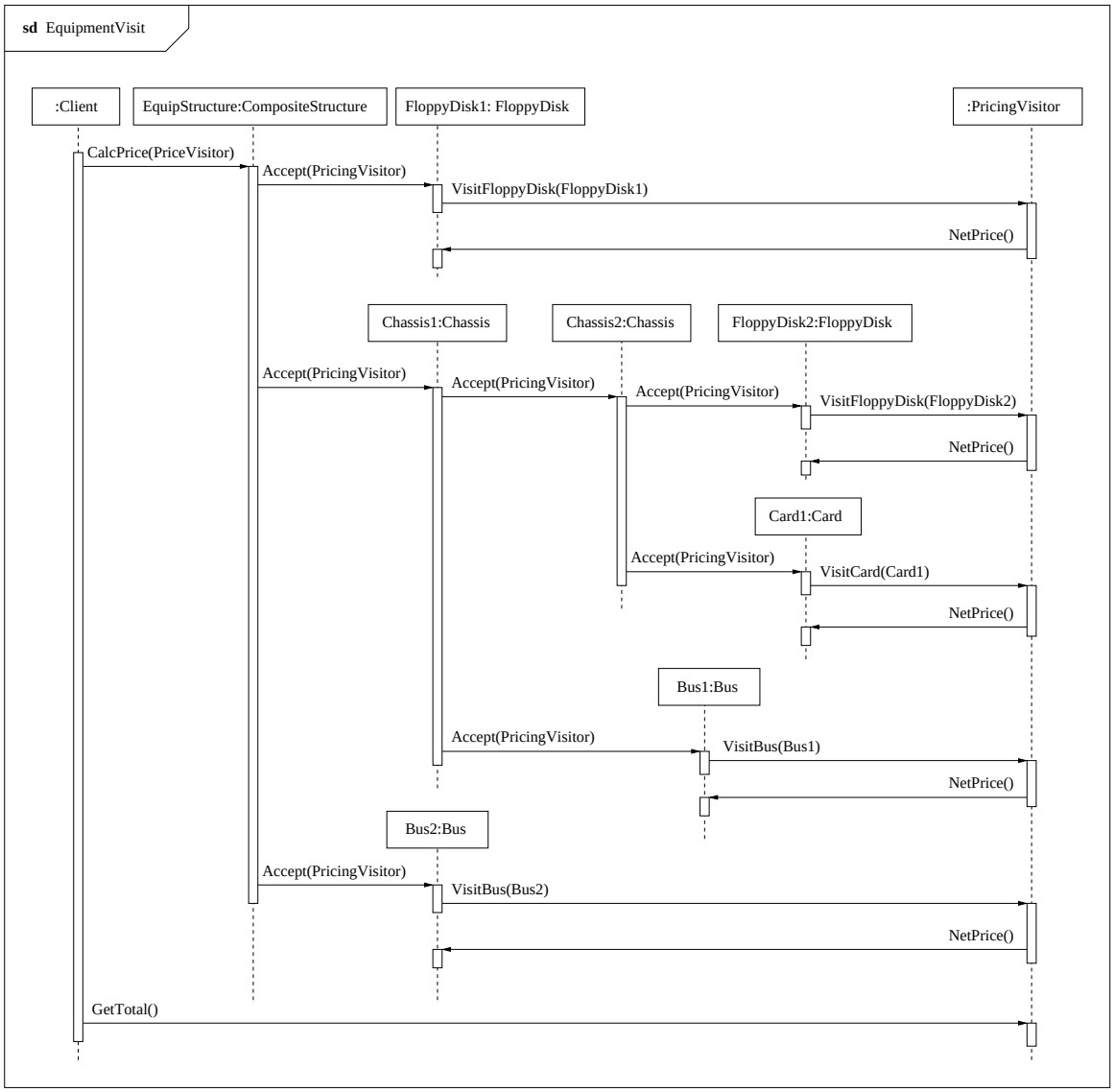



Figure 4.10: A Conforming Visitor Sequence Diagram

*pyDisk2:FloppyDisk* and *Card1:Card*, which violates the recursion specified in the upper compartment of *alt* operator in the Visitor IPS. Second, the sequence of *NetPrice* and *VisitFloppyDisk* messages between *Bus1:Bus* and *Pricing:Visitor* violates the sequence specified in the lower compartment of *alt* operator.

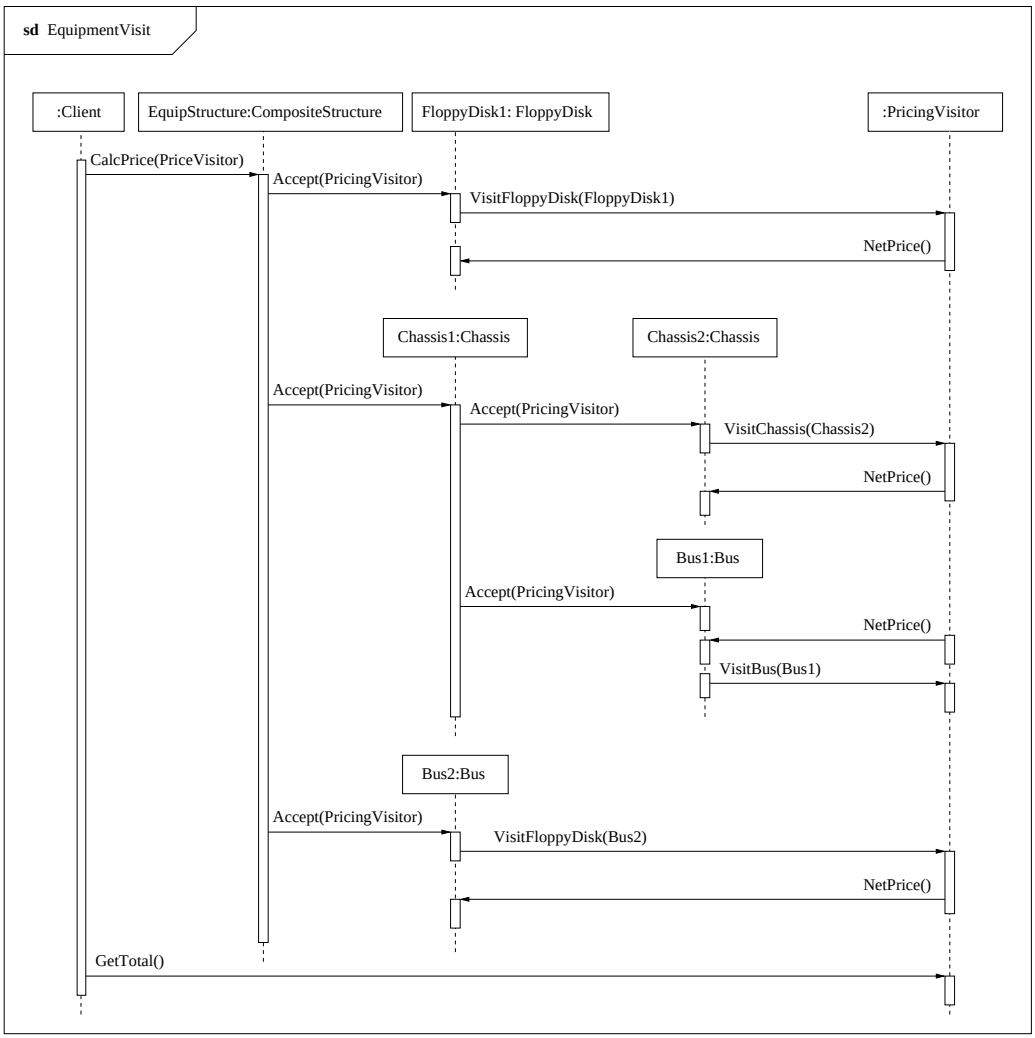



Figure 4.11: A Non-conforming Visitor Sequence Diagram

## 4.2 Specifying the Abstract Factory Pattern

The Abstract Factory pattern provides a way to create families of related objects called products without specifying their concrete classes through objects called factories. The pattern enables a separation of abstraction from implementation for products and factories to make it easy to change them without changing clients. Only the abstraction is revealed to clients so that clients do not need to concern how concrete products are created.

### 4.2.1 The Abstract Factory SPS

The class diagrams characterized by an Abstract Factory SPS have factory and product classifiers which are abstract or concrete and a client class associated with an abstract factory and abstract product. The abstract factory contains creation operations. The client calls the operations to create products and concrete factories perform the actual creation.

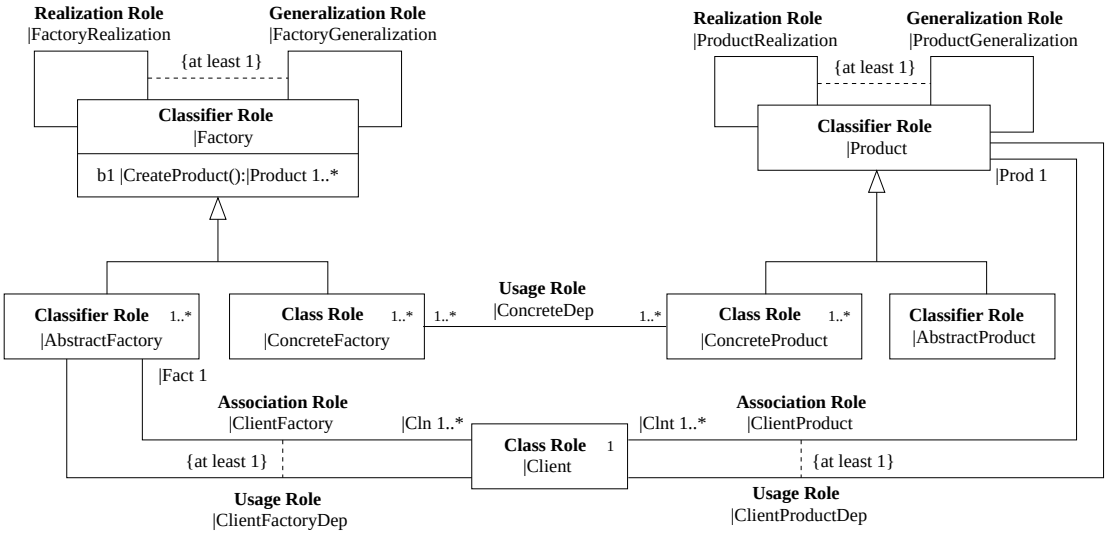



Figure 4.12: An Abstract Factory SPS

An SPS for the Abstract Factory pattern is shown in Fig. 4.12. The *Factory* role hierarchy comprises of an abstract role *Factory* which is not realizable, and its specializations of *AbstractFactory* and *ConcreteFactory*. A conforming factory structure must have at least one abstract classifier that plays *AbstractFactory* role, at least one concrete class that plays *ConcreteFactory* role, and one relationship that plays either *FactoryGeneralization* or *FactoryRealization* between *AbstractFactory* classifiers and *ConcreteFactory* classes. The *Product* role hierarchy is interpreted similarly except that *AbstractProduct* is optional.

The multiplicity on the *Client* role restricts that a conforming model must have exactly one conforming class of *Client*. The pre-defined constraint $\{at\,least\,1\}$ shown

between the *ClientFactory* and *ClientFactoryDep* roles constraints that there must be at least one relationship playing one of the roles between *Factory* classifiers and the *Client* class. Similarly, the *Client* class is connected to one or more *Product* classifiers via association or usage dependency.

#### 4.2.1.1 Well-formedness Rules

The metamodel-level constraints defined on the Abstract Factory SPS are as follows:

- A classifier that conforms to *AbstractFactory* must be an interface or an abstract class:

  **context** |AbstractFactory **inv**:

  self.oclIsTypeOf(Interface) or

  (self.oclIsTypeOf(Class) and self.isAbstract = true)

  A similar constraint is defined for *AbstractProduct*.

- A classifier that conforms to *ConcreteFactory* must be a concrete class:

  **context** |ConcreteFactory **inv**: self.isAbstract = false

  A similar constraint is defined for *ConcreteProduct*.

- A relationship that conforms to *FactoryRealization* must have an interface or a type at its supplier end and a concrete class at its client end:

  **context** |FactoryRealization **inv**:

  (self.supplier.oclIsTypeOf(Interface) or

  (self.supplier.oclIsTypeOf(Class) and self.supplier.isAbstract = true) and

  self.client.oclIsTypeOf(Class)

  A similar constraint is defined for *ProductRealization*.

- A relationship that conforms to *FactoryGeneralization* must have its parent and child to be the same type:

**context** |FactoryGeneralization **inv**:

self.parent.evaluationType() = self.child.evaluationType()

A similar constraint is defined for *ProductGeneralization*.

- A classifier that conforms to *Client* must be a concrete class:

  **context** |Client **inv**: self.isAbstract = false

- An association-end that conforms to *Fact* must have a multiplicity of 1..*:

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

- An association-end that conforms to *Cln* must have a multiplicity of 1..1:

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

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

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

- An association-end that conforms to *Prod* must have a multiplicity of 1..*:

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

- *ConcreteFactory* classes are responsible for creating products:

  **context** |ConcreteDep **inv** self.stereotype.name = 'create'

- A client class uses operations defined in *AbstractFactory* classes:

  **context** |ClientFactoryDep **inv** self.stereotype.name = 'call'

- A client class uses operations defined in *Product* classes:

  **context** |ClientProductDep **inv** self.stereotype.name = 'call'

#### 4.2.1.2 Constraint Templates

*Factory* classifiers must have one or more operations that play *CreateProduct* role to create a new instance of a *Product* classifier. A constraint template for *CreateProduct* is given below:

**context** |Factory:: |CreateProduct(): |Product

**pre**: true

**post**: result = p and p.oclIsNew() = true

#### 4.2.1.3 SPS Specializations

Fig. 4.13 shows two SPSs that are specializations of the Abstract Factory SPS shown in Fig. 4.12. Fig. 4.13(a) has the following specialized properties:

- Conforming classifiers of *AbstractFactory* are restricted to UML interfaces, and conforming classifiers of *Product* are restricted to classes:

  **context** |AbstractProduct **inv**:

  self.oclIsTypeOf(Class) and self.isAbstract = true

- The relationships between *AbstractFactory* and *ConcreteFactory* classifiers are restricted to UML $\ll$ *realize* $\gg$ dependencies conforming to *FactoryRealization*, while the relationships between *Product* classifiers are restricted to generalizations conforming to *ProductGeneralization*.

- The relationship between *Client* and *AbstractFactory* classifiers are restricted to association conforming to *ClientFactory*.

- The relationship between *Client* and *Product* classifiers are restricted to associations conforming to *ClientProduct*.

Fig. 4.13(b) specializes the Abstract Factor SPS by restricting *Factory* and *Product* classifiers to be concrete classes. Hierarchies of *Product* and

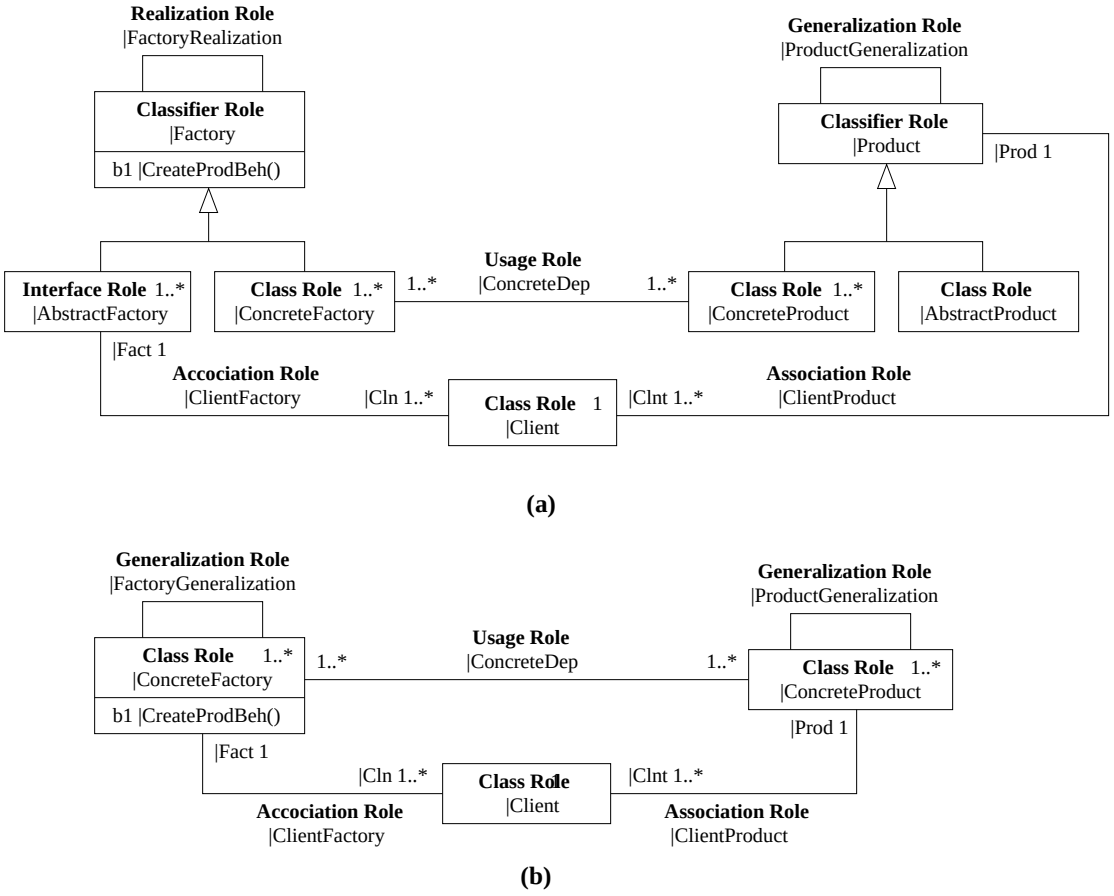



Figure 4.13: Specialized Abstract Factory SPSs

*Factory* classifiers are formed using generalization relationships conforming to *FactoryGeneralization* and *ProductGeneralization*, respectively. The other aspects of the specialization are similar to the specialization in Fig. 4.13(a).

### 4.2.2 Example of a Conforming Class Diagram

Fig. 4.14 shows a conforming class diagram of the Abstract Factory SPS which describes a simple maze game [35] where a maze consists of rooms with four sides of doors and walls, and a concrete factory class *MazeFactory* is responsible for creating *Maze* product with its sub-products of *Door*, *Wall*, and *Room*. The operations *makeMaze*, *makeDoor*, *makeWall*, and *makeRoom* in *MazeFactory* all play the *Cre-*

*ateProduct* behavioral role in *Factory*. Bold stereotypes indicate the role that the model element plays.

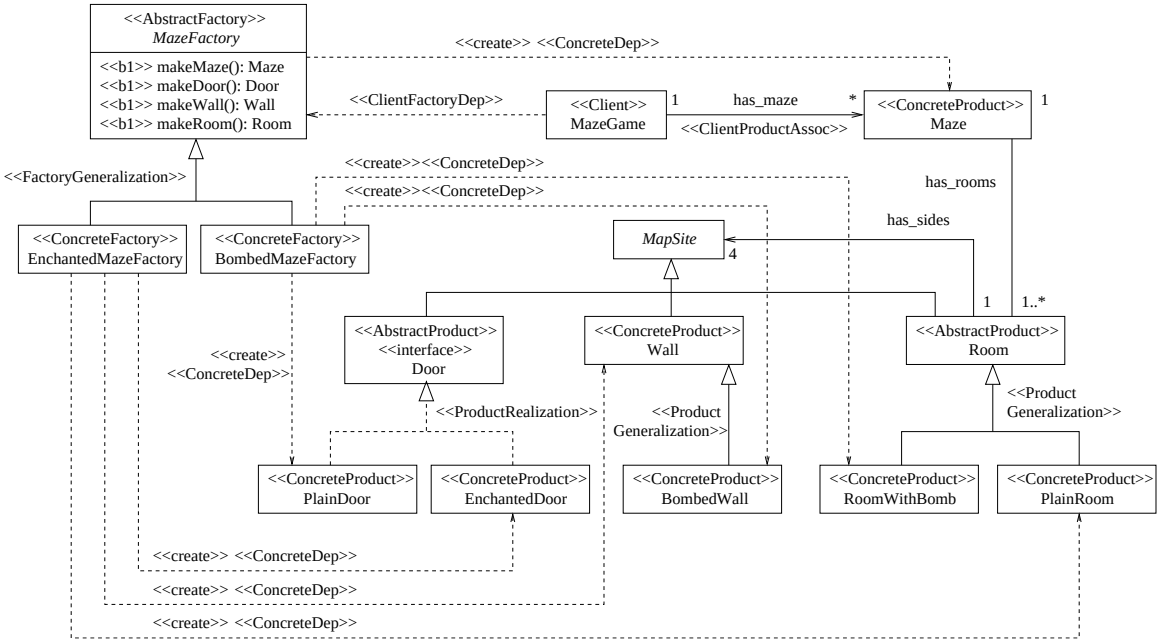



Figure 4.14: A Conforming Class Diagram of the Abstract Factory SPS with Hierarchies

Two types of mazes are produced: *EnchantedMaze* and *BombedMaze*. *EnchantedMazeFactory* is responsible for creating *BombedMaze* products with its sub-products of *EnchantedDoor*, *Wall*, and *PlainRoom*, and *BombedMazeFactory* is responsible for creating *BombedMaze* products with its sub-products of *PlainDoor*, *BombedWall*, and *RoomWithBomb*.

### 4.2.3 Example of a Non-conforming Class Diagram

Fig. 4.15 shows a class diagram that does not conform to the Abstract Factory SPS because it does not have abstract classifiers that play *AbstractFactory* role, which violates the multiplicity constraint (1..*) defined in *AbstractFactory*. This consequently results in another violation that there should be relationships that plays *FactoryRealization* or *FactoryGeneralization* roles.

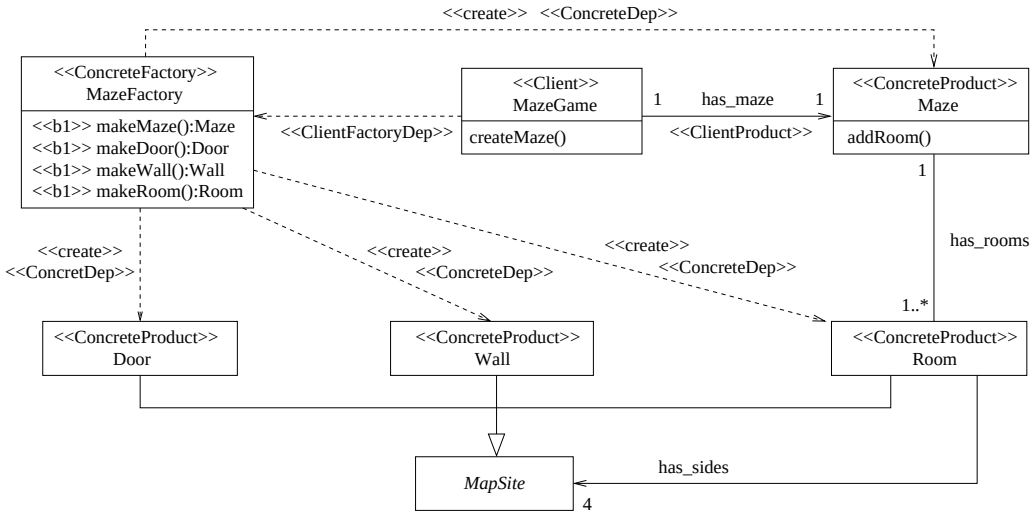



Figure 4.15: A Conforming Class Diagram of the Abstract Factory SPS with no Hierarchy

## 4.3 Specifying the Iterator Pattern

The Iterator pattern provides a way to access to the elements of an aggregate object without exposing its internal structure by separating responsibility for access and traversal from the aggregate and putting it into an object called *iterator*. Such a separation facilitates defining different traversal policies without bloating the interface of the aggregate object. The Iterator pattern enables a separation of abstraction from implementation for aggregates and iterators to make it easy to change them without changing clients.

### 4.3.1 The Iterator SPS

The class diagrams characterized by an Iterator SPS have structures of aggregate and iterator classifiers that are abstract or concrete, and an item class associated with the abstract aggregate. Aggregates are responsible for creating iterators which define a traversal fashion.

An SPS for the Iterator pattern is shown in Fig. 4.16. The *Aggregate* role hierarchy

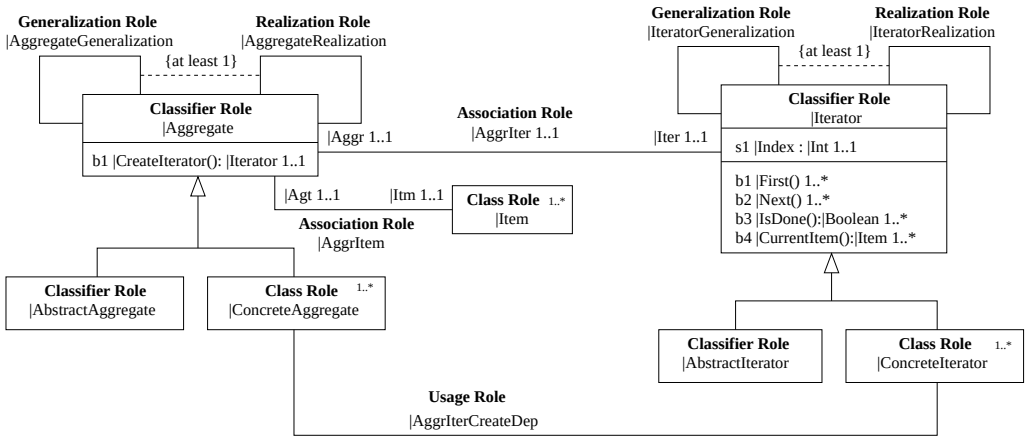



Figure 4.16: An Iterator SPS

comprises of *Aggregate* which is not realizable, and its specializations of *AbstractAggregate* and *ConcreteAggregate*. A conforming structure of the hierarchy must have at least one concrete class that conforms to *ConcreteAggregate*. A conforming aggregate classifier must possess exactly one operation playing *CreateIterator* role that creates an iterator. The *Iterator* role hierarchy is interpreted similarly. A conforming iterator classifier must have exactly one attribute that plays *Index* role and operations that plays *First*, *Next*, *IsDone*, and *CurrentItem* roles. There must be exactly one concrete class that plays *Item* role. The *AggrItem* specifies that an object of a classifier playing *Aggregate* is an aggregate of the objects of the item class. The *AggrIterCreateDep* specifies that concrete aggregate classes are responsible for creating objects of a concrete iterator class. The *AggrIter* specifies that iterators keep track of the current item in aggregates.

#### 4.3.1.1 Well-formedness Rules

The metamodel-level constraints defined on the Iterator SPS are as follows:

- A classifier that conforms to *AbstractAggregate* must be an interface or an abstract class:

**context** |AbstractAggregate **inv**:

self.oclIsTypeOf(Interface) or

(self.oclIsTypeOf(Class) and self.isAbstract = true)

A similar constraint is defined for *AbstractIterator*.

- A classifier that conforms to *ConcreteAggregate* must be a concrete class:

  **context** |ConcreteAggregate **inv**: self.isAbstract = false

  A similar constraint is defined for *ConcreteIterator*.

- A relationship that conforms to *AggregateRealization* must have an interface or a type at its supplier end and a concrete class at its client end:

  **context** |AggregateRealization **inv**:

  (self.supplier.oclIsTypeOf(Interface) or

  (self.supplier.oclIsTypeOf(Class) and self.supplier.isAbstract = true) and

  self.client.oclIsTypeOf(Class)

  A similar constraint is defined for *IteratorRealization*.

- A relationship that conforms to *AggregateGeneralization* must have its parent and child to be the same type:

  **context** |AggregateGeneralization **inv**:

  self.parent.evaluationType() = self.child.evaluationType()

  A similar constraint is defined for *IteratorGeneralization*.

- A classifier that conforms to *Item* must be a concrete class:

  **context** |Item **inv**: self.isAbstract = false

- An association-end that conforms to *Aggr* must have a multiplicity of 1..1:

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

- An association-end that conforms to *Iter* must have a multiplicity of 0..*:

  **context** |Iter **inv**: self.lowerBound() = 0 and self.upperBound() = *

- An association-end that conforms to *Agt* must have a multiplicity of 1..1:

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

- An association-end that conforms to *Itm* must have a multiplicity of 0..*:

  **context** |Itm **inv**: self.lowerBound() = 0 and self.upperBound() = *

- *ConcreteAggregate* classes are responsible for creating iterators:

  **context** |AggrIterCreateDep **inv** self.stereotype.name = 'create'

#### 4.3.1.2 Constraint Templates

Constraint templates for the behavioral feature roles *First*, *Next*, *IsDone*, and *CurrentItem* are given below:

- A *CreateIterator* operation creates an iterator:

  **context** |Aggregate :: |CreateIterator(): |Iterator

  **post**: result = i and i.oclIsNew() = true

- A *First* operation moves the index to the first index:

  **context** |AbstractIterator :: |First()

  **post**: |Index = self.|Aggregate.|Item → asSequence()

  → indexOf(self.|Aggregate.|Item → asSequence() → first())

- A *Next* operation moves the index to the next index:

  **context** |AbstractIterator :: |Next()

  **pre**: |Index ≥ 1 and |Index < self.|Aggregate.|Item

  → asSequence() → size()

  **post**: |Index = |Index@pre + 1

- An *IsDone* operation checks if the current item is the last one:

  **context** |AbstractIterator :: |IsDone(): Boolean

  **post**: self.|Aggregate.|Item → asSequence() → at(|Index)

  = self.|Aggregate.|Item → asSequence() → last()

- A *CurrentItem* operation returns the current item at which the index points:

  **context** |AbstractIterator :: |CurrentItem(): |Item

  **pre**: |Index ≥ 1 and |Index ≤ self.|Aggregate.|Item

  → asSequence() → size()

  **post**: |index = |index@pre and result = self.|Aggregate.|Item

  → asSequence() → at(|Index)

### 4.3.2 Example of a Conforming Class Diagram

Fig. 4.17 shows a class diagram that conforms to the Iterator SPS describing a television remote control application that allows viewers to surf channels through features like *channel next* and *previous*.

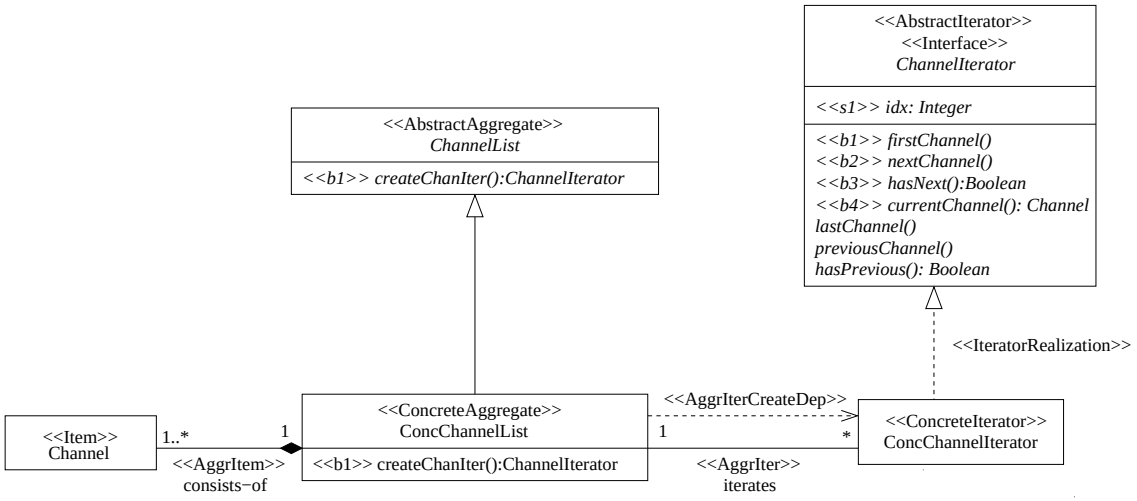



Figure 4.17: A Conforming Iterator Class Diagram

Stereotypes with model elements represent the roles that the model elements play.

For example, *createChanIter* operation in *ChannelList* and *ConcChannelList* plays the *CreateIterator* role in Fig. 4.16. *lastChannel*, *previousChannel*, and *hasPrevious* are application-specific operations. *lastChannel* operation returns the last channel in the list, *previousChannel* returns the previous channel of the current channel, and *hasPrevious* checks if the current channel is the first channel in the list. Instantiated by substituting role names in the SPS constraint templates to conforming model elements, pre- and post-conditions for *createChanIter*, *firstChannel*, *nextChannel*, *hasNext*, and *currentChannel* operations are given below:

**context** ChannelList :: createChanIter(): ChannelIterator

**post**: result = i and i.oclIsNew() = true

**context** ChannelIterator :: firstChannel()

**post**: idx = self.ChannelList.Channel → asSequence()

→ indexOf(self.ChannelList.Channel → asSequence() → first() )

**context** ChannelIterator :: nextChannel()

**pre**: idx ≥ 1 and

idx < self.ChannelList.Channel →

asSequence() → size()

**post**: idx = idx@pre + 1

**context** ChannelIterator :: hasNext(): Boolean

**post**: self.ChannleList.Channel

→ asSequence() → at(idx)

= self.ChannelList.Channel → asSequence() → last()

**context** ChannelIterator :: currentChannel(): Channel

**pre**: idx ≥ 1 and

idx ≤ self.ChannelList.Channel → asSequence() → size()

**post**: idx = idx@pre and

result = self.ChannelList.Channel → asSequence() → at(idx)

Note that the multiplicity (1..*) at the end of Channel does not violate the multiplicity constraint (0..*) defined on *Aggr* role as long as its lower bound is greater than or equal to the lower bound of the constraint and its upper bound is smaller than or equal to the upper bound of the constraint.

### 4.3.3 Example of a Non-conforming Class Diagram

Fig. 4.18 shows a class diagram that does not conform to the Iterator SPS for the following violations. First, *ConcChannelList* has two operations that play *CreateIterator* role, which violates the constraint that there should be exactly one operation playing the role. Second, *ConcChannelList* and *ChannelIterator* do not have generalizations or realizations, which violates the constraints in the *Aggregate* and *Iterator* role hierarchies. Third, the multiplicity (0..1) at the end of *ConcChannelList* does not satisfy the multiplicity constraint (1..1) on *Aggr* role.

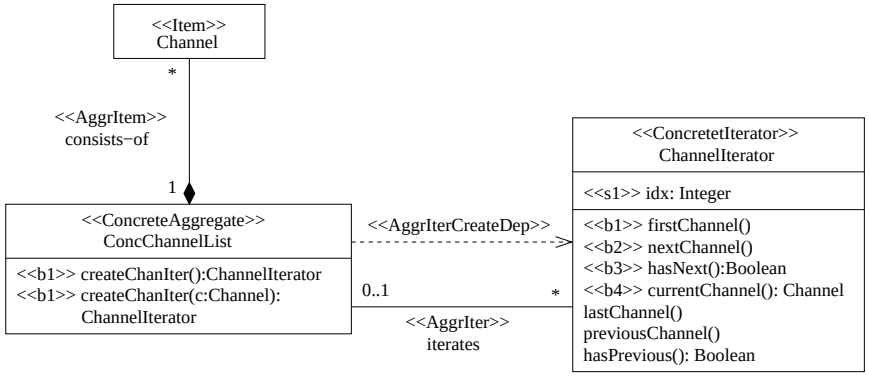



Figure 4.18: A Non-conforming Iterator Class Diagram

### 4.3.4 The Iterator SMPS

Fig. 4.19 presents an SMPS for the *Iterator* role in the *Iterator* pattern. The SMPS depicts the following behavior:

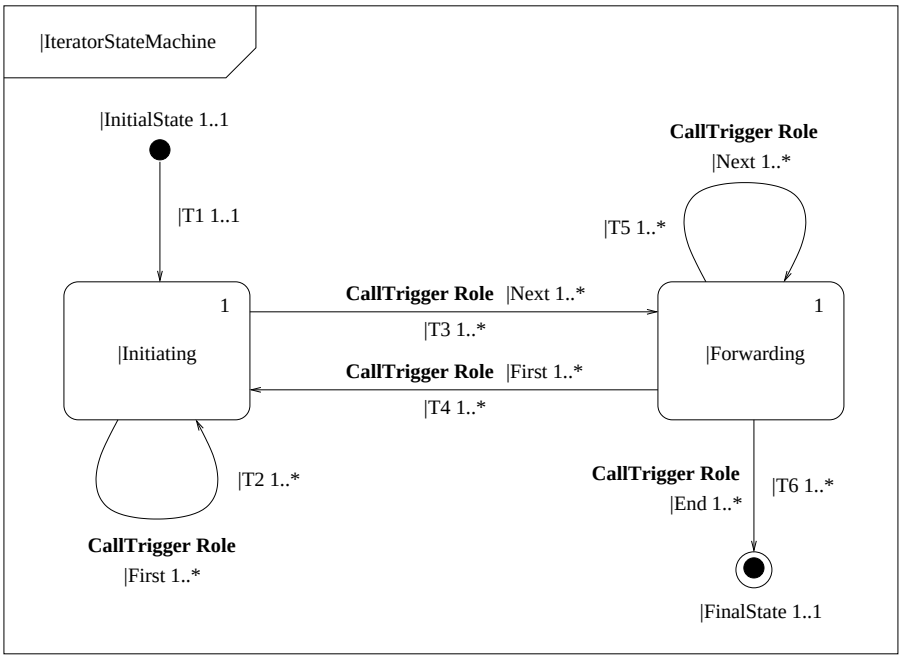



Figure 4.19: An SMPS of *Iterator* Role in Fig. 4.16

- ***Initiating***: When an object of a class playing *Iterator* role is created, the object moves its state from an initial state to a state that plays *Initiating* role through a transition of *T1*. In this state, the object has two possible transitions: 1) the object may stay its current state when a call trigger of *First* is received; or 2) the object may move to a state of *Forwarding* when a call trigger of *Next* is received.

- ***Forwarding***: An object in this state has three possible transitions: 1) the object may move back to a state of *Initiating* when a call trigger of *First* is received; 2) the object may stay at the current state if a call trigger of *Next* is

received; or 3) the object terminates its lifetime at a state of *FinalState* when a call trigger of *End* occurs.

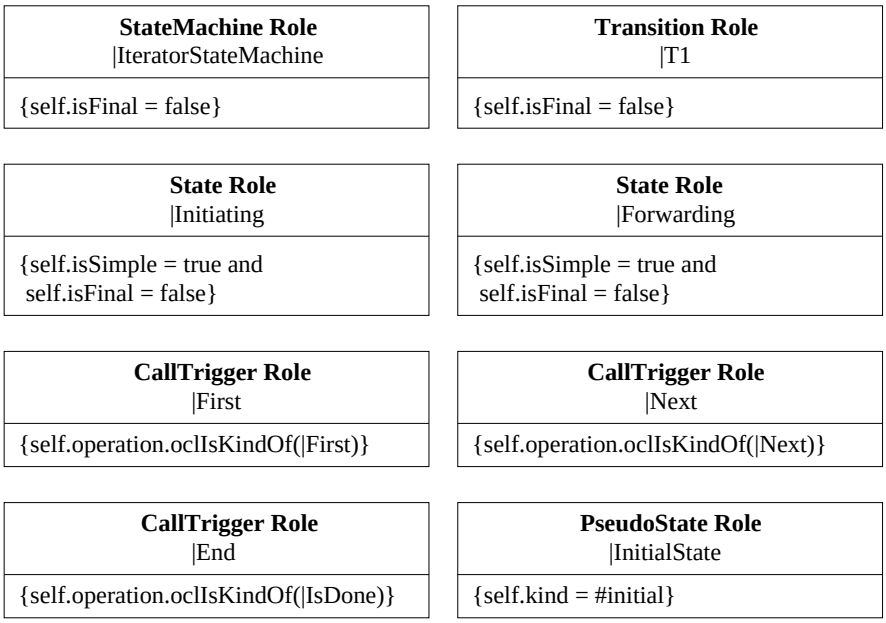
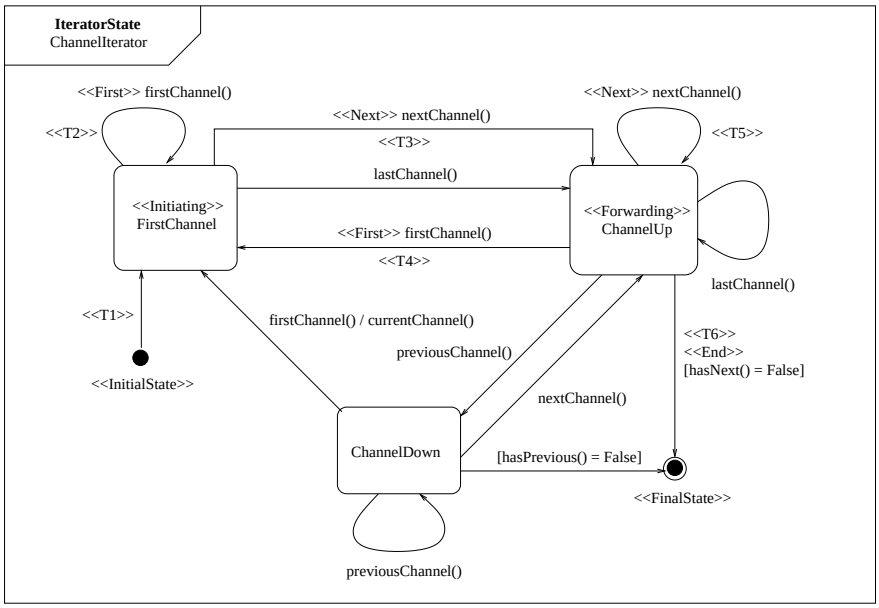



Figure 4.20: Metamodel-level Constraints

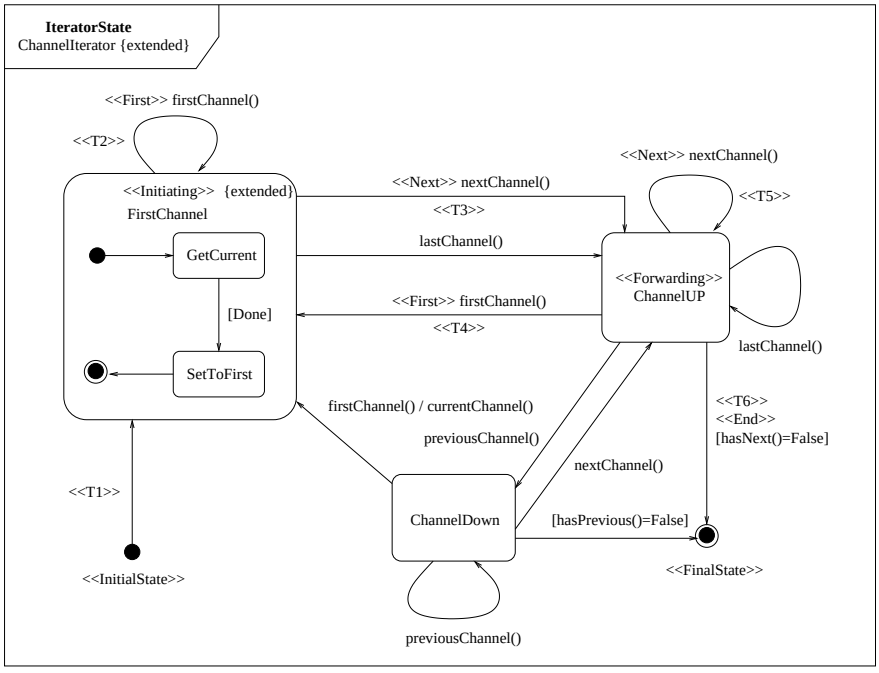
Metamodel-level constraints for the Iterator SMPS are defined in Fig. 4.20. They describe the following:

- Conforming state machines of *IteratorStateMachine* and transitions playing *T1* may be extended (isFinal = false). Similar constraints are defined for the other transition roles.
- States playing *Initiating* and *Forwarding* must be simple states (isSimple = true) and may be extended.
- Triggers playing *First*, *Next*, and *End* roles must be call triggers that are caused by *First*, *Next*, and *isDone* operation calls in Fig. 4.16.
- States playing *InitialState* must be initial states.

### 4.3.5 Examples of Conforming Statecharts




(a) A Television Remote Statechart




(b) A Specialization of the Television Remote Statechart

Figure 4.21: Conforming Iterator Statecharts

Fig. 4.21(a) shows a statechart diagram that conforms to the Iterator SMPS in Fig. 4.19. Model elements that play the roles in Fig. 4.19 are stereotyped. Fig. 4.21(b) shows a case in which a specialization of the television remote state machine *does not* conform to the Iterator SMPS. The simple state *FirstChannel* that plays the *Initiating* role in Fig. 4.19 is extended to a composite state to save the current index before resetting the iterator. This violates the constraint (*isSimple* = *true*) defined in *Initiating* role in Fig. 4.20, and thus the specialization does not conform to the SMPS. In order to disallow such cases, the constraint described in the SMPS must hold.

### 4.3.6 Example of a Non-conforming Statechart

The statechart shown in Fig. 4.22 does not conform to the Iterator SMPS because there are no triggers that play *Next* and *End* roles and no transitions that play *T3* and *T6* roles in the Iterator SMPS.

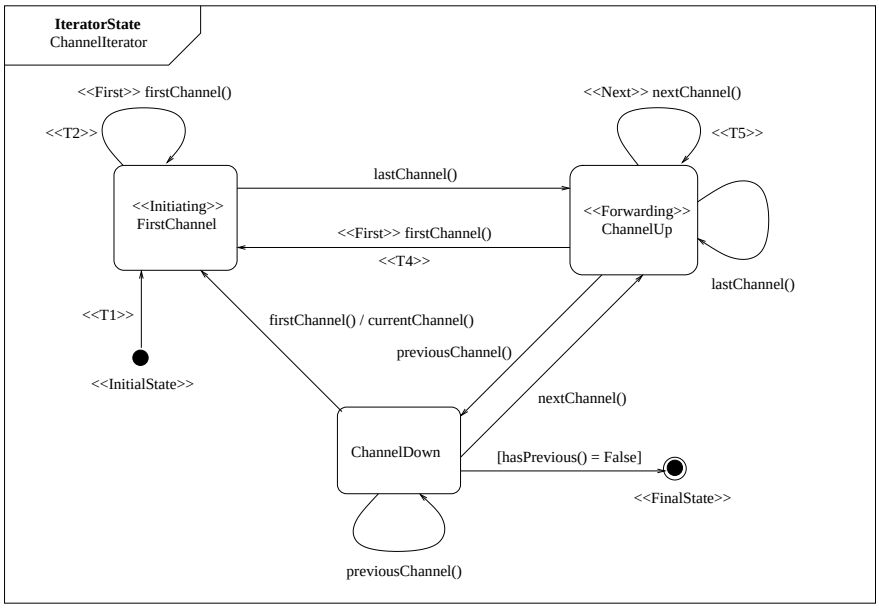



Figure 4.22: A Non-conforming Iterator Statechart

## 4.4 Lessons Learned

This chapter has described specifications of Visitor, Abstract Factory, and Iterator design patterns using the RBML. The RBML has been presented to a graduate-level software engineering course that has about fifteen students, and used by them to develop specifications of design patterns. All the students were familiar with the UML and design patterns and had used them in previous courses. Our collective experience revealed the following about the RBML:

- The students were able to create specifications for patterns that did not involve the use of recursion in the interaction diagrams after two lectures on the pattern specification notation. Students who are not familiar with the UML metamodel experienced some difficulty in presenting RBML concepts.

- Pattern behaviors that are localized in methods or in objects (e.g., see the Factory Method and the Iterator patterns) could not be fully captured by operation templates or interaction diagrams. This was the motivation for the development of SMPSs to specify localized behaviors of pattern participants. It is important to note that the RBML is restricted to descriptions of structure and behavior that can be expressed in the UML.

- Defining recursive behaviors (as required by the Visitor and Decorator patterns) using the UML 1.4 interaction diagrams was difficult for lack of constructs, and resulted in complicated representations of IPSs. The UML 2.0 sequence diagram notation used in this chapter offers a richer set of constructs, including constructs for packaging and referencing interactions. The interpretation of these constructs needed to be adapted to fulfill the research's needs (e.g., the repeat construct is an adaptation of the UML 2.0 loop construct), but sequence diagram "look and feel" in IPSs is maintained. The Visitor IPS given in this

chapter illustrates how these constructs can be used to represent a range of behaviors concisely.

# Chapter 5

# Using the RBML to Specifying Domain Patterns

Cost-effective development of large computer-based systems can be realized through systematic reuse of application domain-specific design experience [12, 72, 82]. Such experience can be captured by application domain-specific design patterns (henceforth called domain patterns), that is, patterns specifying design solutions for well-defined families of applications.

This chapter describes how the RBML can be used to express domain patterns that define domain-specific UML sub-languages. Use of the sub-languages to build models of applications in the domains results in reuse of design experience embedded in the domain patterns. For example, developers can use RBML specifications of a pattern in a domain to create UML diagrams for the applications in the domain.

The RBML is used to create a domain pattern for a *checkin-checkout* (CICO) application domain. The primary purpose of applications in this domain is to provide services for checking in and out items. Applications within this domain include video rental, car rental, and library systems. The RBML CICO pattern is used to obtain UML models of a library system and a car rental system. The work presented in this chapter has been published [57, 58].

This chapter is organized as follows. Section 5.1 describes how the RBML can be

used to develop a CICO domain pattern, and Section 5.2 illustrates how the CICO domain pattern is used to build UML models of a library application and a car rental application. Section 5.3 gives an overview of related work, and Section 5.4 concludes the chapter with the lessons learned from this case study.

## 5.1 Specifying the CICO Pattern

The CICO domain pattern characterizes a family of checkin-checkout applications that manage item check in or check out. Applications within this domain include video rental, car rental and library systems. Some features of CICO applications characterized by the domain pattern are given below:

1. Items that can be checked out and in have unique identifiers.

2. Items are maintained in one or more collections (e.g., a library system can have a collection of journals, a collection of references, and a collection of general books).

3. CICO applications maintain a list of registered users, that is, users that are authorized to check in or out items. Users can be grouped into different categories (e.g., a university library system may group users into faculty and student categories).

4. A user can checkout an item (referred to as *lending* in this section) if it is available and the check out does not violate lending policies (e.g., a policy may constrain the number of items a user in a particular category can have checked out at any time).

5. A checked out item can be checked in. The CICO domain pattern covers only those applications in which an item can be checked in only if it was previously checked out.

### 5.1.1 CICO SPS

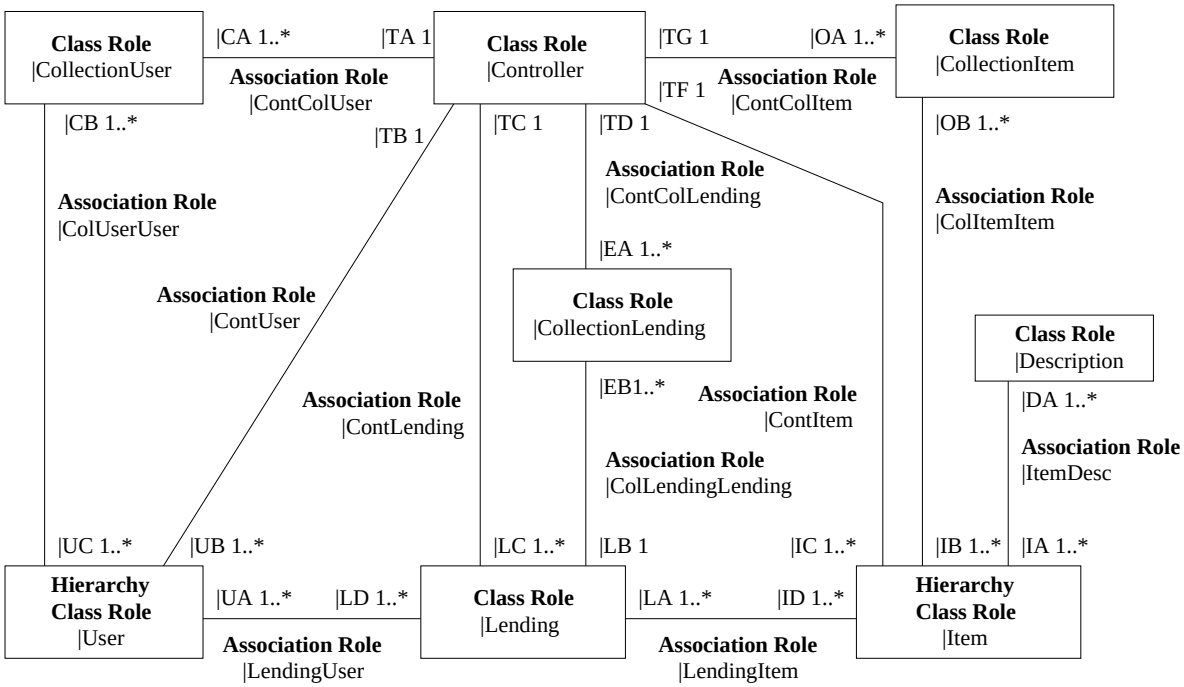



Figure 5.1: The CICO SPS

Fig. 5.1 shows the SPS characterizing static structure diagrams that conform to the CICO domain pattern. The SPS consists of roles that specify domain-specific concepts such as registered user (*User*), collection of registered users (*CollectionUser*), item check out details (*Lending*), item (*Item*), item description (*Description*), and checkin/checkout manager (*Controller*).

The folded form of role hierarchies in Fig. 5.1 is used to represent item and user role hierarchies. Fig. 5.2(a),(b) show the unfolded forms of the $User$ and $Item$ role hierarchies. These role hierarchies specify class generalization hierarchies. Fig. 5.2 also shows the CICO domain pattern classifier roles with their feature roles.

The following is an overview of the properties defined in each classifier and hierarchy role shown in Fig. 5.1. Constraint templates for some of the more interesting feature roles are described. Multiplicity and other properties specified in association-

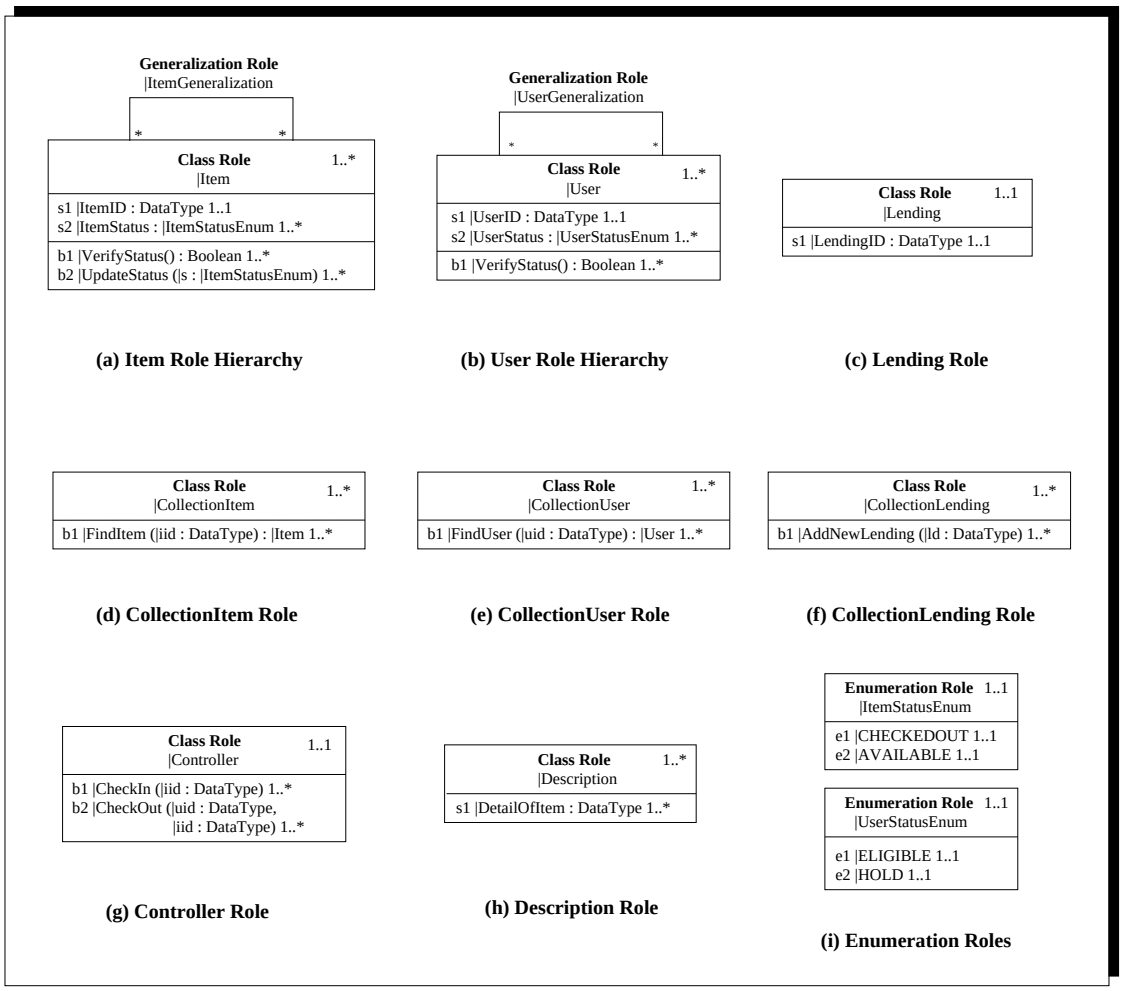



Figure 5.2: CICO Role Hierarchies

end roles are not given in this section.

**Item role hierarchy (Fig. 5.2(a)):** *Item* role has *ItemID*, *ItemStatus*, *VerifyStatus* and *UpdateStatus* feature roles. The *ItemID* role specifies a structural feature that uniquely identifies items, and the *ItemStatus* role specifies a structural feature that is used to indicate whether an item is checked in or checked out.

The types associated with structural features that conform to *ItemID* must be instances of the *DataType* metaclass. The types associated with structural features that conform to *ItemStatus* must conform to *ItemStatusEnum*.

The $VerifyStatus$ behavioral role specifies a behavior that returns true if the item is available for checkout and false otherwise:

**context** |Item :: |VerifyStatus (): Boolean

**pre** : true

**post**: *if* |ItemStatus = |AVAILABLE then result = true

*else* result = false

*UpdateStatus* specifies a behavior that changes the status of an item. For example, whenever an item is checked out (or checked in), an update behavior is performed to change the status of the item:

**context**|Item :: |UpdateStatus (|s : |ItemStatusEnum)

**pre** : true

**post**: |ItemStatus = |s

**User role hierarchy (Fig. 5.2(b)):** The *User* role has *UserID*, *UserStatus*, and $VerifyStatus$ feature roles where $VerifyStatus$ specifies a behavior that returns true if a user can check out an item and false otherwise.

**Lending role (Fig. 5.2(c)):** The *Lending* role characterizes classes defining information about a particular item checkin or checkout. The role has $LendingID$ structural feature role.

**CollectionItem role (Fig. 5.2(d)):** The $CollectionItem$ characterizes classes defining groups of items. It includes a behavioral role $FindItem$ specifying a behavior that locates an item given the item's ID.

**CollectionUser role (Fig. 5.2(e)):** The $CollectionUser$ role characterizes classes representing collections of users. It includes a behavioral role $FindUser$ specifying a

behavior that locates a user given the user's ID.

**CollectionLending role (Fig. 5.2(f)):** The *CollectionLending* role characterizes classes describing objects that maintain a collection of checkin and checkout details. It has a *AddNewLending* behavior role that characterizes a behavior that adds new lending information to the collection.

**Controller role (Fig. 5.2(g)):** The *Controller* characterizes classes that manage the checkin and checkout of items. The role includes two behavioral roles, *CheckIn* and *CheckOut* representing checkin and checkout behaviors, respectively. Constraint templates for the *CheckIn* and *CheckOut* are given below.

*CheckIn Precondition:* The item must have been checked out.

*CheckIn Postcondition:* The *FindItem* operation is called and if the operation returns the item and if the item's status indicates that the item has been checked out, the item's status is changed to indicate it is now available for checkout by calling the item's *UpdateStatus*() operation:

**context** |Controller :: |CheckIn (|id : |ID)
  **pre** : item.|ItemStatus = |CHECKEDOUT
  **post**: **let** message: OclMessage =
    |CollectionItem^^|FindItem(|id) → any(true)
    **in**
    message.hasReturned() and message.result() = item
    and item@pre.|VerifyStatus() = false
    and item^|UpdateStatus(|AVAILABLE)

*CheckOut Postcondition:* The *FindUser* and *FindItem* operations are called. If the retrieved user is eligible to checkout the item and the item is available, a record of the checkout is included in *CollectionLending* and the item status is updated to indicate

that it has been checked out:

**context** |Controller :: |CheckOut (|uid:|ID, |iid:|ID)

**pre** : true

**post**: **let** itemMessage: OclMessage =

|CollectionItem^^|FindItem(|iid) → any(true),

userMessage: OclMessage = |CollectionUser^^|FindUser(|uid)

→ any(true)

**in**

userMessage.hasReturned() and userMessage.result() = user

and user@pre.|VerifyStatus() = true

and itemMessage.hasReturned() and itemMessage.result() = item

and item@pre.|VerifyStatus() = true

and |Collectionlending → exists (lendinfo | lendInfo.oclIsNew()

and lendInfo.|User = user and lendInfo.|Item = item)

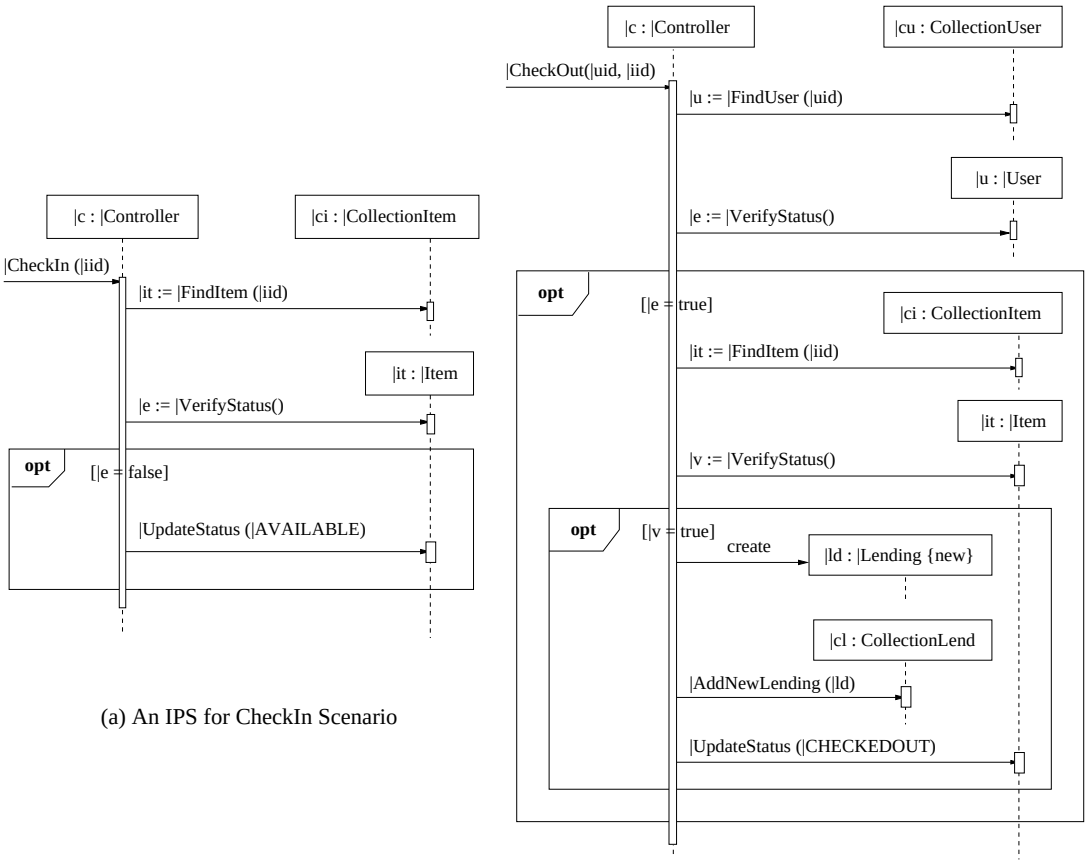
and item^|UpdateStatus(|CHECKEDOUT)

It is important to keep in mind that the above specifications describe patterns and no attempt is made to specify complete CICO behavior. For example, the above templates can be used to create initial specifications for checkin and checkout operations, which can then be extended by designers to meet application-specific requirements not captured by the pattern. This is further discussed and illustrated in Section 5.2.

### 5.1.2 CICO IPSs

Fig. 5.3(a) shows an IPS for a checkin scenario. The item is found in the item collection, and its status is checked to determine whether it is checked out. If the item is checked out the status of the item is updated.

Fig. 5.3(b) shows an IPS for a checkout scenario. An instance of a classifier that




(a) An IPS for CheckIn Scenario

(b) An IPS for CheckOut Scenario

Figure 5.3: IPSs for CheckIn and CheckOut Scenarios

conforms to *Controller* invokes *FindUser* with the user ID *uid* to obtain a matching user "*u*" from a collection of users. The status of the user is verified, and if the user is allowed to checkout the item the requested item is retrieved. The status of the item is queried to determine if it can be checked out. If the item can be checked out, a lending record is created. The record is then added to a collection of lendings (*CollectionLending*). The status of the item is updated to indicate that it has been checked out.

## 5.2 Building Models Using the CICO Pattern

Application developers can use the CICO domain pattern to produce an initial set of UML diagrams for a CICO application. Details can then be added to the diagrams in order to satisfy application-specific requirements not addressed in the initial set of diagrams. In this section, the CICO pattern is used to create UML diagrams for a library system and a vehicle rental system. A set of diagrams is produced by binding roles to elements representing solution concepts in the systems. The diagrams are then extended to satisfy application-specific requirements.

### 5.2.1 A Library System

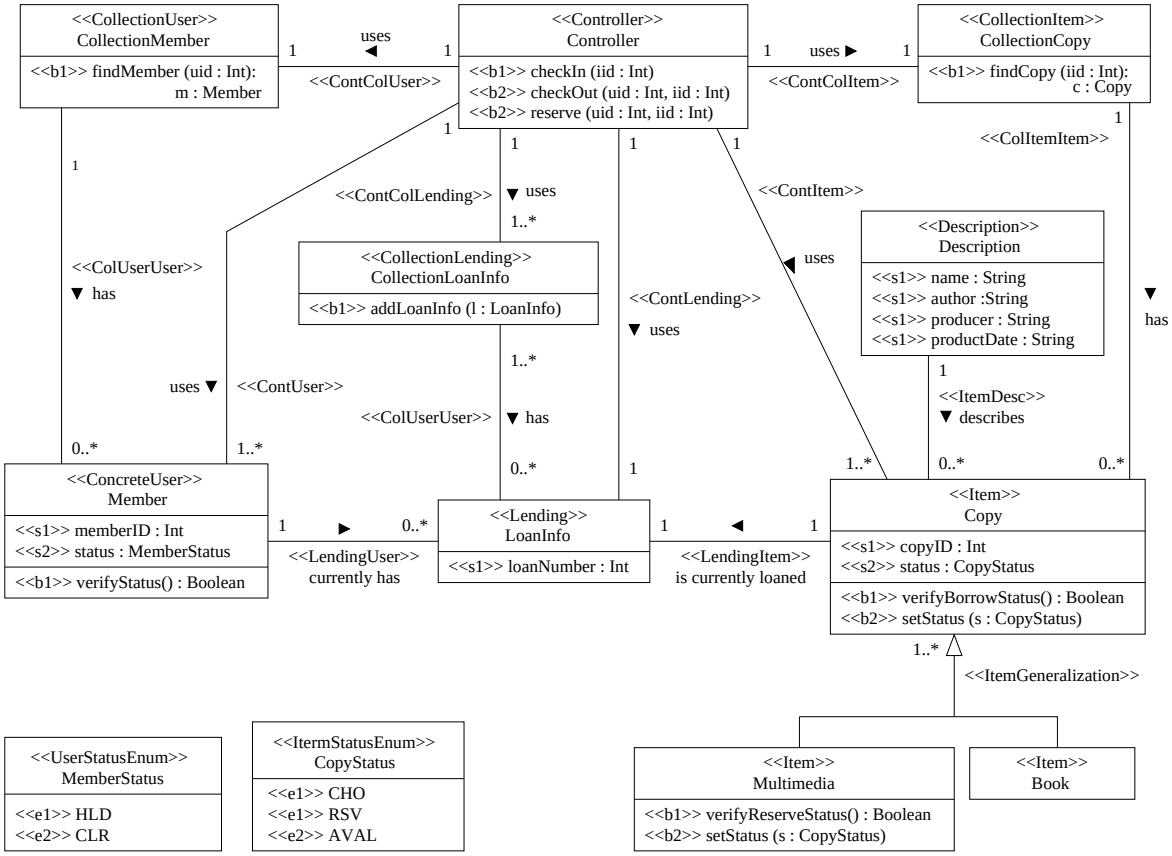



Figure 5.4: A CICO Conformant Library Class Diagram

The library system described in this section has a collection of items referred to as *copies*. A copy can be a book or a multimedia item. Users that can checkin and checkout copies are referred to as *members*. Fig. 5.4 shows a class diagram obtained from the CICO SPS. The stereotypes on the diagram elements indicate the CICO SPS roles they are bound to. Some of these bindings are described below (in the following, the symbol $\mapsto$ is to be read "binds to"):

• The *Controller* role in the SPS is bound to the *Controller* class in the Library class diagram. The *CheckOut* role is bound to an operation that checks out copies (*checkOut*) and an operation that reserves copies (*reserve*). The bindings indicate that the *reserve* operation is intended to behave as specified by the *CheckOut* role. For this application, only multimedia copies can be reserved.

• The *Copy* hierarchy is obtained from the *Item* hierarchy defined in the CICO SPS. For example, $Copy \mapsto Item$, $Multimedia \mapsto Item$, and $Book \mapsto Item$ are bindings that produce the classes in the *Copy* hierarchy. The two generalization relationships shown in Fig. 5.4 are bound to the *ItemGeneralization* role. The *copyID* attribute is the only one that plays the $ItemID$ role as required by the role's binding multiplicity (1..1).

• The $VerifyStatus$ feature role in the *Item* role is bound to *verifyBorrowStatus* in the *Copy* class and to $verifyReserveStatus$ in the *Multimedia* class. The *verifyBorrowStatus* returns true if the copy is available for checkout and false otherwise. The *verifyReserveStatus* returns true if the copy can be reserved and false otherwise. The *Multimedia* class has two operations that are bound to the $VerifyStatus$ role: $verifyBorrowStatus$ inherited from *Copy* and $verifyReserveStatus$ defined in the class. This is consistent with the binding multiplicity (1..$*$) associated with the $VerifyStatus$ role. The pre- and postconditions of the $verifyBorrowStatus$ operation is obtained by instantiating the constraint template for the *VerifyStatus* role:

**context** Copy :: verifyBorrowStatus (): Boolean

**pre** : true

**post**: *if* status = AVAL then result = true

*else* result = false

• The following are bindings for enumeration type values:

$CHO \mapsto CHECKEDOUT$, $RSV \mapsto CHECKEDOUT$, $HLD \mapsto HOLD$, $CLR \mapsto ELIGIBLE$.

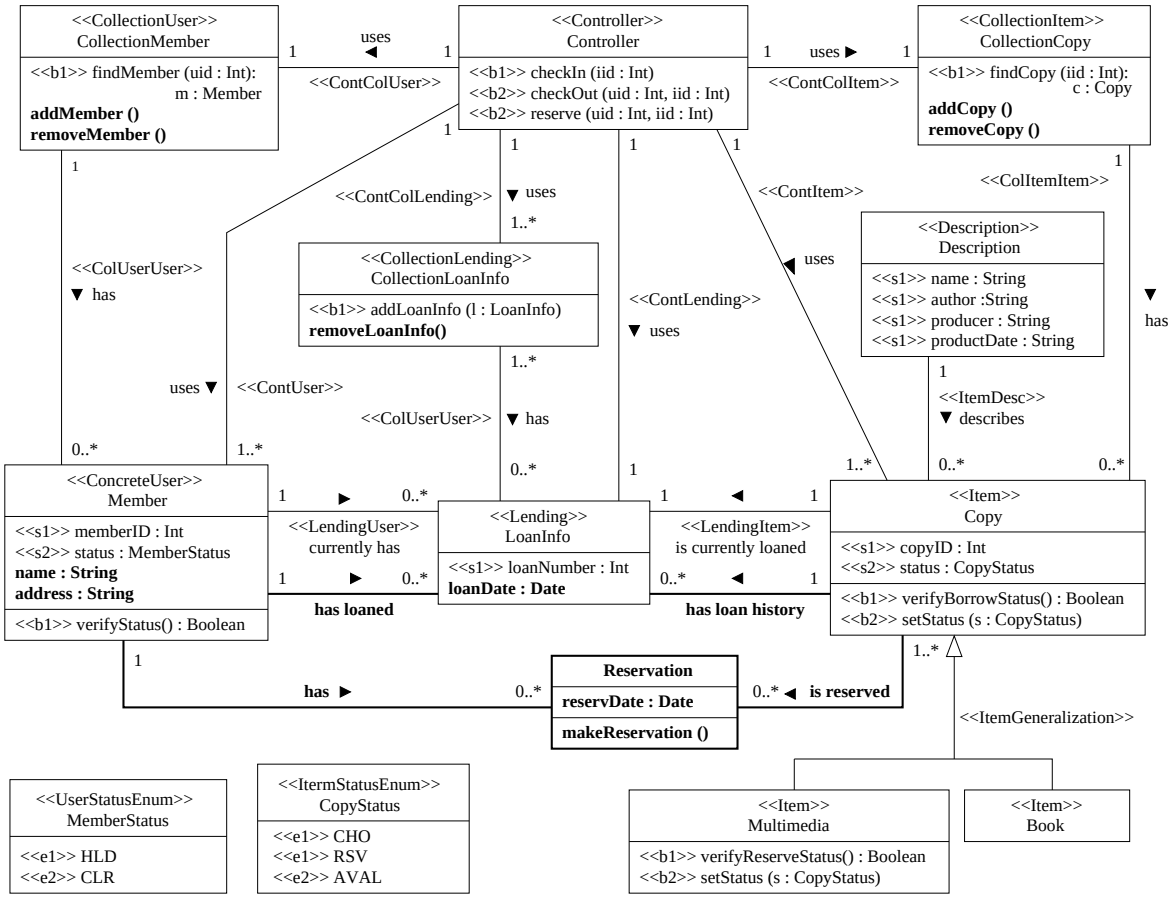



Figure 5.5: The Completed Library Class Diagram

In addition to the properties specified in the CICO pattern, the library system is required to (1) track lending history, (2) record reservations, (3) maintain contact information on members, (4) record the date a copy is checked out, (5) support adding and removing members and copies, and (6) support removal of lending information

from the system. Additional class diagram elements are needed to address the above requirements. Fig. 5.5 shows the completed library system class diagram (additional elements are shown in bold typeface). Some of the elements added to the diagram are described below:

• The "*has loan history*" and "*has loaned*" associations are added to support tracking of lending history.

• The *Reservation* class and attached associations are added to support recording of reservations.

• The attributes *name* and *address* in $Member$ are added because this information is used in the application to support activities that require contacting the member.

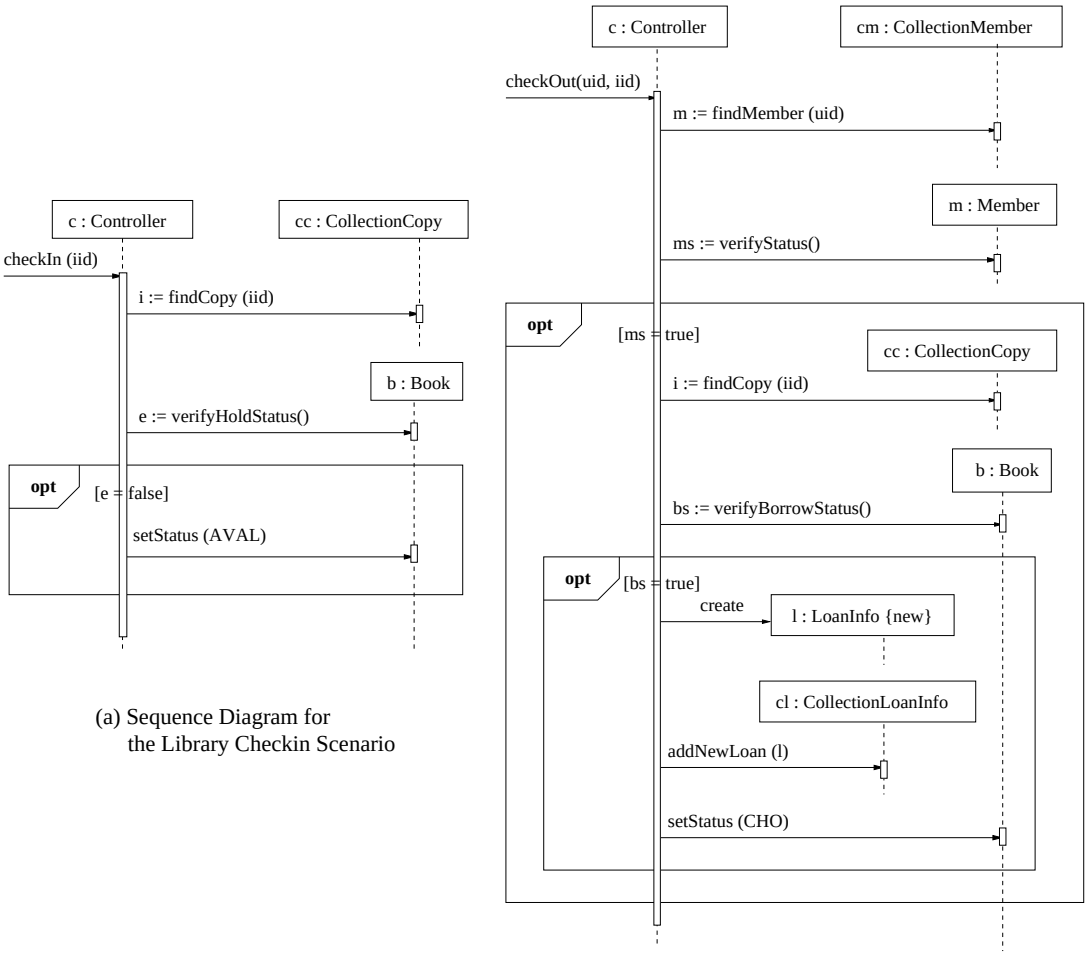
Sequence diagrams can also be obtained from the CICO IPSs by binding sequence diagram model elements to roles. The class and feature bindings are the same as those used to produce the class diagram. Figure 5.6 shows sequence diagrams obtained from the CICO IPSs in Fig. 5.3.

### 5.2.2 A Vehicle Rental System

Fig. 5.7 shows a vehicle rental class diagram created using the CICO SPS. The diagram describes a design in which customers rent vehicles. Two types of vehicles can be rented: trucks and leisure vehicles. Unlike the library example, the $Item$ hierarchy is bound to a multi-level generalization hierarchy: the $Vehicle$ class is specialized by $Truck$ and *Leisure* and *Leisure* is further specialized by *Van* and *Sedan*.

The class diagram also includes classifiers and other diagram elements not specified by the CICO SPS. Like the Library system, there is a *Reservation* class. There is also an $InsurancyPolicy$ class associated with the $Vehicle$ class. An interface class, $CollectionVehicle$, is also added to the model to act as a common interface for the different types of vehicle collections.

Vehicle rental scenarios obtained from the CICO IPSs are shown in Fig. 5.8.




(a) Sequence Diagram for the Library Checkin Scenario

(b) Sequence Diagram for the Library Checkout Scenario

Figure 5.6: CICO Conformant Library Scenarios

## 5.3 Related Work

Early work on domain-specific languages [106] tended to focus on providing language interfaces for assembling code components into programs. These languages focus on downstream development phases (detailed design specification and implementation in code). Domain-specific design specification and architectural languages have begun to appear (e.g., see [42]). We are not aware of any approaches that allow developers to specify reusable static and behavioral UML models and use them to develop

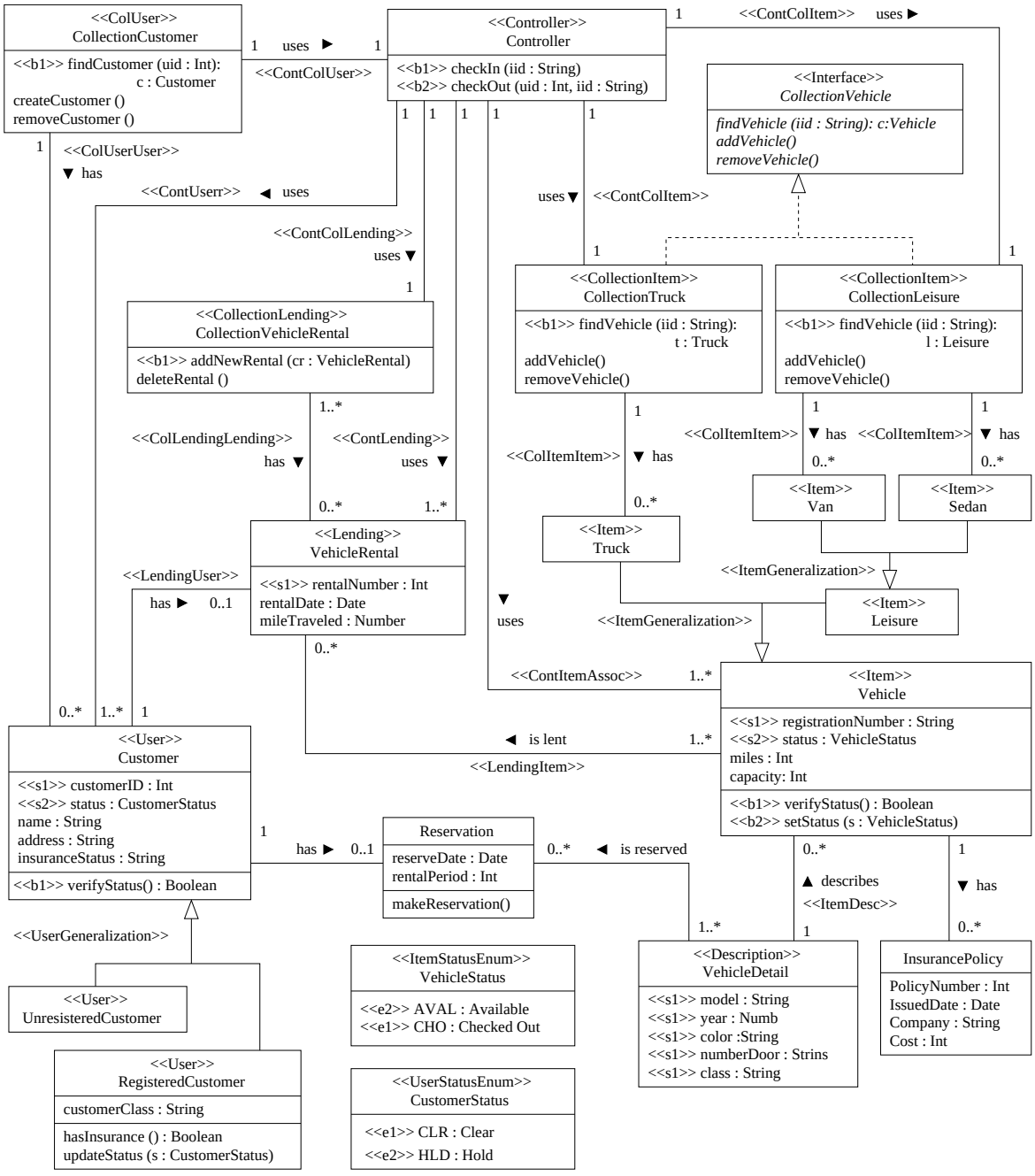
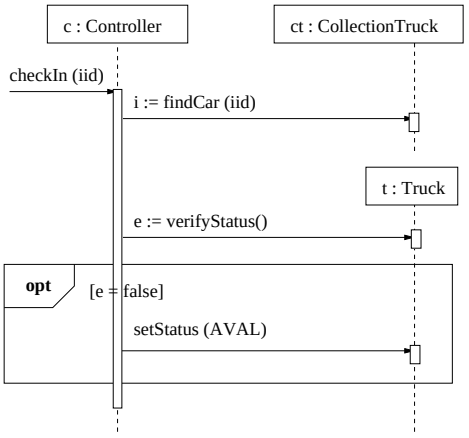


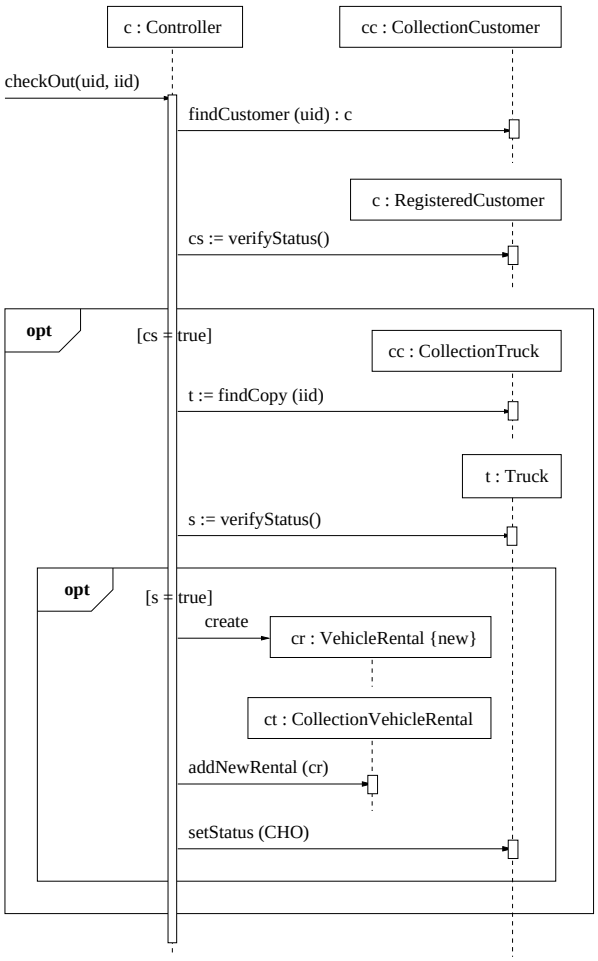

Figure 5.7: A CICO Conformant Vehicle Rental Class Diagram

applications.

Other forms of reusable experiences packaged for vertical reuse are frameworks [88] and domain-specific architectures (e.g., see [29, 43, 67, 101]). There is a considerable




(a) Sequence Diagram for
the Car Rental Checkin Scenario




(b) Sequence Diagram for the Car Rental Checkout Scenario

Figure 5.8: CICO Conformant Vehicle Rental Scenarios

body of work on domain engineering processes and domain modeling notations (e.g., see [72, 6, 43, 101, 54]). Our approach can complement the above efforts by providing a notation for representing patterns. Tool vendors can use the pattern specifications to create specializations of UML constructs that have the properties specified by the patterns.

Pattern languages for specifying Business Resource Management patterns have been developed (e.g., see [14, 15]). Braga *et al.* [14] use Class diagrams to describe three patterns related to resource rental, trade and maintenance activities. They use

the diagrams to stamp out class diagrams describing application-specific activities such as library service, medical attendance, video rental, and real estate rental. Their approach supports only specification of structural properties.

## 5.4 Lessons Learned

This chapter has described the potential of the RBML to define a domain pattern as a domain-specific UML sub-language that defines a family of UML models in the domain.

Modeling tools that allow users to define and utilize domain patterns are needed to support systematic reuse of domain-specific modeling experience. One can envisage a development environment in which domain engineers develop domain patterns and embed them in modeling tools so that application developers can use them to develop application-specific models. The RBML can be used as the base for such modeling environments. A prototype tool, RBML-Pattern Instantiator (RBML-PI) (see Chapter 7), that supports this approach has been developed.

Another approach would be based on the UML sub-language defined by a domain pattern. A tool that exploits this view of a domain pattern will allow a modeler to construct models that conform to the sub-language. Such a tool would look like a typical modeling tool, that is, the tool would have a symbol area and a drawing area. The symbol area includes symbols representing the elements characterized by the domain pattern. In the case of the CICO pattern, there will be symbols for *Item* and *User* classes (or hierarchies). Modelers would select a symbol from the symbol area and drag it onto the drawing area. For example, dragging an *Item* class to the drawing area would result in an item class being displayed with slots for features that play the feature roles defined in the *Item* role. The metamodel (i.e., the specialized UML metamodel defined by the pattern) constrains how the symbols can be connected together in the drawing area.

The RBML can also be used to define lightweight UML profiles. A lightweight profile defines an extended UML metamodel that does not add or remove UML metamodel elements or constraints, that is, it simply extends the features associated with existing metamodel elements. A problem with existing mechanisms used to define UML profiles is that they are defined informally using text. Such informality becomes an obstacle to the development of tools. The RBML can be used to specify profiles in a form that can facilitate the development of tool support.

# Chapter 6

# Using the RBML to Specifying Access Control Aspects

In this chapter, RBML templates, a special form of RBML specifications that uses parameters instead of roles, are used to specify two popular access control aspects, Role-Based Access Control (RBAC) [26] and Mandatory Access Control (MAC) [91], and propose a Hybrid Access Control (HAC) by merging RBAC and MAC. These specifications can be used to incorporate the access control policies into a model. RBML templates are used to support systematic composition of the policies with a model.

An RBML template consists of an SPS template and IPS templates which are restricted forms of SPSs and IPSs, respectively. An SPS template comprises of templates that correspond to SPS roles (e.g., classifier roles, association roles). For example, a classifier template correspond to a classifier role. Templates are instantiated by binding their parameters to application elements. In this work, only SPS templates are used to characterize the static structure of the policies.

The rest of the section is organized as follows. Section 6.1 gives an overview of aspects. Section 6.2.4 describes how RBML templates can be used to specify RBAC, MAC, and HAC. Section 6.3 illustrates how the RBAC specification can be used to develop secure systems using a banking application. Section 6.4 gives an overview of

related work, and Section 6.5 concludes the section with the lessons learned from this study.

## 6.1 Overview of Aspects

Separation of concerns that cut across a system is essential in the development of the system to reduce complexity. There have been programming languages and design methods that provide useful modularity mechanisms. They have, however, limitation for concerns that crosscut over the system. Such concerns cannot be encapsulated in single modules. Aspect-oriented software development (AOSD) is a new software development paradigm for separation of concerns (SOC) which provides mechanism to localize the crosscutting concerns and weave them into the system components.

AOSD allows to improve software quality components such as understandability, adaptability, maintainability, and reusability by localizing concerns. Realizing these benefits requires addressing some of non-trivial questions and issues like 1) what types of pervasive dependability concerns can be effectively encapsulated, and what forms of encapsulations are appropriate, 2) how to resolve conflicts when a system is required to satisfy a number of conflicting dependency goals, 3) how should weaving (composition) be performed, and 4) how does one determine whether the woven model meets desired goals.

AOSD is rooted from programming initiatives known as aspect-oriented programming (AOP) [56]. AOP promotes code reuse by localizing the implementation of design features that cut across multiple functional units. In AOP, an aspect is a concept that cleanly encapsulates a crosscutting concern that cannot be localized to a single class. A simple example is writing logging messages whenever certain methods are called. Instead of manually inserting the message logging code to many classes, an aspect that includes the code to log the message can be used for systematic insertion to the classes using the insertion points (known as joint points).

Research initiatives have taken to develop techniques and mechanisms that provide support for multi-dimensional separation of concerns in the design phase of software development known as aspect-oriented modeling (AOM) [38]. AOM techniques allow developers to encapsulate dependability concerns in aspect models that can be composed with models of functionality. While AOM techniques share many benefits with AOP techniques, the abstraction provided by AOM techniques help designers communicate aspects and increase traceability from design to code which can be automatically generated. In this work, design aspects are viewed as a particular type of design patterns that addresses cross-cutting concerns.

There are many problems to be solved in AOM. One of the major problems is to develop aspect modeling languages. Using object-oriented methods and languages (e.g., UML) for modeling aspects can be considered as a partial solution, but not a suitable solution since UML was not designed to provide constructs to describe crosscutting concerns. Special support for designing aspects is needed to support the design process and traceability in AOM.

## 6.2 Specifying Access Control Aspects as Patterns

An access control policy provides a set of rules, concepts, and guidelines for defining access control policies. Two popular policies are Role Based Access Control (RBAC) and Mandatory Access Control (MAC). In this section, RBAC and MAC are specified. The resulting specifications are then merged to produce a Hybrid Access Control (HAC) specification. HAC can be used when organizations have to merge policies that are based on RBAC and MAC, or as a base for defining access control policies in a military domain where RBAC and MAC policies are often used together.

An access control policy is described by a set of RBML templates, a restricted form of RBML specifications that uses parameters instead of roles. Instantiating an RBML template results in a UML model that describes an application-specific policy

that conforms to the access control policy.

RBAC and MAC specifications are merged to produce a HAC specification. To merge the two access control policies, the following are first determined: (1) the elements in the two policies that will be merged, (2) the elements that will appear "as is" in the merged specification, and (3) the elements that will be modified or removed in the composed specification.

### 6.2.1 Overview of RBAC

RBAC constraints can be organized as follows: *Core RBAC*, *Hierarchical RBAC*, *Static Separation of Duty Relations*, and *Dynamic Separation of Duty Relations*.

Core RBAC embodies the essential aspects of RBAC. The constraints specified by Core RBAC are present in any RBAC model. Core RBAC requires that users be assigned to roles (job function), roles be associated with permissions (approval to perform an operation on an object), and users acquire permissions by being assigned to roles. Core RBAC does not place any constraint on the cardinalities of the user-role assignment relation or the permission-role association. Core RBAC also includes the notion of user sessions. A user establishes a session during which he activates a subset of the roles assigned to him. Each user can activate multiple sessions; however, each session is associated with only one user. The operations that a user can perform in a session depend on the roles activated in that session and the permissions associated with those roles.

Hierarchical RBAC adds features supporting role hierarchies. Hierarchies are used to describe a structure of roles in an organization. Role hierarchies define an inheritance relation among the roles. Role $r1$ inherits from role $r2$ only if all permissions of $r2$ are also permissions of $r1$ and all users of $r1$ are also users of $r2$. The inheritance relationship is reflexive, transitive and anti-symmetric.

Static Separation of Duty (SSD) relations are necessary to prevent conflict of

interests that arise when a user gains permissions associated with conflicting roles (roles that cannot be assigned to the same user). SSD relations are specified for any pair of roles that conflict. An SSD relation places a constraint on the assignment of users to roles, that is, membership in one role that takes part in the relation prevents the user from being a member of the other conflicting role. SSD relations are symmetric, but it is neither reflexive nor transitive. SSD relations may exist in the absence of role hierarchies (referred to as SSD RBAC), or in the presence of role hierarchies (referred to as hierarchical SSD RBAC). The presence of role hierarchies complicates the enforcement of the SSD relations: before assigning users to roles not only should one check the direct user assignments but also the indirect user assignments that occur due to the presence of the role hierarchies.

Dynamic Separation of Duty (DSD) relations aim to prevent conflict of interests as well. A DSD relation places a constraint on the roles that can be activated in a user's session. If one role that takes part in a DSD relation is activated, the user cannot activate the related (conflicting) role in the same session. Fig. 6.1 shows a model of RBAC [26].

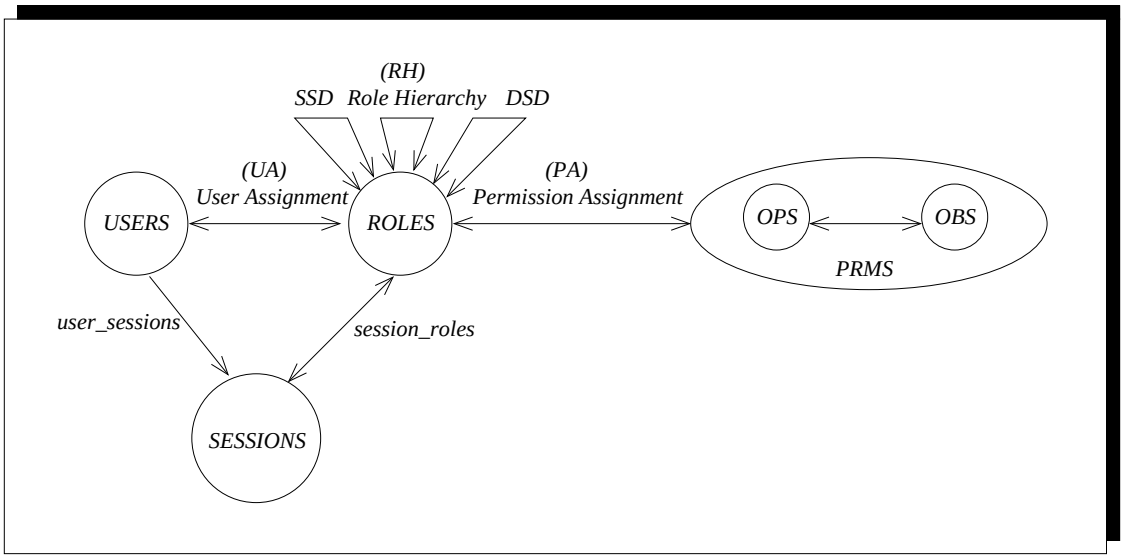



Figure 6.1: RBAC

The RBAC in Fig. 6.1 consists of: 1) a set of users ($USERS$) where a user is an intelligent autonomous agent, 2) a set of roles ($ROLES$) where a role is a job function, 3) a set of objects ($OBS$) where an object is an entity that contains or receives information, 4) a set of operations ($OPS$) where an operation is an executable image of a program, and 5) a set of permissions ($PRMS$) where a permission is an approval to perform an operation on objects. The cardinalities of the relationships are indicated by the absence (denoting one) or presence of arrows (denoting many) on the corresponding associations. For example, the association of user to session is one-to-many. All other associations shown in the figure are many-to-many. The association labeled *Role Hierarchy* defines the inheritance relationship among roles. The association labeled *SSD* specifies the roles that conflict with each other. The association labeled *DSD* specifies the roles that cannot be activated within a session by the same user.

An application that satisfies the constraints specified by RBAC is referred to as a conforming model, otherwise it is said to be non-conforming. An example of a hierarchical SSD constraint is given below (see [26] for a more formal definition):

- Let $assigned\_users(r{:}ROLE)$ be the set of all users that are assigned to the role $r$.
- Let $SSD$ be the collection of all pairs ($r1$, $r2$) such that users cannot be assigned to both role $r1$ and $r2$.
- Let $inherits\_from(r{:}ROLE)$ be the set of all roles $\{r_1, r_2, \ldots, r_n\}$ such that permissions of $r_j$ $(j = 1 \ldots n)$ are also permissions of role $r$ and users of role $r$ are also users of role $r_j$.
- Let $senior\_role(r{:}ROLE)$ be the set of all roles that inherit from role $r$.
- Let $authorized\_users(r{:}ROLE)$ be the set of all users that are directly assigned

to $r$ and any role belonging to $senior\_role(r)$.

- Hierarchical SSD Constraint:

  $\forall\ (r1, r2) \in SSD$, $authorized\_users(r1) \cap authorized\_users(r2) = \emptyset$

Consider a banking application. The roles of interest are: *FinancialSupervisor*, *AccountsReceivableClerk*, and *BillingClerk*. The role $FinancialSupervisor$ inherits from the roles $AccountsReceivableClerk$ and $BillingClerk$. The roles $AccountsReceivableClerk$ has an SSD relationship with the role $BillingClerk$. Let users $Finn$, $Adam$, and $Bill$ be assigned to the roles $FinancialSupervisor$, $AccountsReceivableClerk$, and $BillingClerk$ respectively. For the given application, the following configuration is given:

- $assigned\_users(FinancialSupervisor) = \{Finn\}$
- $assigned\_users(AccountsReceivableClerk) = \{Adam\}$
- $assigned\_users(BillingClerk) = \{Bill\}$
- $inherits\_from(FinancialSupervisor) = \{AccountsReceivableClerk, BillingClerk\}$
- $senior\_role(AccountsReceivableClerk) = \{FinancialSupervisor\}$
- $senior\_role(BillingClerk) = \{FinancialSupervisor\}$
- $SSD = \{(AccountsReceivableClerk, BillingClerk)\}$
- $authorized\_users(AccountsReceivableClerk) = \{Adam, Finn\}$
- $authorized\_users(BillingClerk) = \{Bill, Finn\}$

The above is an example of a non-conforming application because it does not satisfy the hierarchical SSD constraint:

$authorized\_users(AccountsReceivableClerk) \cap authorized\_users(BillingClerk)$
$= \{Finn\} \neq \emptyset$

### 6.2.2 Specifying RBAC

Based on the description of RBAC given in the previous subsection, participants of *User*, *Role*, *Session*, *Permission*, *Object*, and *Operation* are identified as classifier templates for hierarchical RBAC with SSD and DSD as shown in Fig. 6.2. The symbol "|" denotes parameters.

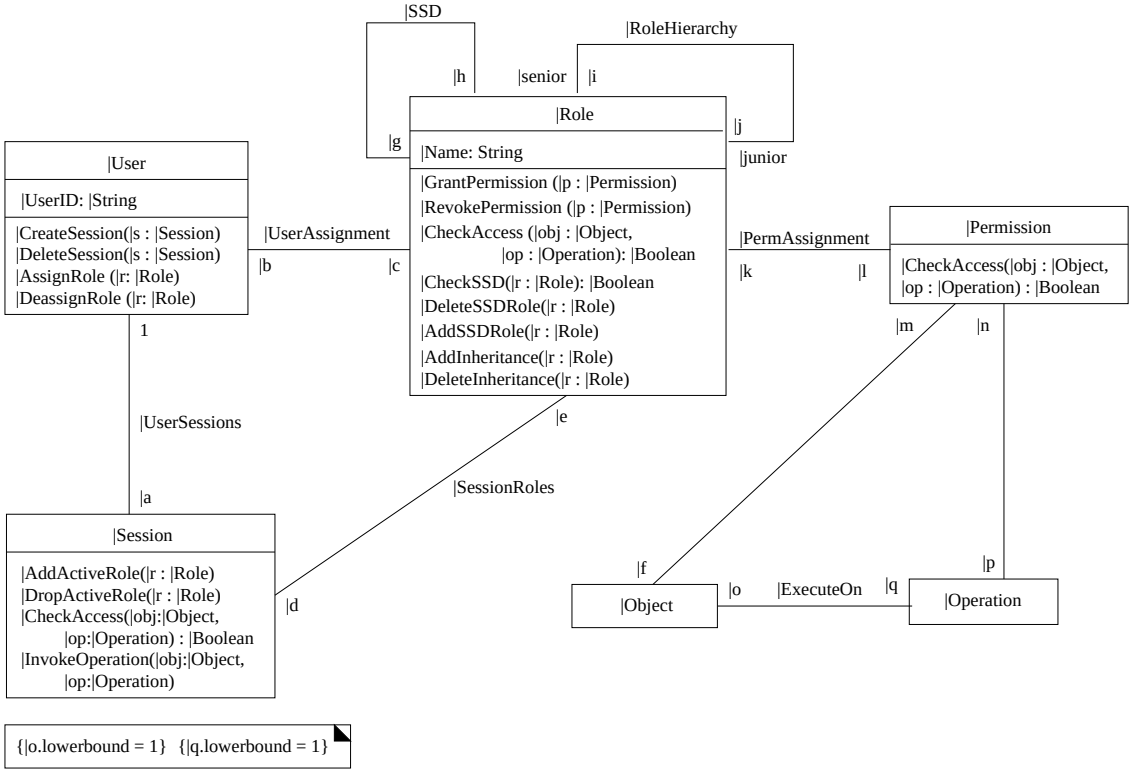



Figure 6.2: RBAC Template

The *User* class template represents users. A user can create a new session (*CreateSession*), delete a session (*DeleteSession*), associate self with a new role *AssignRole* and remove an associated role (*DeassignRole*). A *UserSessions* link (i.e., an instance of an association obtained by binding the parameters of $UserSessions$ to values) is created by a *CreateSession* operation (i.e., an operation obtained by binding the operation parameters to values) and deleted by a *DeleteSession* operation. The operation *AssignRole* creates a *UserAssignment* link; the *DeassignRole* removes a *UserAssign-*

*ment* link.

Unlike the multiplicities on association-end roles in RBML specifications which constraint the number of association-ends that can be attached to conforming classifiers, a multiplicity on an association-end in RBML templates specifies the multiplicity on the corresponding association-end in an instantiated model. For example, in Fig. 6.2 the multiplicity "1" on the *UserSessions* association-end specifies the multiplicity at an instance of the *User* template that a session can only be associated with one user.

The *Role* class template is used to produce classes representing roles with behavior that associates a new permission with the role (*GrantPermission*), deletes an existing permission associated with the role (*RevokePermission*), adds an immediate inheriting role (*AddInheritance*), deletes an immediate inheriting role (*DeleteInheritance*), adds a role to the set of conflicting roles (*AddSSDRole*), deletes a role from the existing set of conflicting roles (*DeleteSSDRole*), checks the SSDs associated with a role in the presence of hierarchies (*CheckSSD*), checks whether a role has some permission (*CheckAccess*), checks whether there is a DSD relation between two roles or not (*CheckDSD*), deletes a role from the existing set of roles in a DSD relation (*DeleteDSDRole*), and adds a role to the set of roles in DSD relation (*AddDSDRole*).

The *Session* class template is associated with the operations: *AddActiveRole* (activates a role in a session), *DropActiveRole* (deactivates a role in a session), and *CheckAccess* (checks whether the role has the permission to perform an operation on an object). The operations *GrantPermission* and *RevokePermission* are responsible for creating and deleting, respectively, *PermAssignment* links. An *SSD* link is created by the *AddSSDRole* operation; this link is deleted by the *DeleteSSDRole* relation. The operation *AddInheritance* adds a link *RoleHierarchy*; *DeleteInheritance* deletes this link.

The *Permission* class template is associated with one operation *CheckAccess* that

checks whether the role has the permission to perform the operation on the object.

The OCL constraints in Fig. 6.2 restrict the values that can be bound to multiplicity parameters. {|o.lowerbound = 1} restricts the multiplicities that can be bound to the parameter *o* to ranges that have a lower bound of 1.

Each operation parameter is associated with an OCL template expression that produces OCL pre- and post-conditions when the template parameters are bound to values. Pre- and post-condition templates associated with the *CreateSession* and *GrantPermission* operations are given below:

**context** |User::|CreateSession():(|s:|Session)

**post**: result = |s and

|s.oclIsNew() = true and

self.|Session → includes(|s)

**context** |Role::|GrantPermission (|p:|Permission)

**pre**: self.|Permission → excludes(|p)

**post**: self.|Permission → includes(|p)

RBAC constraints that restrict SSD and DSD relationships are expressed as OCL template expressions. Examples of these constraints are given below:

- SSD constraint. A user cannot be assigned to two roles that are involved in an SSD relation:

  **context** |User **inv**:

  self.|Role → forAll(r1, r2|r1.|SSD → excludes(r2))

- Hierarchical SSD constraint. There cannot be roles in an SSD relation which have the same senior role:

  **context** |Role **inv**:

**let** allSenior(r1) = r1.senior → union(r1.senior → collect(r2|allSenior(r2)))

**in**

self.|SSD → forAll(r1|allSenior(r1) → excludesAll(allSenior(self))

- DSD constraint. A user cannot activate two roles in DSD relation within a session:

**context** |User **inv**:

|self.|Session.|Activates → forAll(r1, r2|r1.|DSD → excludes(r2))

### 6.2.3 Specifying MAC

The MAC policy used in this research is adapted from the Bell-La Padula model [91]. The Bell-La Padula model is defined in terms of a security structure $(L, \geq)$. $L$ is the set of security levels (e.g., top secret, secret, confidential, unclassified), and $\geq$ is the dominance relation between these levels that a higher security level dominates a lower security level. The main components of this model are objects, users, and subjects. Objects contain or receive information. Each object in the Bell-La Padula model is associated with a security level which is called the classification of the object. User, in this model, refers to human beings. Each user is also associated with a security level that is referred to as the clearance of the user. Each user is associated with one or more subjects. Subjects are processes that are executed on behalf of some user logged in at a specific security level. The security level associated with a subject is the same as the level at which the user has logged in.

The mandatory access control policies in the Bell-La Padula model are specified in terms of subjects and objects. The policies for reading and writing objects are given by the Simple Security and Restricted-$\star$ Properties.

- *Simple Security Property*: A subject $S$ may have read access to an object $O$ only if the security level of the subject $L(S)$ dominates the security level of the

object $L(O)$, that is, $L(S) \geq L(O)$.

- *Restricted-$\star$ Property*: A subject $S$ may have write access to an object $O$ only if the security level of the subject $L(S)$ equals to the security level of the object $L(O)$, that is, $L(O) = L(S)$.

The static structural aspects of the MAC is described in the SPS template shown in Fig. 6.3.

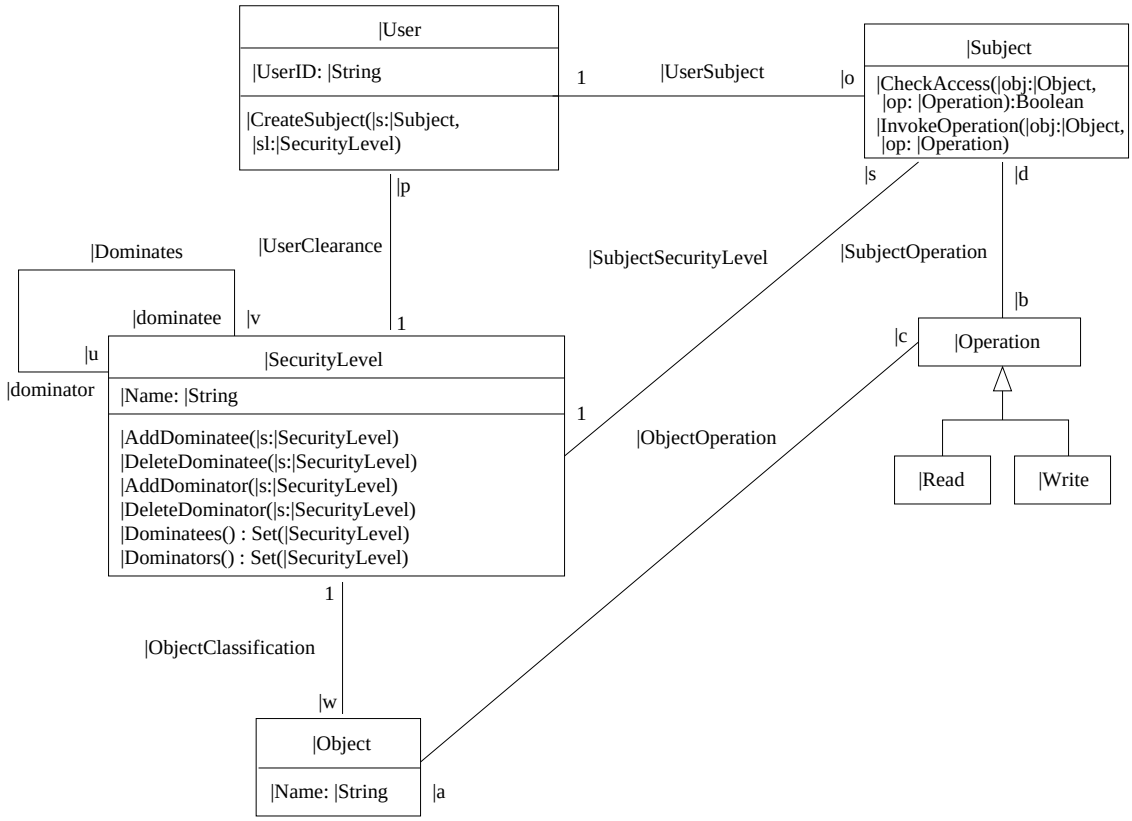



Figure 6.3: MAC Template

The *User* class template represents users. A user can create a new session (*CreateSubject*). The *Subject* class template subjects with behavior that checks whether the requested operation can be invoked on the object (*CheckAccess* and executes the requested operation on the object *InvokeOperation*.

The *SecurityLevel* class template represents the structure of security levels with behavior that adds a new dominatee in the structure (*AddDominatee*), deletes an existing dominatee from the structure (*DeleteDominatee*), adds a new dominator in the structure (*AddDominator*), deletes an existing dominator from the structure (*DeleteDominator*), get the set of security levels of dominatees (*Dominatees*, and get the set of security levels of dominators (*Dominators*.

MAC constraints that restrict reading and writing objects and user access to subjects are expressed as follows:

- Reading constraint. The security level of the subject dominates the security level of the object:

  **context** |Read**inv**:

  self.|Subject.|SecurityLevel $\geq$ self.|Object.|SecurityLevel

- Writing Constraint. The security level of the subject equals to the security level of the object:

  **context** |Write **inv**:

  self.|Subject.|SecurityLevel = self.|Object.|SecurityLevel

- User access to subjects. The security level of the user accessing the subject dominates or equal to the security level of the subject:

  **context** |Subject **inv**:

  (self.|User.|SecurityLevel.|dominator $\rightarrow$ includes (self.|SecurityLevel)) or

  (self.|User.|SecurityLevel = self.|SecurityLevel)

### 6.2.4 Specifying HAC

Situations can arise in which organizations have to merge policies that are based on different access control policies. Examples of such situations are when organizations

using different access control policies are merged (through acquisitions or consolidations) or when the organizations need to share protected resources in a joint endeavor. In such cases one needs to ensure that unauthorized persons are not inadvertently given access to protected resources, and that authorized persons are not denied access to resources as a result of emergent properties of the merged policies.

This subsection describes how RBAC and MAC can be merged. The following describes the steps to merge them:

**Step 1** Identify the entities in each of the access control policies.

**Step 2** Compare the definition of an entity in one specification to that of another in the second specification, and determine which represent similar concepts and which represent dissimilar concepts.

**Step 3** Matched entities (those representing similar concepts as determined in the previous step) are merged in the composed specification.

**Step 4** Dissimilar entities that must be present in the composed specification are added "as is" to the composed specification, or are modified (as determined in step 2). Entities that have been identified for elimination are not added to the composed specification.

The following are observed about the elements in MAC and RBAC:

- *User*, *Object*, *Operation* are used in both the specifications and they each refer to the same concepts (see Section 6.2.1 and 6.2.3).
- *Subject* (used in MAC) and *Session* (used in RBAC) refer to the same concept [**?**].
- *SecurityLevel* appears in one specification (MAC) but not in the other (RBAC).

Based on the above observations, the algorithm is applied to generate the hybrid access control specification shown in Fig. 6.4.

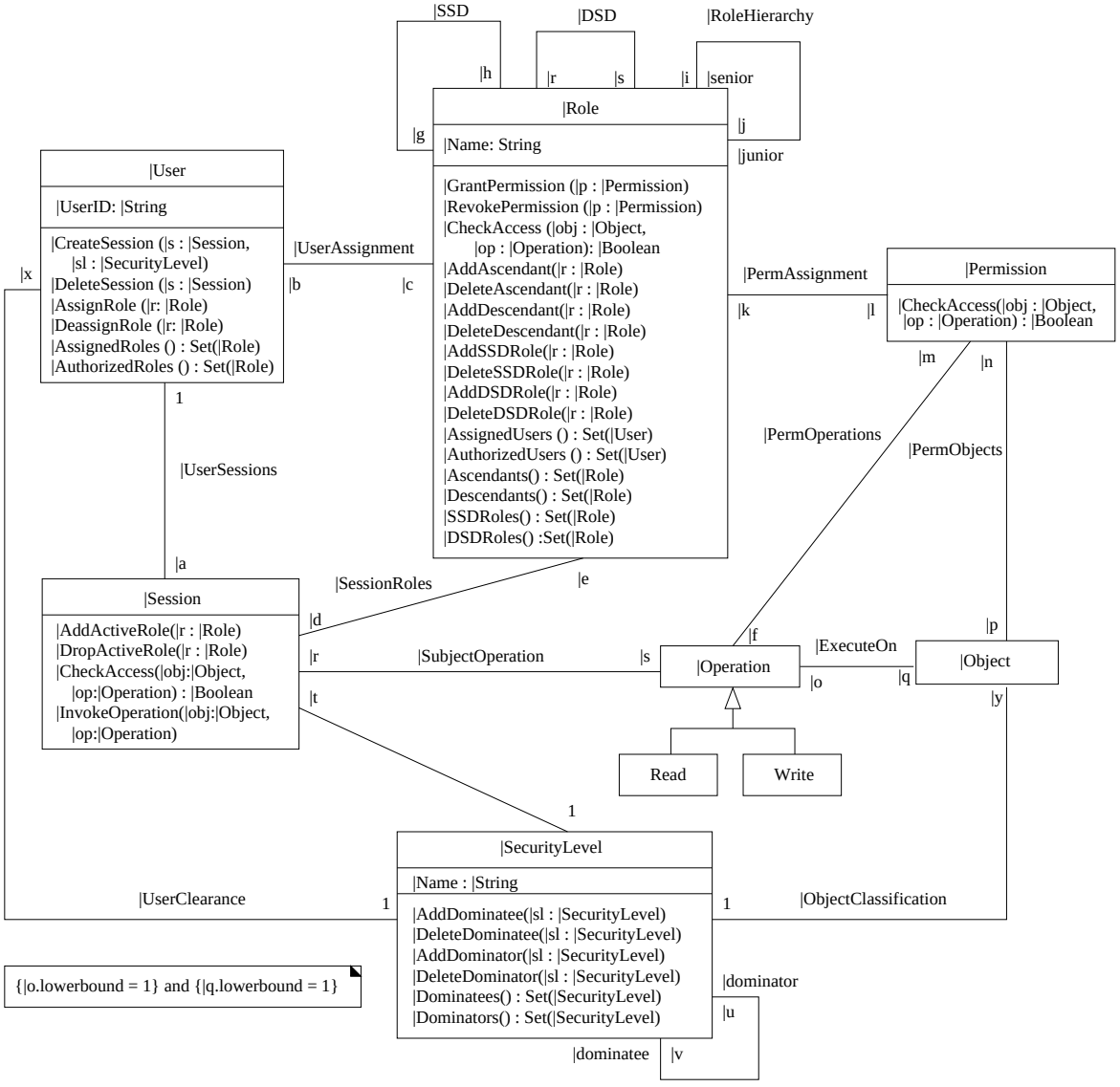



Figure 6.4: HAC Template

1. The *User* elements are merged in the two specifications. The attribute parameters are identical in both RBAC and MAC. The MAC specifies only one operation *CreateSubject* for *User*. The merged element includes the operations from the *User* element in RBAC and also those of MAC. The merged element

appears in the hybrid specification and this is referred to as *User*.

2. The elements *Object* and *Operation* in both the specifications are identical. Each of these elements are added to the hybrid specification.

3. The element *SecurityLevel* (present in MAC but not RBAC) is added to the hybrid specification.

4. The elements *Session* and *Subject* refer to the same concept. These elements are merged. Since *Subject* is associated with *SecurityLevel*, the merged element is now associated with a security level. The merged element is referred to as *Session* in the hybrid specification. From comparing these two elements in the two specifications, RBAC *Session* is found to have operations, such as, *AddActiveRole* and *DropActiveRole* that are not present in *Subject*. These operations are added in the *Session* element of HAC. Both *Subject* and *Session* have *CheckAccess* and *InvokeOperation*. These operations if different must be merged. The *CheckAccess* operation in HAC is changed to reflect this. The merging of *Subject* and *Session* also affects other elements. For instance, consider the class template *User* in HAC. From Step 1, the operations of *User* are *CreateSession*, *DeleteSession*, *AssignRole*, *DeassignRole*, and *CreateSubject*. Since *Subject* and *Session* are merged, only one operation called *CreateSession* is needed (because the merged entity in HAC is called *Session*). Moreover, the *CreateSession* of RBAC must be changed because now the security level also has to be passed as a parameter.

## 6.3 Applying the RBAC Specification

To illustrate application of the RBAC specification to a model, a simple banking application is used taken from [17]. The application is used by various bank officers

to perform transactions on customer deposit accounts, customer loan accounts, ledger posting rules, and general ledger reports. The transactions include 1) create, delete, or modify customer deposit accounts, 2) create or modify customer loan accounts, 3) modify the ledger posting rules, and 4) create general ledger report. A class diagram (the primary model) for the application is shown in Fig. 6.5.

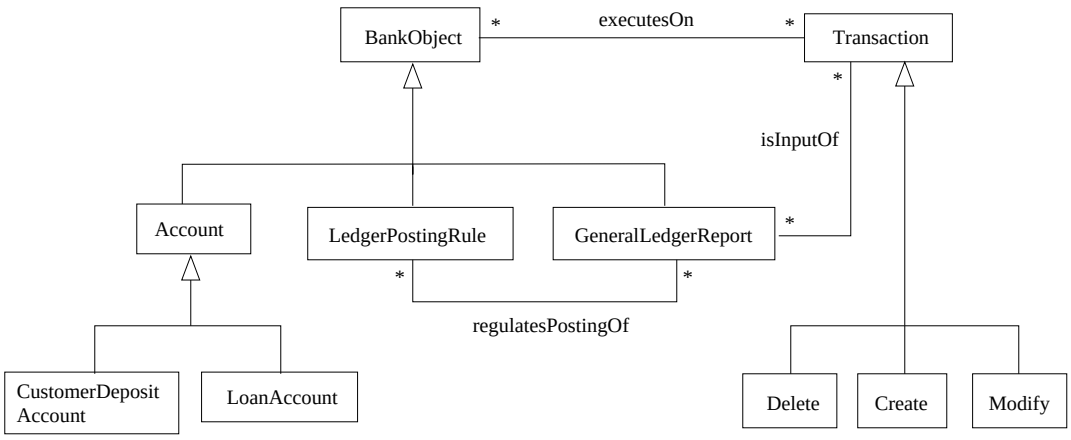



Figure 6.5: A Banking System Primary Model

Access control policies are not specified in the primary model. RBAC features can be incorporated into the primary model by composing an instance of the RBAC template in Fig. 6.2 with the primary model. The composition is carried out as follows:

1. *Instantiating the RBAC template*: To incorporate RBAC features into a primary model, the modeler must first instantiate the RBAC template by binding parameters to model elements that represent concepts in the domain of the primary model. Some of elements in the RBAC template may be elements in the primary model. Class diagrams obtained from the RBAC template are referred to as *context-specific RBAC instance*. Fig. 6.6 shows a context-specific RBAC instance for the banking application.

   In the diagram *BankRole*, *BankObject*, and *Transaction* are bound to *Role*,

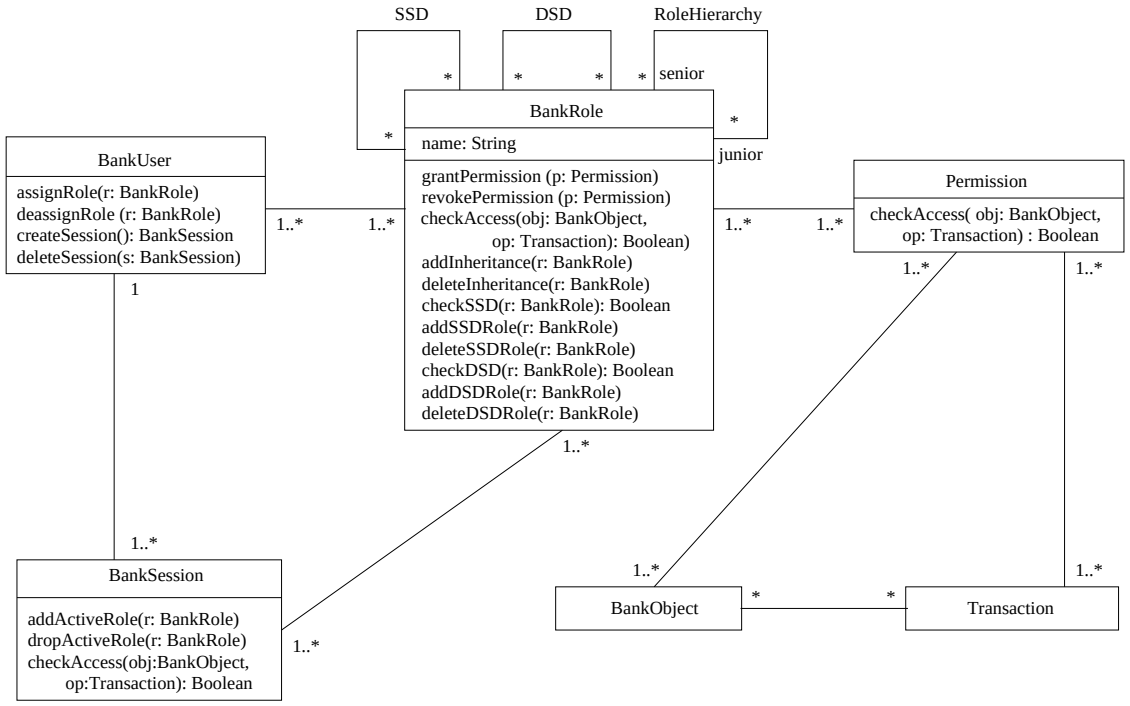



Figure 6.6: A Context-Specific RBAC Class Diagram

*Object*, and *Operation* in the RBAC template.

2. *Merging the context-specific instance with the primary model*: The view defined by the context-specific RBAC instance is merged with the view defined in the primary model to obtain a composed model. Elements in the instance and the primary model are merged if and only if they have the same syntactic type (i.e., UML metamodel class) and name. Model elements in the context-specific RBAC instance that do not exist in the primary model are added to the primary model.

The result of the composition is a composed model in which access control features specified by the context-specific RBAC instance are incorporated into the primary model. The composed model for the banking system is shown in Fig. 6.7. In the diagram, *BankObject* and *Transaction* in the context-specific RBAC instance

were merged with *BankObject* and *Transaction* in the primary model, and *BankUser*, *BankRole*, *BankSession*, and *Permission* classifiers were added to the primary model.

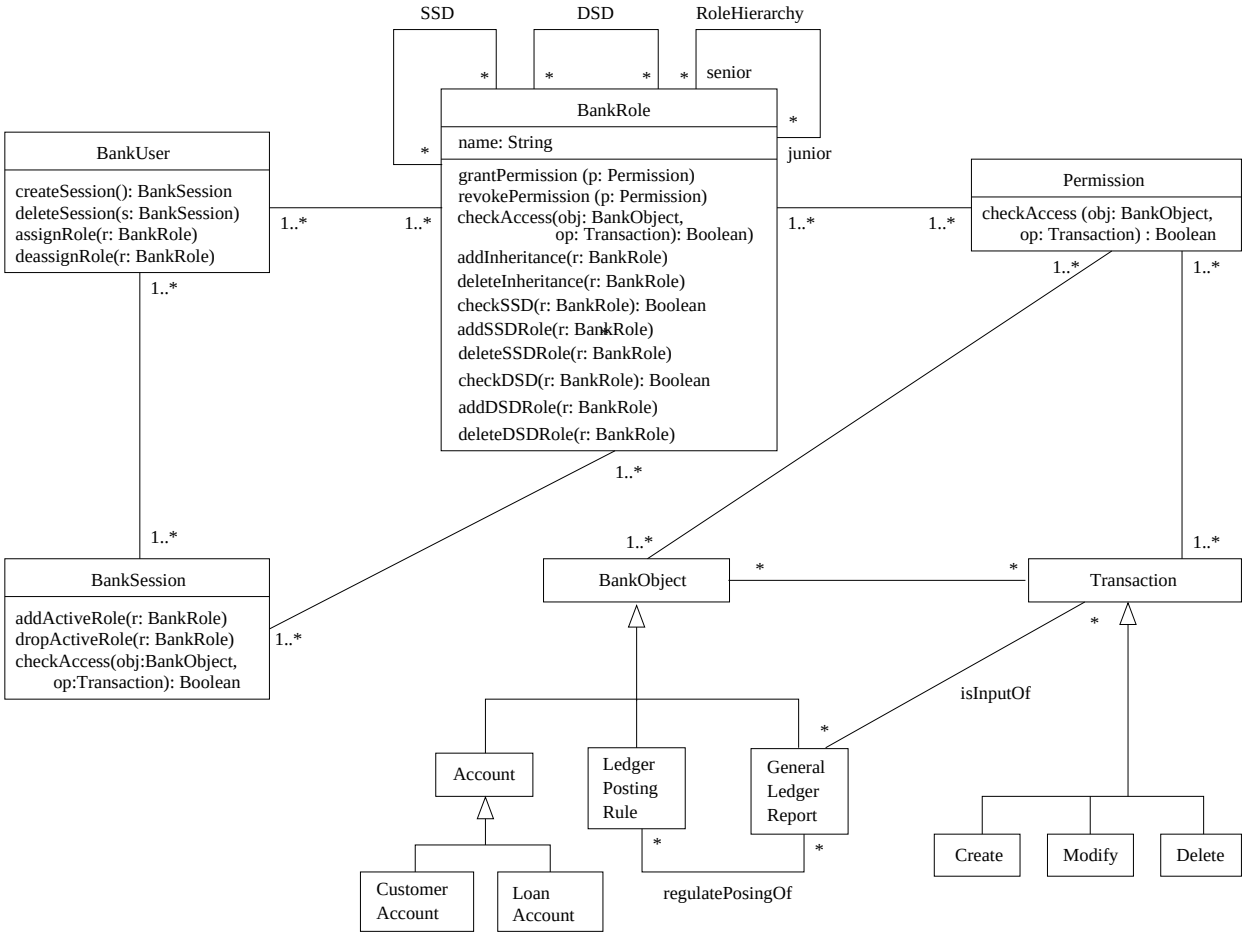



Figure 6.7: Composed Model

The RBAC banking model supports four types of policies: 1) *core policies* that conform to core RBAC, that is, policies that determine user-role and role-permission assignments, 2) *hierarchical policies* that conform to hierarchical RBAC, that is, policies that determine inheritance relationships between roles, 3) *SSD policies* that conform to SSD RBAC, that is, policies that determine what roles are conflicting, and 4) *DSD policies* that conform to DSD RBAC, that is, policies that determine what roles to be activated in a session.

## 6.4 Related Work

A large volume of research exists in the area of specification of access control policies. Formal logic-based approaches [9, 10, 13, 20, 45, 51, 75] are often used to specify security policies. They assume a strong mathematical background which makes them difficult to use and understand. Other researchers have used high-level languages to specify policies [48, 49, 74, 86]. Although high-level languages are easier to understand than formal logic-based approaches, they are not analyzable.

Researchers [68, 53] have also investigated extending UML for representing access control. Lodderstedt *et al.* [68] propose SecureUML and define a vocabulary for annotating UML-based models with information relevant to access control. Jürgens [53] models security mechanisms based on the multi-level classification of data in a system using an extended form of the UML called UMLsec. The UML tag extension mechanism is used to denote sensitive data. Statechart diagrams are used to model the dynamic behavior of objects, and sequence diagrams are used to model protocols.

Several researchers have looked into integrating the mandatory access control and role-based access control models. Osborn [76] examines the interaction between RBAC and MAC, and discusses the possible structures of role graphs that do not violate the constraints imposed by MAC. In their approach when a subject is assigned to a role, the subject can perform all the privileges in the role. Nyanchama and Osborn [73] discuss the realization of MAC in role-based protection systems.

Phillips *et al.* [79] examine the unification of MAC and RBAC into a security model and enforcement framework for distributed applications. In their work, RBAC is extended to include MAC to ensure that the clearance of users playing roles meet or exceed classification of resources, services, and methods being utilized. A role is assigned a classification, and the authorized user must possess a classification greater than or equal to the role classification.

## 6.5 Lessons Learned

In this study, the potential of the RBML to specify cross-cutting functionality has been evaluated using templates. Two access control policies, Role-Based Access Control (RBAC) and Mandatory Access Control (MAC), are studied. The specifications of RBAC and MAC can be used to systematically incorporate the policies into a model. A Hybrid Access Control (HAC) is proposed by merging the specifications of the RBAC and MAC. HAC can be used in the domains where both RBAC and MAC policies are needed such as the military domain.

The study has shown that RBML templates are suitable to specify the security aspects in a form that supports systematic composition (weaving) of the aspects with applications to incorporate properties of aspects. The template form is effective in instantiating policies through one-to-one binding between parameters and application elements. The instantiated model is then composed with a primary model.

When establishing the notion of RBML templates, it was found that the semantics of association-end roles in RBML specifications needs to be adapted in RBML templates. In RBML specifications, a multiplicity on an association-end role constraints the number of associated-ends that can be attached to a classifier. This is adapted in RBML templates that association-ends are parameterized or specified with a constant value. An instantiated association-end then has a value that is either the constant value or passed through the parameter.

The work described in this chapter focuses on specifying the static structure of the access control policies. A complete specification should also include descriptions of behavioral aspects of the policies. These aspects characterize allowed and prohibited behaviors by the policies. These can be specified in a template form of IPSs (see [62, 84] for examples).

The RBAC policies for a given application can be tested against a set scenarios, some of which can be obtained by instantiating the IPS templates of RBAC. For ex-

ample, in order to evaluate the impact of an RBAC policy on a system, test scenarios that model prohibited behaviors can be obtained by instantiating RBAC interaction patterns that describe prohibited behaviors. Such tests can be used to determine if the manner in which the policies are addressed in the design are sufficient to prevent unauthorized access.

A tool that allows developers to create and instantiate RBML templates has been developed (see Chapter 7). The composition process presented in this chapter is done manually. Subsequent work will focus on developing a tool that automates, in part, the composition process.

The RBML has been used in other aspect research groups at CSU and NASA/Ames Research Center. The group at CSU uses the RBML to define design aspects as crosscutting modeling concept [30, 36, 37, 50]. NASA uses the RBML to specify crosscutting requirements of a system [7, 104].

# Chapter 7

# RBML Tool Support

This chapter describes a prototype tool named RBML Pattern Instantiator (RBML-PI) that supports the RBML. RBML-PI allows one to generate UML models from RBML pattern specifications. Unlike other tools [28, 69, 77] where only uniform instantiations are allowed, RBML-PI allows structural variations of instantiations through variation points where user can specify application information.

A major benefit of the RBML is the utilization of UML modeling tools. Since the RBML is based on the UML, UML modeling tools can be used to build RBML specifications. In this work, IBM Rational Rose is used to build RBML specifications. RBML-PI takes RBML specifications as input to generate class diagrams and sequence diagrams. After instantiation, application developer completes the model by binding application-specific elements to the generated elements in the diagrams.

In this chapter, the CheckIn-CheckOut (CICO) domain-specific pattern presented in Chapter 5 is used to demonstrate the tool. The CICO pattern is used to generate design models of a library system and a vehicle rental system.

The rest of the chapter is organized as follows. Section 7.1 gives an overview of RBML-PI. Section 7.2 describes how RBML-PI can be used to generate models from the CICO pattern. Section 7.3 discusses related work. Section 7.4 concludes the chapter.

## 7.1 RBML Pattern Instantiator

RBML-PI is built on top of Rational Rose as an add-in component developed in C++. Several UML modeling tools (IBM Rational Rose, Together, ArgoUML, Poseidon, Microsoft Visio) were considered as a base tool of RBML-PI. Rose was chosen for the following reasons:

- Full support for sequence diagrams is needed. Other tools do not support sequence diagrams or are lack of features. For example, support for redirection of message source and target is not allowed or very limited.

- RBML-PI was developed in collaboration with NASA/Ames Research Center for the Air Traffic Control System project. They developed a tool based on Rose that generates statemachines from scenarios (i.e., sequence diagrams). Rose was chosen for the support of code generation from statemachines (using a tool such as RoseRT) which many UML tools do not support. Code generation from statemachines is useful for execution of generated state machines. In this research, Rose was chosen to be compliant with their tool, so that the two can work together. RBML-PI can generate sequence diagrams that conform to a pattern specification, and the generated sequence diagrams can be fed to a statemachine generator to obtain statemachines.

- Rose provides a good API called Rose Extensibility Interface that allows one to access the UML metamodel for manipulation of UML models.

One of the limitations of using Rose is that add-ins must be COM-compliant object which limits one to MS/Windows. Alternatively, XMI-based approaches can be investigated to overcome the limitation.

The development of the tool benefited from the UML syntax used as a based in the RBML. First, drawing features provided by Rose can be used to build RBML

specifications. Second, processing RBML specifications can be supported by the UML modeling tool through the provided API, because RBML specifications can be treated as UML models. Third, generated models can be manipulated in Rose like other UML models built by Rose. For example, changing a class or an operation in an instantiated class diagram requires corresponding changes in related sequence diagrams. Such changes can be supported by Rose.

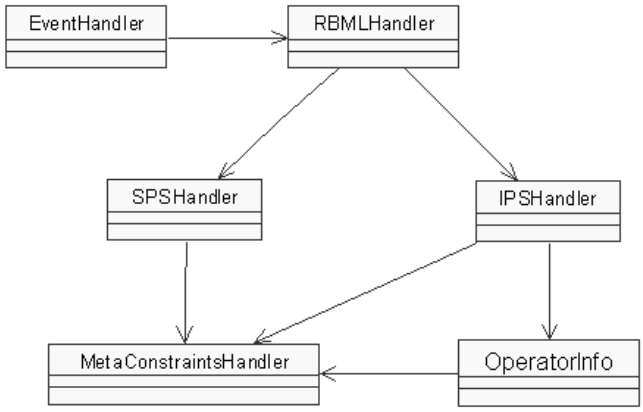



Figure 7.1: RBML-PI Class Diagram

The core structure of RBML-PI is shown in Fig. 7.1. It consists of *EventHandler*, *RBMLHandler*, *SPSHandler*, *IPSHandler*, *MetaConstraintsHandler*, and *OperationInfo*. *EventHandler* intercepts the event of clicking RBML menu and initializes RBML-PI. *RBMLHandler* initiates calls to the *SPSHandler* and *IPSHandler* in sequence (because IPSs are dependent on an SPS) to generate a class diagram and sequence diagrams. *SPSHandler* and *IPSHandler* read metamodel-level constraints specified in the pattern specification from *MetaConstraintHandler*. *OperatorInfo* supports UML sequence diagram operators (e.g., repeat, alt) defined in the UML 2.0. For example, if a *repeat* operator is defined in an IPS, RBML-PI instantiates the messages in the repeat box until the condition defined in the repeat is met. Currently RBML-PI supports only *repeat* operator.

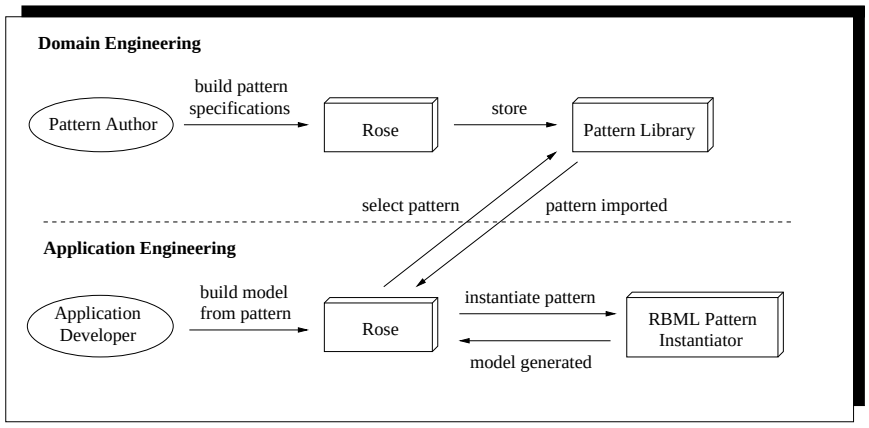



Figure 7.2: Overview of Tool Use

Fig. 7.2 shows how RBML-PI can be used. A pattern author uses Rose to build RBML pattern specifications and put them into a pattern library, which is a folder that contains pattern specifications. An application developer chooses a pattern to use from the library and load it into Rose. After loading, the metamodel-level constraints (e.g., realization multiplicities) specified in the specification may be further constrained for the application requirements. The pattern specification is instantiated using RBML-PI to generate UML models by binding the generated model elements to the application concepts of the system. Application-specific elements that are not part of the pattern specification may be added to complete the model. Pattern properties that are not specified, but found to be necessary during the application of the pattern are fed to pattern author for adjustment.

## 7.2 Instantiating RBML Specifications

This section demonstrates how RBML-PI can be used to develop models of a library system and a vehicle rental system from the CICO pattern presented in Chapter 5.

### 7.2.1 CICO Pattern Specification

The CICO pattern specification is built in a package using Rose that contains an SPS and IPSs of the CICO pattern. The package is then exported into a pattern library. Fig. 7.3 shows the SPS of the CICO pattern. The SPS consists of roles that define domain concepts; registered user (*User*), collection of registered users (*CollectionUser*), item check out details (*Lending*), item (*Item*), item description (*Description*), and checkin/checkout manager (*Controller*).

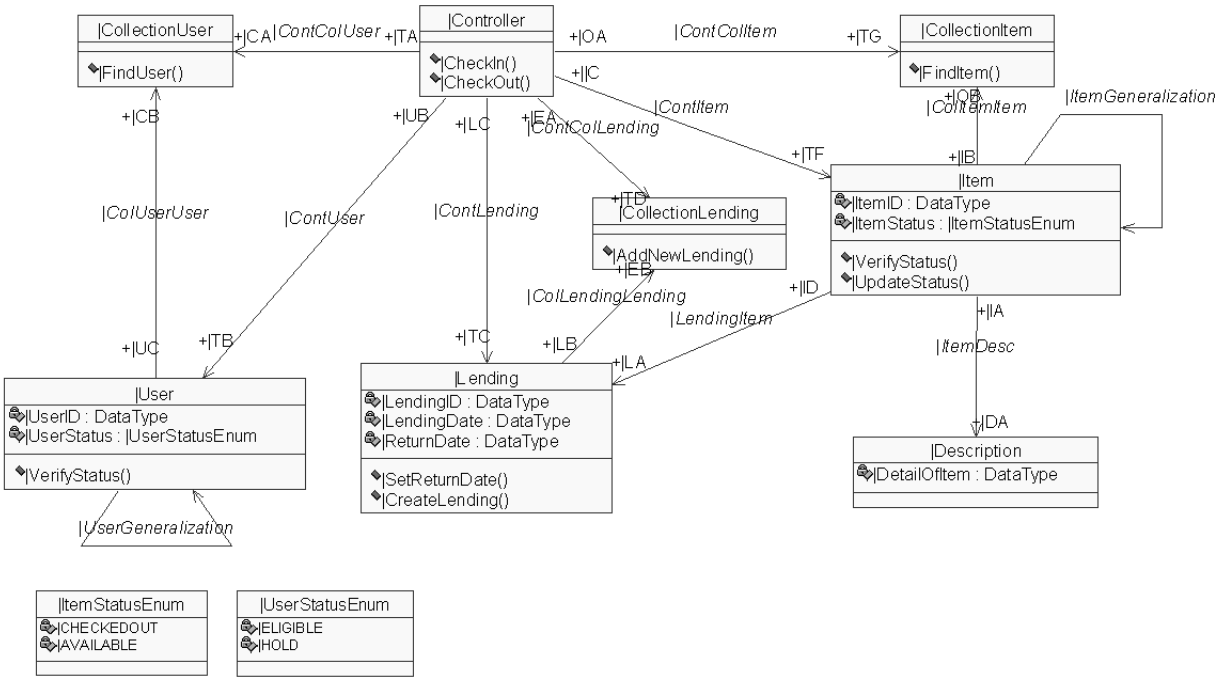
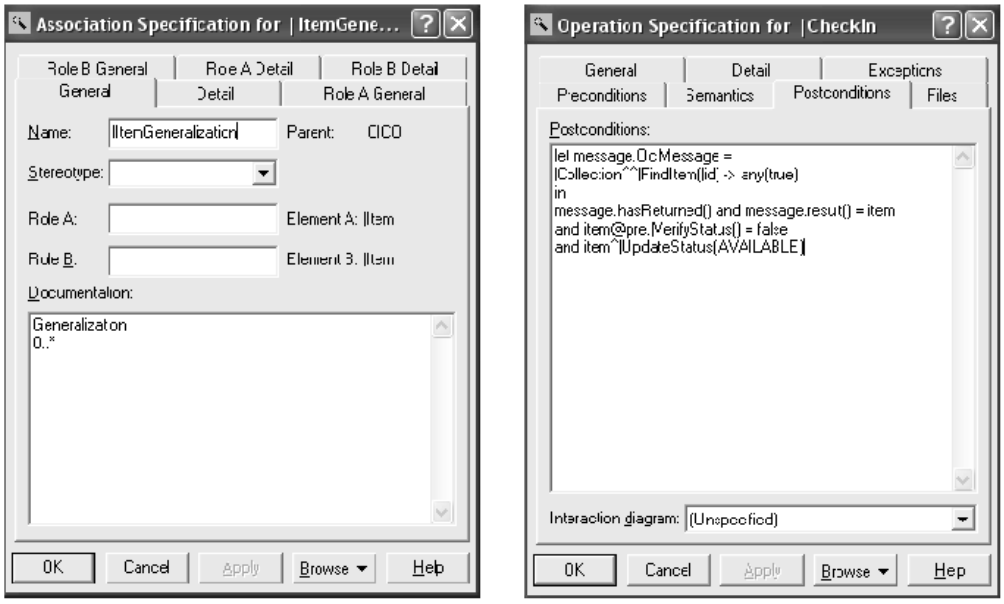



Figure 7.3: CICO SPS

Fig. 7.4(a) shows examples of the metamodel-level constraints for the *ItemGeneralization* relationship role. They describe that the base metaclass of the *ItemGeneralization* role is *Generalization* and there can be zero or more (*0..**) generalizations that play the role. These constraints are used to determine the generalization /specialization structure of items in an instantiated model.

Constraint templates in the RBML define model-level constraints. They can be instantiated to obtain application-specific model constraints during instantiation. Fig. 7.4(b) shows the postcondition of the *CheckIn* behavior in *Controller*. They are used to generate post conditions for an CheckIn operation. Currently RBML-PI provides limited support for instantiation of constraint templates that it simply copies constraint templates into an instantiated model.




(a) Metamodel-Level Constraints

(b) Constraint Template

Figure 7.4: CICO Constraints

Fig. 7.5 shows an IPS for a check-in scenario. It describes that the item is found in the item collection, and its status is checked to determine whether it was checked out before. If the item was checked out, the return date is set and the status of the item is updated to available.

Fig. 7.6 shows an IPS for a check-out scenario. An instance of a *Controller* classifier invokes *FindUser* with the user ID *uid* to obtain a matching user "*u*" from a collection of users. The status of the user is verified. If the user is allowed to check out

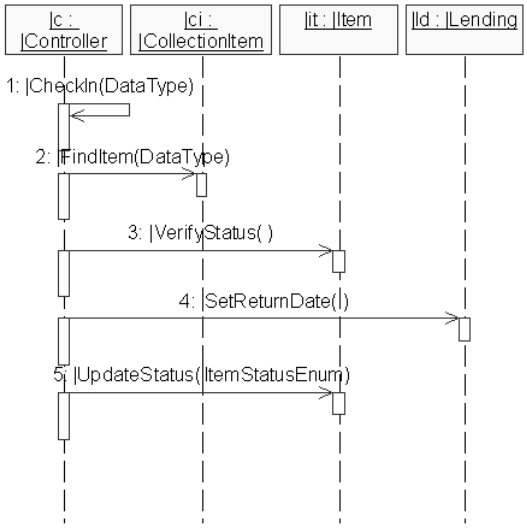



Figure 7.5: CICO CheckIn IPS

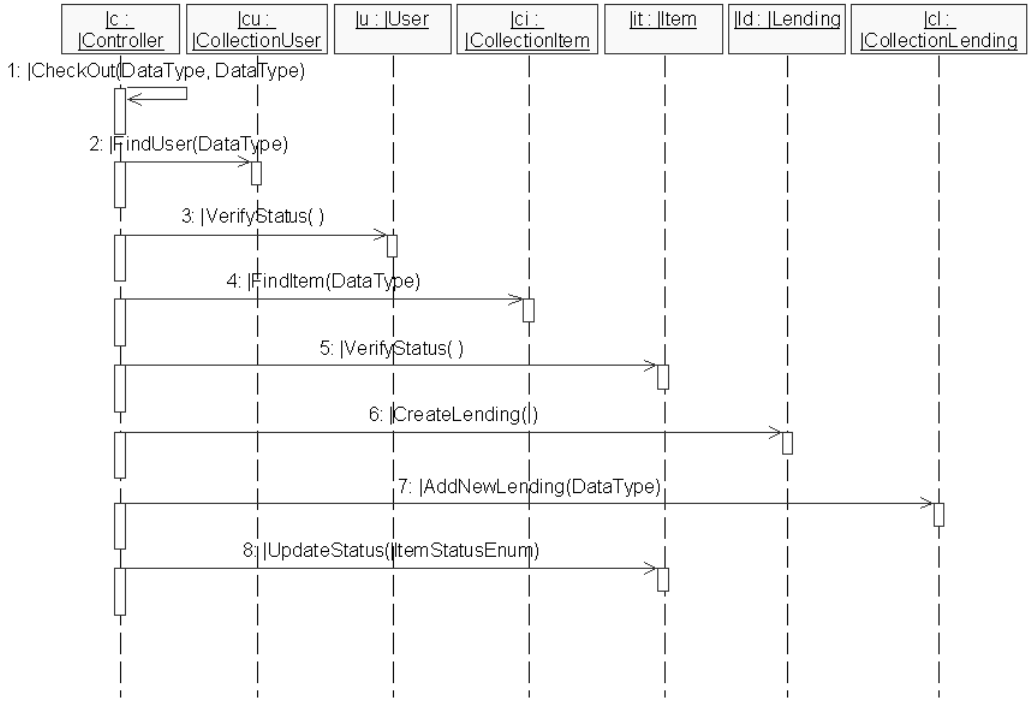



Figure 7.6: CICO CheckOut IPS

the item, the requested item is retrieved. The status of the item is checked to determine if the item can be checked out. If the item can be checked out, a lending record is created. The record is then added to a collection of lendings (*CollectionLending*), and the status of the item is updated to unavailable.

### 7.2.2 CICO Pattern Instantiation

An application developers import the CICO pattern package from the pattern library into the current project for instantiation. In Rose, the directory structure of the project is shown on the left-hand side where the CICO package is imported under the Logical View folder as shown in Fig. 7.7.

The instantiation process includes 1) generating an instantiated model using RBML-PI, 2) renaming of generated model elements specific to the application, and 3) adding additional application-specific model elements to complete the model.

When generating an instantiated model, RBML-PI takes the metamodel-level constraints specified in the specification to determine the structure of the model being generated. For example, RBML-PI uses the lower bound of the multiplicity in Fig. 7.4(a) to determine the number of generalizations to generate. In this example, no generalizations are generated since the lower bound is set to zero. Upper bounds are not used in instantiation, but they are used in checking whether the complete model still conforms to the pattern specification (conformance checking has not been implemented yet). For example, if the *Item* in Fig. 7.3 has a multiplicity of *1..3*, the complete model cannot have more than three *Item* classes.

Realization multiplicities can be further constrained by application developer during instantiation, but cannot be weakened. If multiplicities are weakened, then pattern conformance is broken. For example, if a library system requires at least two types (*Book* and *Multimedia*) of *Copy*, then the application developer may further restrict the multiplicity *0..** of *ItemGeneralization* to *2..** as shown in Fig. 7.7. There

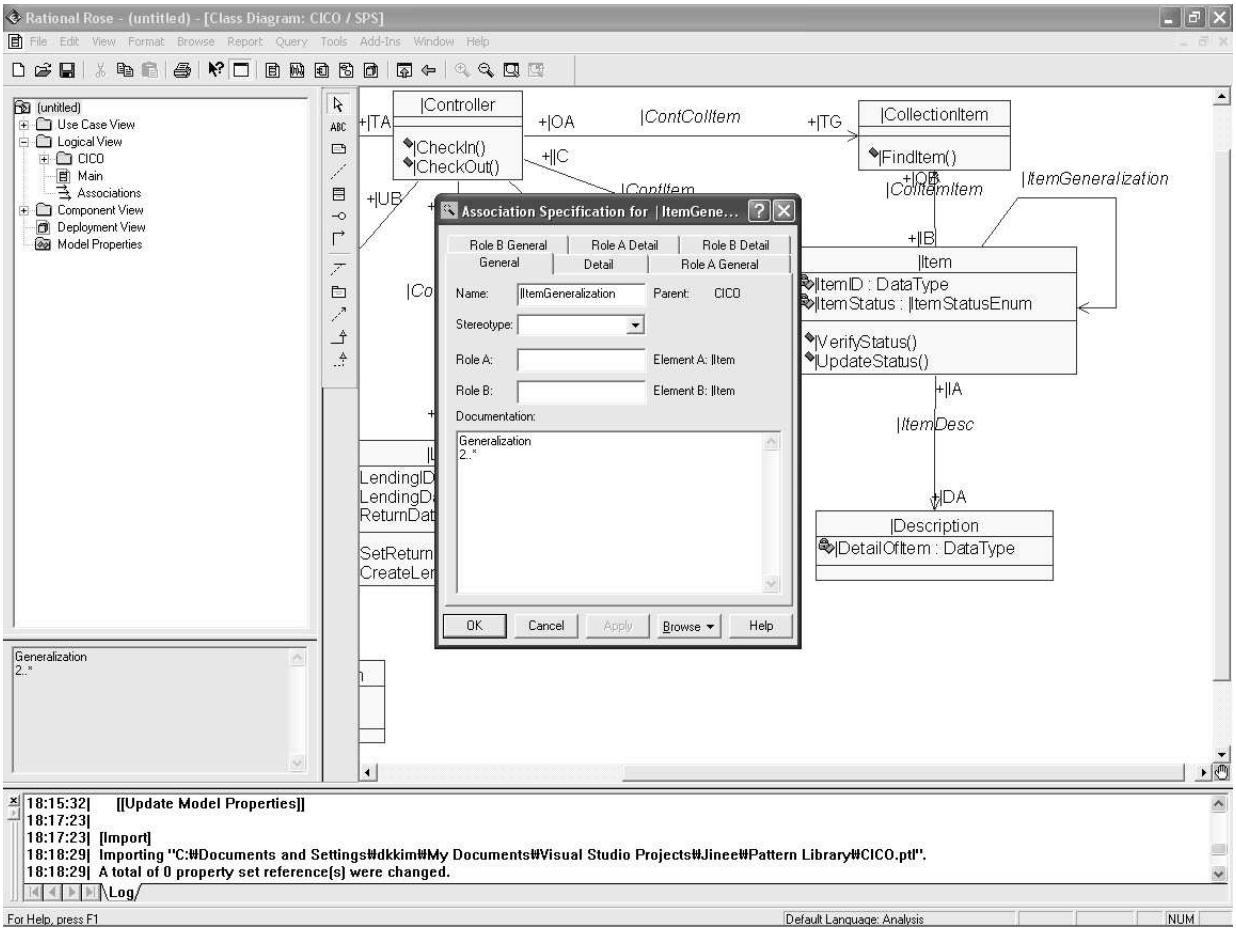



Figure 7.7: Further Restiction of Pattern Property

might be situations where generated model constructs are not needed. However, it is recommended not to remove them for the sake of conformance.

Fig. 7.8 shows a class diagram generated from the CICO SPS in Fig. 7.3. The following are some of notable points to the diagram:

- The class *DescriptionG0* contains four instantiated attributes resulted from the multiplicity *4..** (not shown) defined in the *DetailOfItem* structural feature role in *Description* in Fig. 7.3. The multiplicity is further restricted from the original multiplicity *1..**.

- The class *ControllerG0* contains two instantiated operations *CheckOutG0* and

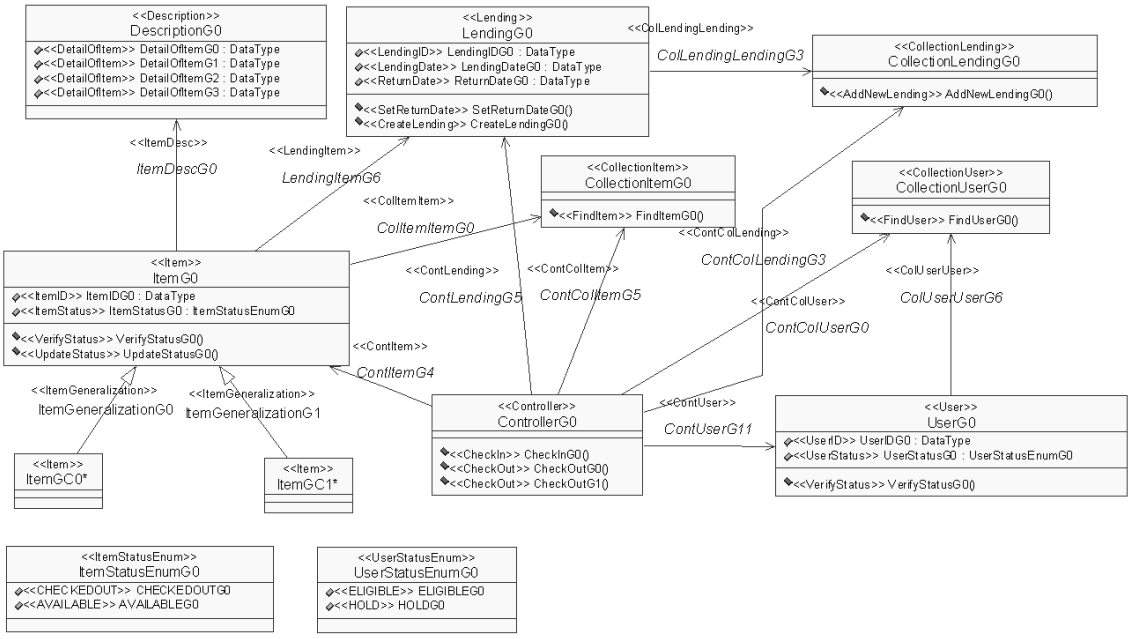



Figure 7.8: An Instantiated Class Diagram for a Library System

*CheckOutG1* resulted from the multiplicity *2..** (not shown) defined in the *CheckOut* role in Fig. 7.3. The multiplicity is further restricted from the original multiplicity *1..**.

- There are two specializations *ItemGC0* and *ItemGC1* created in the diagram as specified in Fig. 7.7.

- The stereotype on model elements denotes the role from which they are instantiated. It can be used to verify which model elements in the complete model map to which roles in the CICO SPS.

The sequence diagrams that are instantiated from the *CheckIn* and *CheckOut* IPSs are shown in Fig. 7.9 and Fig. 7.10, respectively. Each instantiated message in the diagrams has its own activation bar because the Rose API does not support hierarchical message sequence.

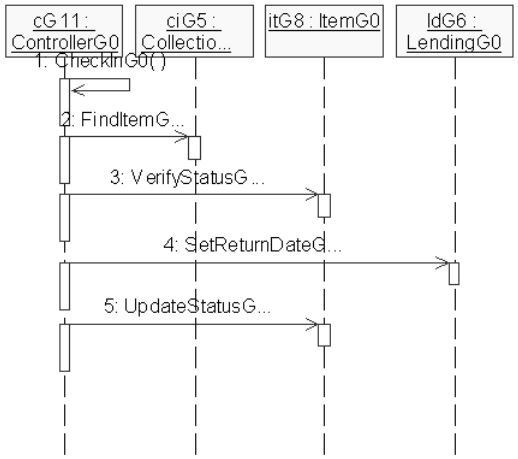



Figure 7.9: Instantiated CheckIn Sequence Diagrams for a Library System

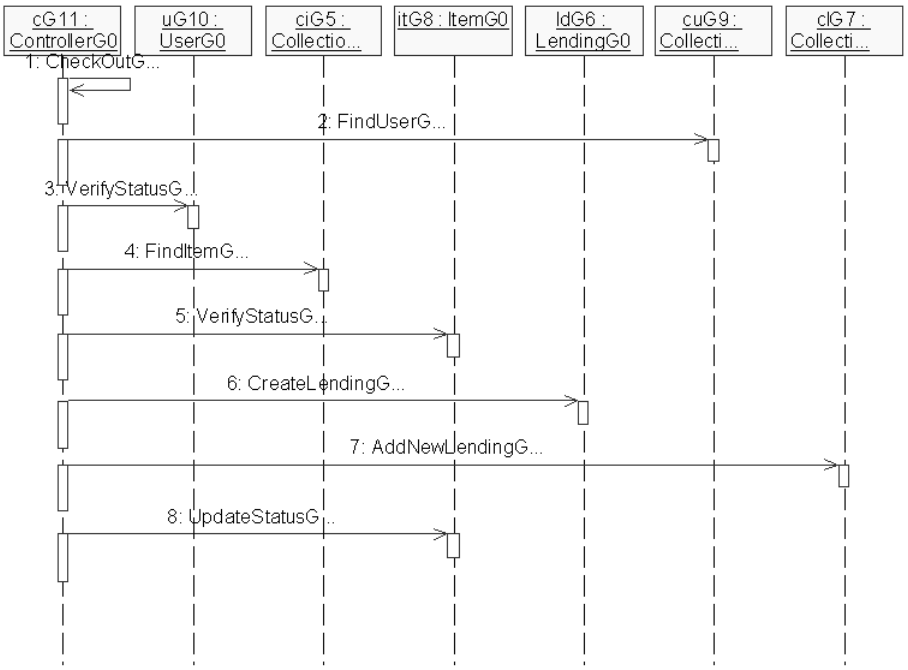



Figure 7.10: Instantiated CheckOut Sequence Diagrams for a Library System

The instantiation is not complete yet. The instantiated model elements have automatically generated names, which need to be bound to application elements. For example, in the completed library class diagram shown in Fig. 7.11, the instantiated

model *ItemG0* is bound to *Copy* in the library system, and its specializations *ItemGC0* and *ItemGC1* are bound to *Multimedia* and *Book*.

Application developer may add application-specific model elements to the instantiated class diagram. In Fig. 7.11, the *Reservation* class is added to provide reservation service, the *name* and *address* are added to maintain contact information of members. Other model elements that are not stereotyped are also application-specific. Fig. 7.12 and Fig. 7.13 show sequence diagrams for check-in and check-out scenarios for a library system.

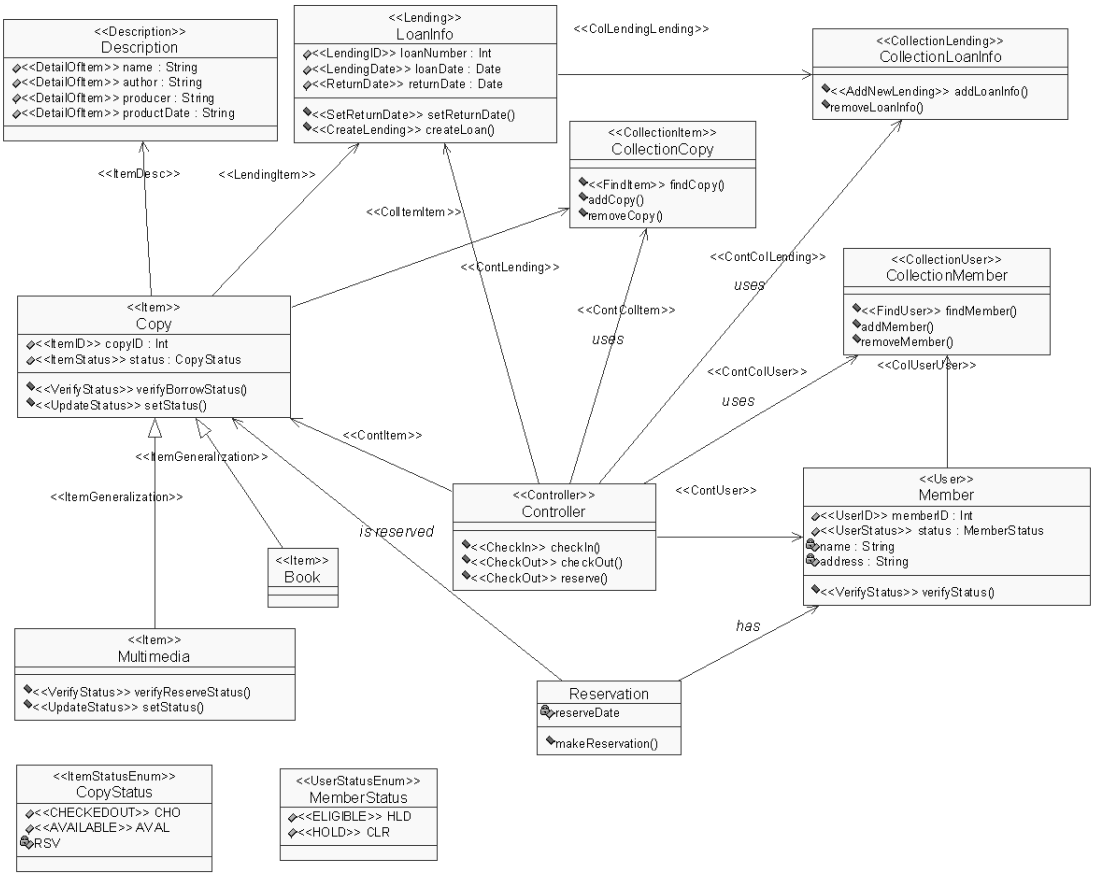



Figure 7.11: A Library Class Diagram

Another example of a vehicle rental system created by RBML-PI is given in Ap-

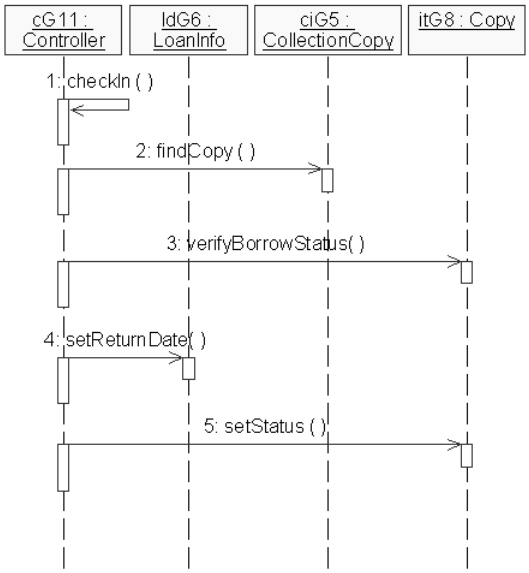



Figure 7.12: A Library CheckIn Sequence Diagram

pendix C.

## 7.3 Related Work

Florijn *et al.* [28] demonstrate a tool developed in Smalltalk that supports generating program elements from a pattern template by instantiation, and integrating the instance with the existing program by binding program elements to roles in the pattern, and checking whether the resulting instance meets the properties of the patterns. Structure of patterns is represented by a tree-like graph where a node represents a program elements (e.g., class, association) which is associated with roles that contain references pointing to program elements. Pattern descriptions are purely structural, and few attention has been has been paid to behavioral aspects of patterns. Their technique is based on non-standard notation, and thus tools that support the notation have to be developed.

Eden [24] proposes LePUS, a formal language to define patterns in terms of pro-

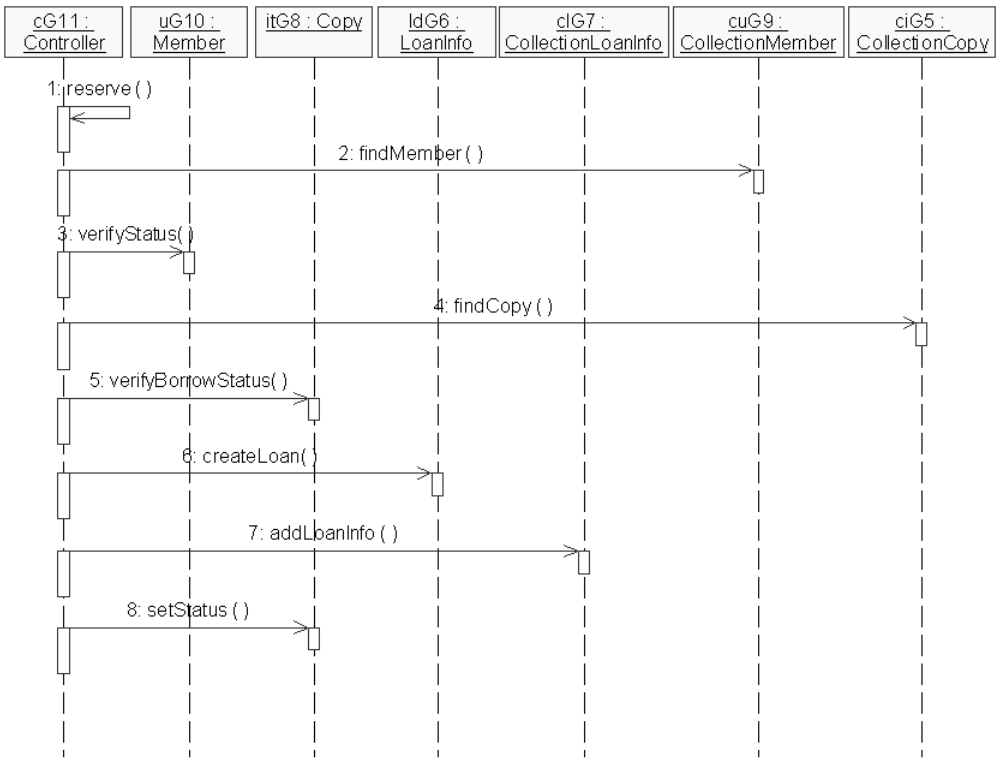



Figure 7.13: A Library CheckOut Sequence Diagram

gram properties, and demonstrates a supporting tool developed in PROLOG that applies patterns to programs and finds patterns. A program is said to implement a pattern if the program conforms to the pattern constraints. LePUS focuses on structural properties of patterns and its application to programs rather than designs, which restricts to a particular programming language, Eiffel.

Pagel and Winter [77] present a metamodel for describing patterns using Object Modeling Technique (OMT) [89], and a tool that implements the metamodel and a pattern repository. Their approach lies along the same line as ours in that they focus on the use of patterns to support the development of designs. Their pattern descriptions are templates which are similar to RMBL templates. However, their tool does not fully support instantiation of the templates.

Mapelsden *et al.* [69] propose the DPML, a visual modeling language, that provides a set of constructs (e.g., interface, method) to specify design patterns, and demonstrate a tool that allows to build UML design models and pattern specifications which is described in non-standard notation and to instantiate them. A pattern instance is a part of a UML object model that is instantiated from a UML design model. Participants in a pattern instance are bound to objects in the object model. In their approach patterns are defined at design level, and thus bindings are done at object level. Like Florijn *et al.* using non-standard notation, the DPML requires exclusive tool support for its notation. No mechanisms are described as to how to specify behavioral aspects of patterns.

## 7.4 Lessons Learned

This chapter has evaluated the potential of the RBML to support the development of tools that enables systematic use of patterns by developing a prototype tool, RBML-Pattern Instantiator (RBML-PI), that generates UML models from RBML specifications.

The development of the tool benefited from the UML syntax used as a base in the RBML. First, drawing features provided by Rose are used to build RBML specifications. Second, processing RBML specifications is fully supported by the UML modeling tool through the provided API since RBML specifications can be treated as UML models because they use the UML syntax. Third, generated models can be manipulated in Rose as other UML models built in Rose. For example, changing a class or an operation in an instantiated class diagram requires corresponding changes in related sequence diagrams. This can be supported by Rose.

This chapter has demonstrated the use of RBML-PI to generate UML class diagrams and sequence diagrams of a library system and a vehicle rental system from the RBML specification of the CheckIn-CheckOut (CICO) pattern. The same ex-

amples were used in Chapter 5 where the models are built manually. It was very time-consuming and tedious. Even a small change like changing a multiplicity for a classifier role from (1..*) to (2..*) was painstaking to find a room for the new classifier in the instantiated diagrams. Technically RBML-PI can generate models with any number of model elements using multiplicity constraints where one can specify the number of instantiations. However, it has not been tried with more than ten instantiations for a single classifier role because of the clutter that would have been created in the diagram which makes the diagram unreadable. In collaboration with NASA, we used RBML-PI to generate a model of a weather forecasting system using the Observer design pattern.

There is a lot of work left to be done. The following describes limitations of RBML-PI that have to be removed in the future versions:

- It does not fully support metamodel-level constraints and constraint templates expressed in the Object Constraint Language (OCL) [103]. For full support, tools that can edit and parse OCL expressions are needed. OCL tools that can be integrated with Rose are currently investigated.

- Currently RBML-PI only supports the hierarchies that have either generalization role or realization role, but not both.

- Currently RBML-PI only supports single-level of generalizations or realizations.

- RBML-PI does not check whether user modifications violate the pattern properties or not. Currently RBML-PI lets users remove instantiated model elements, which may violate pattern constraints.

- Currently RBML-PI only allows application of one pattern. In practice, there may be cases where more than one pattern need to be used to solve a problem.

- Currently RBML-PI only supports SPSs and IPSs. Support for SMPSs is needed.

RBML-PI is currently being used at NASA for evaluating systems behavior in requirement analysis. Subsequent work will focus on developing tools that can verify pattern conformance, and embed patterns properties into models (pattern-based model refactoring).

# Chapter 8

# Conclusion and Future Work

A pattern specification language, the Role-Based Metamodeling Language (RBML), has been developed to achieve the goal of developing a rigorous and practical pattern specification technique that supports the development of tools that enable systematic use of patterns.

Rigor and practicality of the RBML has been demonstrated by using it to specify 1) solutions of popular design patterns, 2) an application-domain pattern, the CheckIn-CheckOut pattern, and 3) access control aspects: Mandatory Access Control (MAC), Role-Based Access Control (RBAC), and Hybrid Access Control (HAC).

The potential of the RBML to support the development of tools that enables systematic use of patterns has been shown by successfully developing a prototype tool, RBML-Pattern Instantiator (RBML-PI) that generates UML models from RBML specifications.

There is much work left to be done. SMPSs need to be further developed to expand their applicability. The work presented in this dissertation only considers state, trigger, and transition roles. The notion of activity roles needs to be defined to support specifying actions, for example, states with actions or triggers with actions. Notation and conformance rules for composite state roles need to be developed. For example, composite state roles can be used for patterns in the domain of component-based systems to capture state hierarchies [46]. Checking consistency between SMPSs

and IPSs also needs to be investigated. For example, a trigger role and an activity role in an SMPS can respectively refer to an incoming message role on a lifeline role and an outgoing message role in an IPS. Consistency checking involves ensuring that the referred roles exist in an IPS.

There is increasing interest in generating state machines from scenarios described by sequence diagrams for analysis of system behavior [93, 7, 104]. Along the same line, SMPSs can be generated from IPSs.

Unlike the descriptions of pattern solutions that have enough information to be codified, pattern problems are often described by just one or two motivating examples (see [16, 35, 80, 92]). Work is planned to investigate how the RBML can be used to specify pattern problems. This becomes especially important in pattern-based model refactoring [31]. A specification of a problem space of a pattern can be used to check whether a pattern is applicable to a design model (i.e., the model is a member of the set of problem models characterized by the problem specification).

In this research, we have focused on functional properties (structural and behavioral properties) of patterns. However, a complete pattern specification with respect to pattern properties should also include specification of non-functional properties such as reliability, performance, security, and resource usage. For example, a stochastically timed extension of UML statechart diagrams that allows one to constrain time values can be used to analyze performance of distributed systems [39]. Extending this work to cover non-functional properties is an area of further work.

The RBML can be used to support Component-based software development (CBSD) and UML profile definition. CBSD concerns the development of software systems from existing components based on a plug-in/plug-out paradigm. An important issue in CBSD is to find the components that meet the functional requirements of the system being built. The RBML can be used to specify the requirements of components [61]. A specification of component requirements characterizes a set of

components. The specification can be used to find a component in the set.

The RBML can also be used to define lightweight UML profiles by defining specializations of the UML metamodel. A lightweight profile defines an extended UML metamodel that does not add or remove UML metamodel elements or constraints, that is, it simply extends the features associated with existing metamodel elements. A problem with existing mechanisms used to define UML profiles is that they are defined in informal terms, and there is no support for specifying families of constraints (e.g., families of operation pre- and post-conditions).

# Appendix A

# Design Pattern Specifications

## A.1 The Visitor Pattern

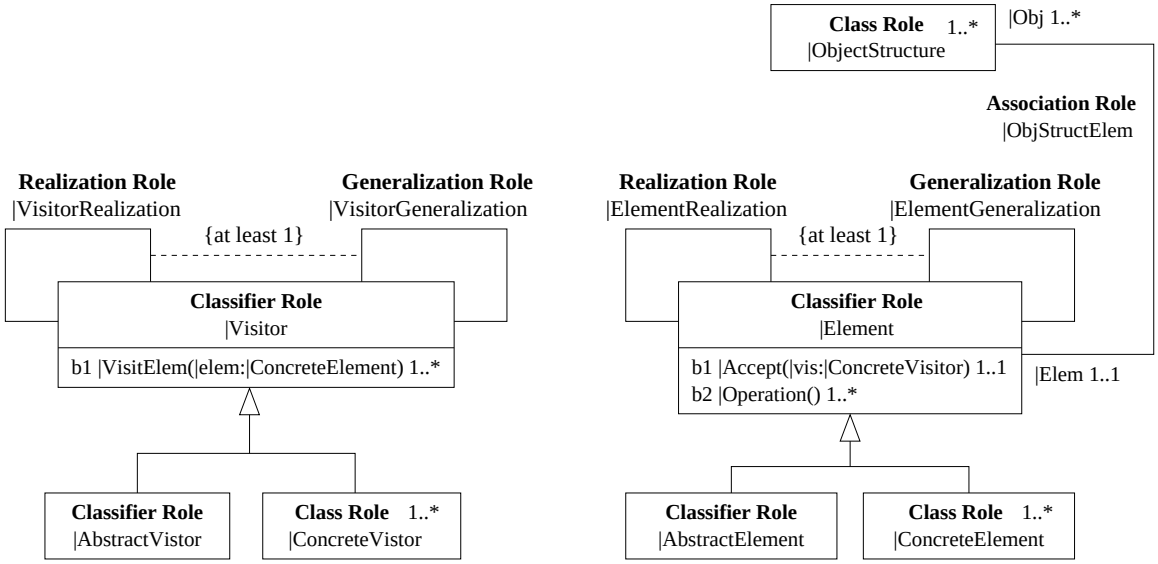



Figure A.1: A Visitor SPS

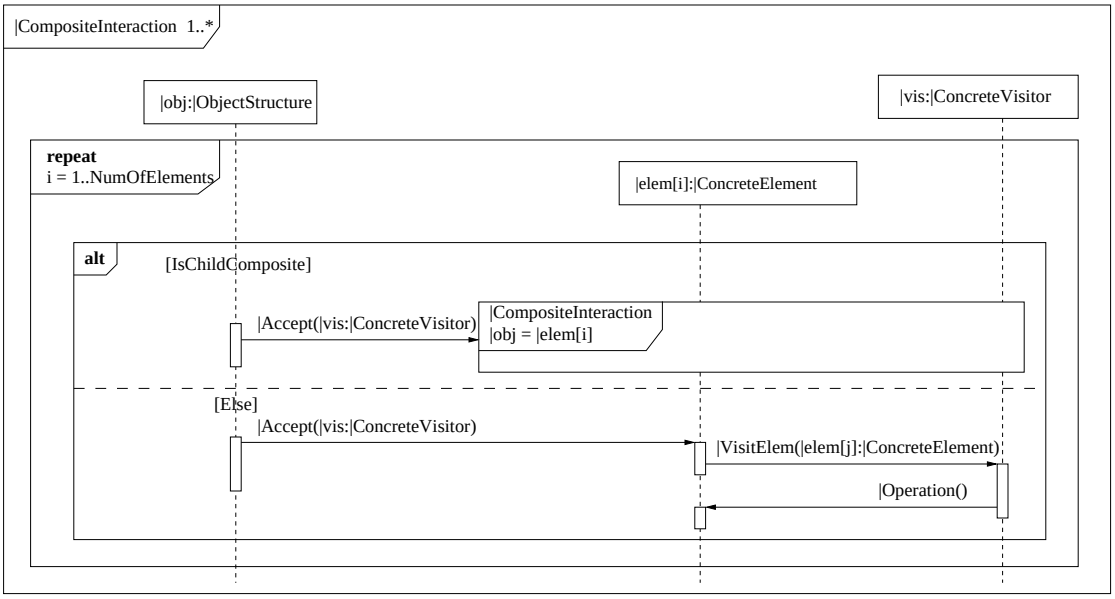



Figure A.2: A Visitor IPS

## A.2 The Abstract Factory Pattern

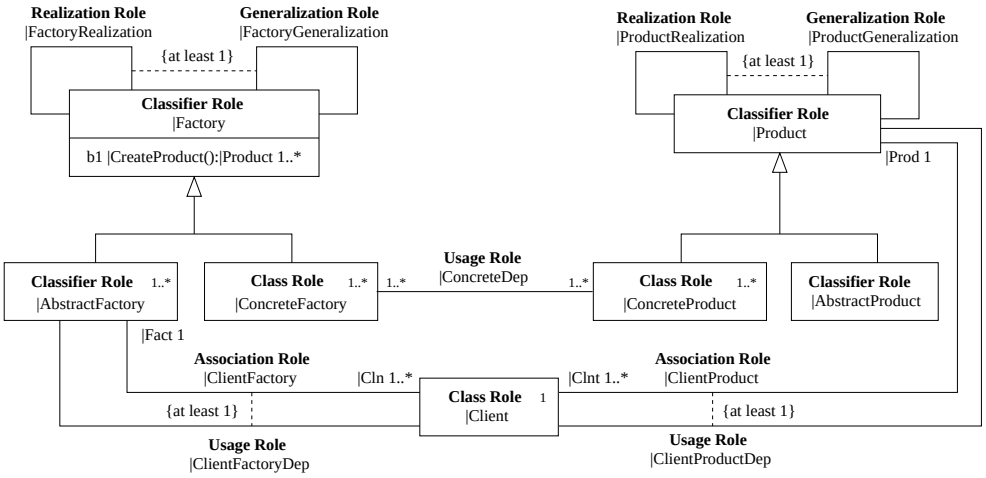



Figure A.3: An Abstract Factory SPS

## A.3 The Iterator Pattern

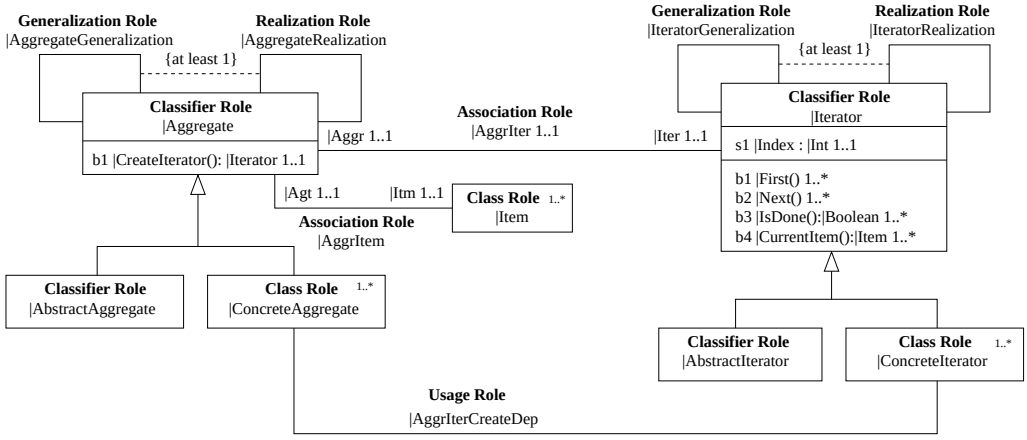



Figure A.4: An Iterator SPS

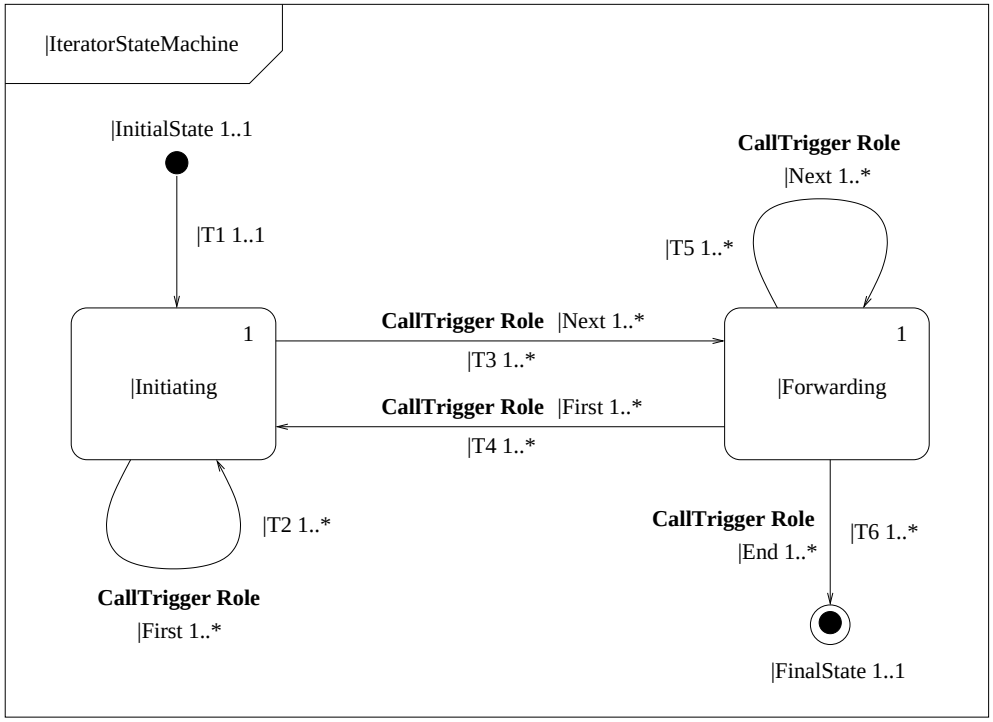



Figure A.5: An Iterator SMPS

## A.4 The Observer Pattern

The Observer design pattern provides a way to maintain consistency between related or dependent objects called *Observer* when one object called *Subject* changes its state. All the dependent observers are notified whenever a state change occurs to the subject. Each observer then queries the subject to synchronize its state with the subject's state. The Observer pattern reduces coupling among observers, which increases reusability.

### A.4.1 SPS

The class diagrams characterized by an Observer SPS have subject and observer classifiers that are abstract or concrete where a subject has associations with observers. A subject class has state attributes and operations for attaching and detaching observers, notifying to observers, and getting and setting a state. An observer class has state attributes for synchronization with the subject's attributes and operations for updating the attributes.

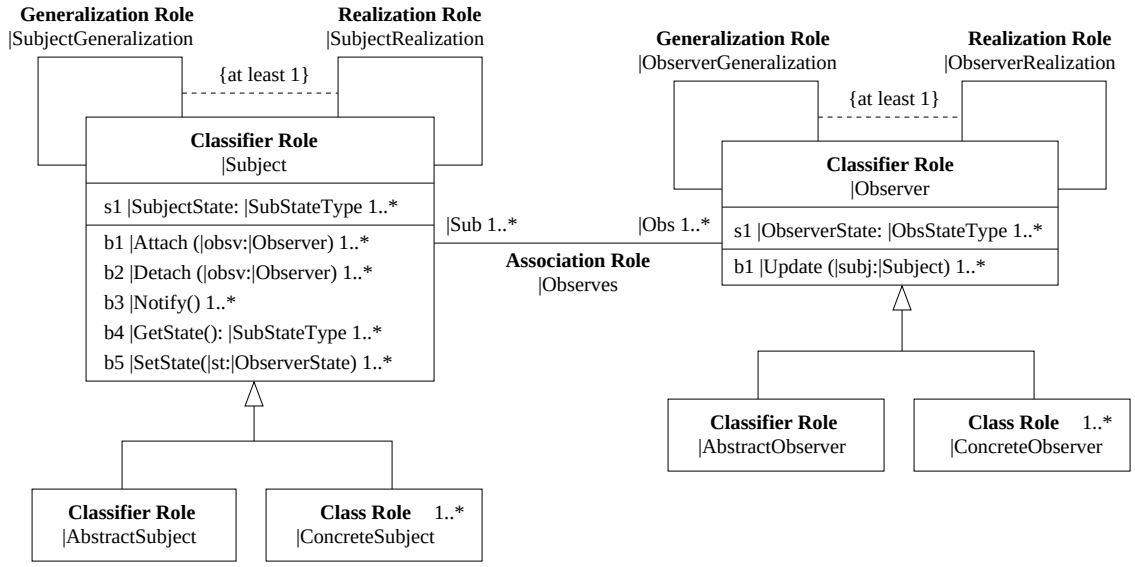



Figure A.6: An Observer SPS

An SPS for the Observer pattern is shown in Fig. A.7. The SPS is an extension

of the simplified Observer SPS in Section 3.2.1.5. The extension includes *Detach*, *GetState*, and *Notify* behaviors in *Subject* in addition to the behaviors defined in the *Subject* in the simplified Observer SPS. *Subject* hierarchy specifies that if there is an abstract class that plays *AbstractSubject*, there must be at least one relationship playing either *SubjectRealization* or *SubjectGeneralization*. A similar interpretation is applied to *Observer* hierarchy.

#### A.4.1.1 Well-formedness Rules

The metamodel-level constraints defined on the Observer SPS are given below:

- A classifier that conforms to *AbstractSubject* must be an interface or an abstract class:

  **context** |AbstractSubject **inv**:

  self.oclIsTypeOf(Interface) or

  (self.oclIsTypeOf(Class) and self.isAbstract = true)

  A similar constraint is defined for *AbstractObserver*.

- A classifier that conforms to *ConcreteSubject* must be a concrete class:

  **context** |ConcreteSubject **inv**: self.isAbstract = false

  A similar constraint is defined for *ConcreteObserver*.

- A relationship that conforms to *SubjectRealization* must have an interface or a type at its supplier end and a concrete class at its client end:

  **context** |SubjectRealization **inv**:

  (self.supplier.oclIsTypeOf(Interface) or

  (self.supplier.oclIsTypeOf(Class) and self.supplier.isAbstract = true) and

  self.client.oclIsTypeOf(Class)

  A similar constraint is defined for *ObserverRealization*.

- A relationship that conforms to *SubjectGeneralization* must have its parent and child to be the same type:

  **context** |SubjectGeneralization **inv**:

  self.parent.evaluationType() = self.child.evaluationType()

  A similar constraint is defined for *ObserverGeneralization*.

- An association-end that conforms to *Sub* must have a multiplicity of 0..*:

  **context** |Sub **inv**: self.lowerBound() = 0 and self.upperBound() = *

- An association-end that conforms to *Obs* must have a multiplicity of 0..*:

  **context** |Obs **inv**: self.lowerBound() = 0 and self.upperBound() = *

#### A.4.1.2 Constraint Templates

The constraint templates for the behavioral feature roles are defined as follows:

- An *Attach* operation attaches an observer object to the subject object:

  **context** |Subject::|Attach(|obsv:|ConcreteObserver)

  **pre**: true

  **post**: self.|Obs = self.|Obs@pre → including(|obsv)

- A *Detach* operation removes an observer object from the subject object:

  **context** |Subject::|Detach(|obsv:|ConcreteObserver)

  **pre**: |ConcreteObserver → includes(|obsv)

  **post**: |ConcreteObserver@pre → excluding(|obsv)

- A *GetState* operation returns the current value of *SubjectState*:

  **context** |Subject::|GetState():|SubjStateType

  **pre**: true

  **post**: result=|SubjectState

- A **SetState** operation sets the subject state:

  **context** |Subject::|SetState(|newState:|SubStateType)

  **pre**: true

  **post**: |SubjectState = |newState

- An *Update* operation changes the value of *ObserverState* to the value obtained from *Subject*, and invokes a *GetState* operation call:

  **context** |Observer::|Update(|subj:|ConcreteSubject)

  **pre**: true

  **post**: **let** observerMessage: OclMessage =

  |ConcreteSubject^^|GetState() → notEmpty() **in**

  observerMessage.hasReturned() and message.result() = st

  |ObserverState = st

### A.4.2 IPS

Fig. 3.12(a) shows an IPS that describes the pattern of interactions between a subject and its observers initiated by the invocation of the subject's $Notify$ operation. The expression $|subj : |ConcreteSubject$ indicates that the lifeline role *subj* is played by an instance of a $ConcreteSubject$ class (i.e., a class that conforms to the $ConcreteSubject$ role defined in the Observer SPS). The lifeline role $obsv[i]$ is played by the $i^{th}$ observer in the set of observers attached to the subject playing the *subj* role. The **repeat** fragment in the IPS indicates that the enclosed interaction is repeated for each observer attached to the subject playing the *subj* role. $NumOfObservers$ is the number of observers attached to the subject. The **repeat** fragment is used to concisely represent parts of conforming interaction diagrams that have a common structure. The IPS describes the following interaction pattern:

- Invocation of a subject's *SetState* operation (i.e., an operation that conforms to

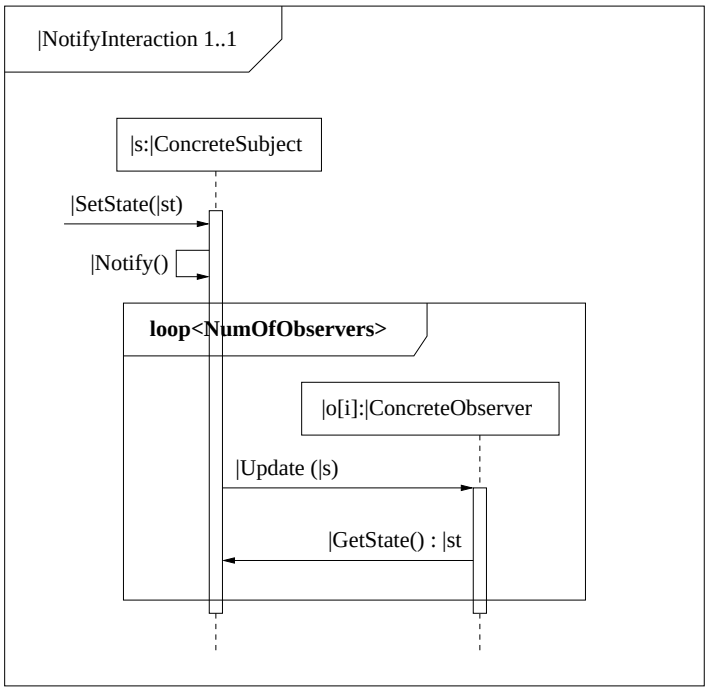



Figure A.7: An Observer IPS

the *SetState* feature role) results in a call to the *Notify* operation in the subject.

- The call to the subject's *Notify* operation results in calls to the *Update* operation in each observer linked to the subject.

- Each *Update* operation calls the *GetState* operation in the subject.

## A.5 The Bridge Pattern

The Bridge design pattern decouples an abstraction from its implementation called *Implementor* so that the two can vary independently. The abstraction and implementor are connected through a relationship called *Bridge*. An abstraction class can be specialized to define subclasses of abstractions. Implementor classes implement the operations defined in the abstraction.

### A.5.1 SPS

The class diagrams characterized by a Bridge SPS have abstraction and implementor classifiers that are abstract or concrete. An abstract (or generalized) abstraction is associated with an abstract (or generalized) implementor. Implementor classifiers have operations that implement the operations defined in the abstraction classifiers.

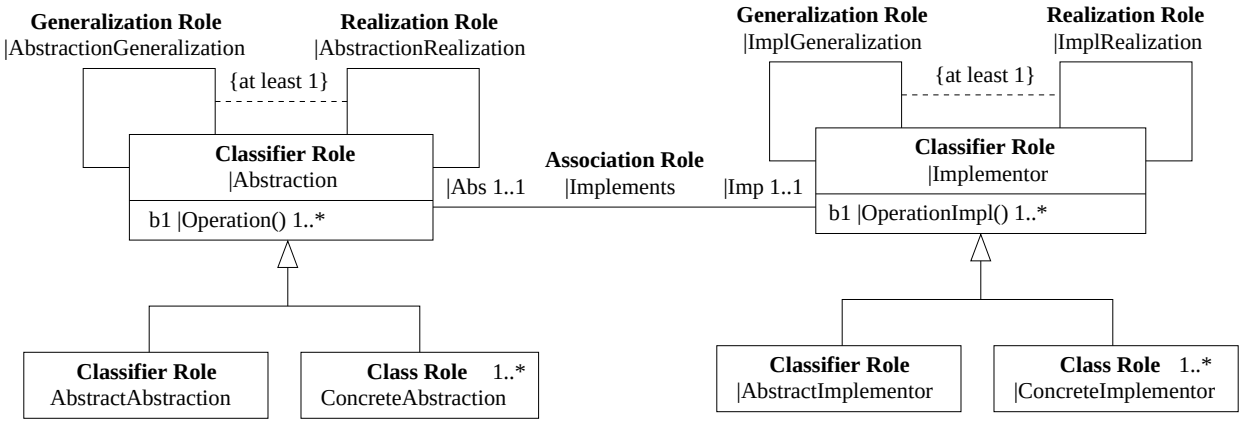



Figure A.8: A Bridge SPS

Fig. A.8 shows a Bridge pattern SPS that consists of hierarchies of *Abstraction* and *Implementation*. *Abstraction* hierarchy specifies the structure of abstraction classifiers where there can be zero or one abstract abstraction and at least one concrete abstraction that specializes or implements the abstract abstraction. An abstract abstraction may not be necessary when there is only one abstraction. This is specified

in the multiplicity 0..* (not shown) on *AbstractAbstraction*. Similarly, *Implementor* hierarchy specifies the structure of implementor classifiers where there can be zero or one abstract implementor and at least one concrete abstraction that specializes or implements the abstract implementor. An abstract implementor may not be necessary when there is only one implementation. This is specified in the multiplicity 0..* (not shown) on *AbstractImplementor*. A classifier that conforms to *Abstraction* role must be a concrete class and have at least one operation playing *Operation* role.

#### A.5.1.1 Well-formedness Rules

The metamodel-level constraints defined on the Bridge SPS are given below:

- A classifier that conforms to *AbstractAbstraction* must be an interface or an abstract class:

  **context** |AbstractAbstraction **inv**:

  self.oclIsTypeOf(Interface) or

  (self.oclIsTypeOf(Class) and self.isAbstract = true)

  A similar constraint is defined for *AbstractImplementor*.

- A classifier that conforms to *ConcreteAbstraction* must be a concrete class:

  **context** |ConcreteAbstraction **inv**: self.isAbstract = false

  A similar constraint is defined for *ConcreteImplementor*.

- A relationship that conforms to *AbstractionRealization* must have an interface or a type at its supplier end and a concrete class at its client end:

  **context** |AbstractionRealization **inv**:

  (self.supplier.oclIsTypeOf(Interface) or

  (self.supplier.oclIsTypeOf(Class) and self.supplier.isAbstract = true) and

  self.client.oclIsTypeOf(Class)

A similar constraint is defined for *ImplementorRealization*.

- A relationship that conforms to *AbstractionGeneralization* must have its parent and child to be the same type:

  **context** |AbstractionGeneralization **inv**:

  self.parent.evaluationType() = self.child.evaluationType()

  A similar constraint is defined for *ImplementorGeneralization*.

- An association-end that conforms to *Abs* must have a multiplicity of 1..1:

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

- An association-end that conforms to *Imp* must have a multiplicity of 1..1:

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

#### A.5.1.2 Constraint Templates

An *Operation* call must invoke an *OperationImpl* operation call. This is defined as follows:

**context** |Abstraction::|Operation()

**pre**: true

**post**: |ConcreteImplementor^|OperationImpl()

### A.5.2 Conforming Class Diagram

Fig. A.9 [35] shows a conforming class diagram of the Bridge SPS. The diagram describes a Window application providing application windows and transient windows that work on both XWindow and PMWindow.

*Window* class defines the window abstraction for client applications, and maintains a reference to a *WindowImp* that declares an interface to the underlying windowing system. Subclasses of the *Window* class are the different kinds of windows that the

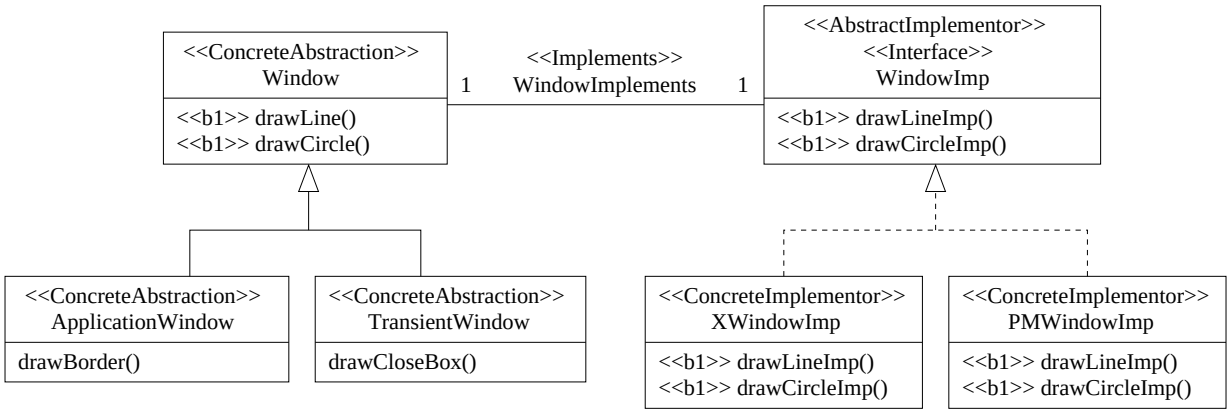



Figure A.9: A Conforming Bridge Class Diagram

application might use such as application windows, transient windows, and icon windows. The *WindowImp* class defines an interface to the underlying window systems such as XWindow System and IBM's Presentation Manager.

### A.5.3 IPS

Fig. A.10 shows a Bridge IPS. It specifies that a *ConcreteAbstraction* forwards client requests to a *ConcreteImplementor*. A conforming class diagram (taken from [35]) of the Bridge SPS is shown in Fig. A.9.

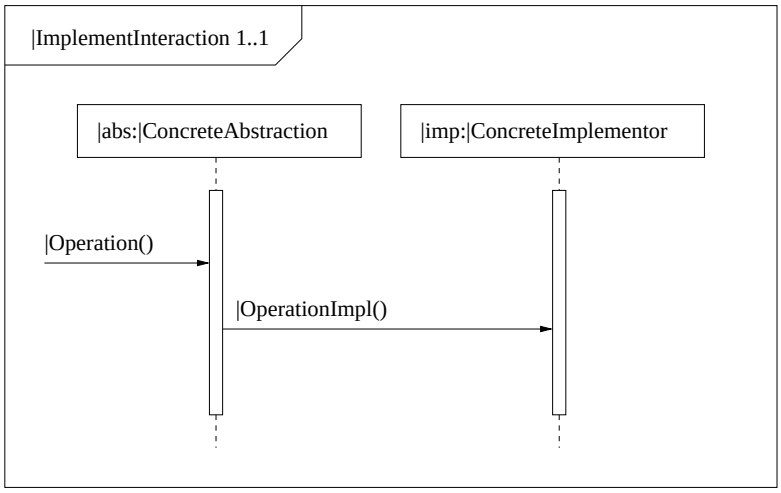



Figure A.10: A Bridge IPS

## A.6 The Decorator Pattern

The Decorator design pattern provides a flexible way to attach additional features (e.g., attributes, operations) to an object called *Component* dynamically through a class called *Decorator*. Components contains a hierarchy that separates abstraction from implementation to facilitate defining new components. The abstract component is implemented by concrete component classes to which additional features can be attached. The decorator class maintains a reference to a component class and defines an interface that conforms to the component's interface.

### A.6.1 SPS

The class diagrams characterized by a Decorator pattern possess component and decorator classifiers that are abstract or concrete. They have a hierarchy that separates abstract from implementation to facilitate defining new components and decorators. An abstract component is associated with an abstract decorator to maintain a reference to a component. This allows adding features recursively. An abstract component has a generalization/realization relationship with an abstract decorator representing concrete decorators themselves are also components.

Fig. A.11 shows a Decorator SPS which specifies the following. There must be one or more classifiers that play *ConcreteComponent* and *ConcreteDecorator*. There should be at least one generalization or realization in the structure of 1) components specified by *CompRealization* and *CompGeneralization*, 2) decorators specified by *DecoRealization* and *DecoGeneralization*, 3) abstract components and abstract decorators specified by *CompDecGen* and *CompDecReal*. This allows, for example, a structure in which an abstract component is specialized by an abstract decorator classifier, and the abstract decorator is realized by concrete decorators. A concrete decorator must have at least one new feature in addition to the features inherited from an abstract decorator. The abstract decorator might not be needed when there

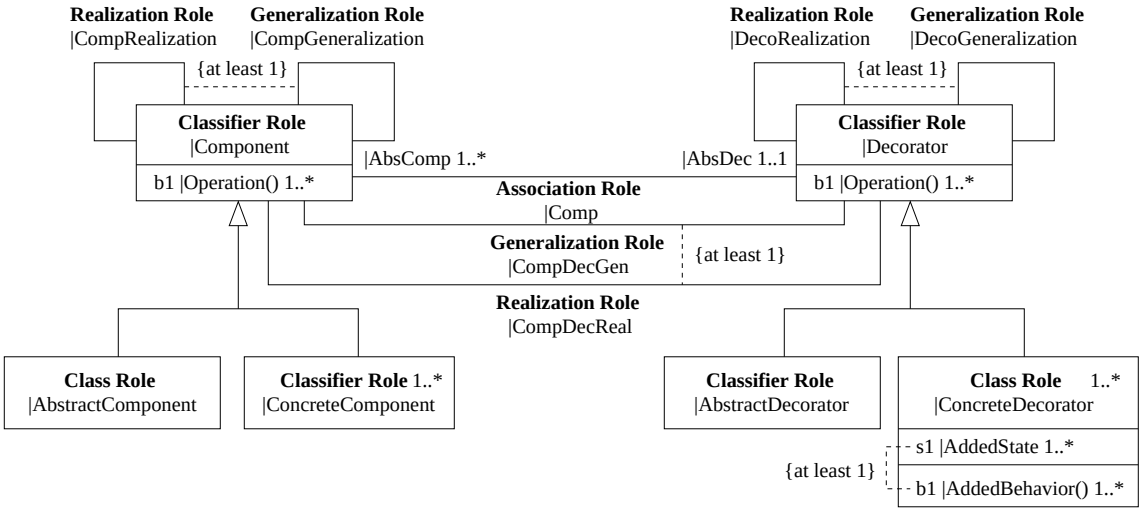



Figure A.11: A Decorator SPS

is only one feature to be added. This is specified by the multiplicity 0..* (not shown) in *AbstractDecorator*.

### A.6.2 Well-formedness Rules

Metamodel-level constraints for *AbsComp* and *AbsDec* roles are defined as:

- A classifier that conforms to *AbstractAbstraction* must be an interface or an abstract class:

  **context** |AbstractComponent **inv**:

  self.oclIsTypeOf(Interface) or

  (self.oclIsTypeOf(Class) and self.isAbstract = true)

  A similar constraint is defined for *AbstractDecorator*.

- A classifier that conforms to *ConcreteComponent* must be a concrete class:

  **context** |ConcreteComponent **inv**: self.isAbstract = false

  A similar constraint is defined for *ConcreteDecorator*.

- A relationship that conforms to *ComponentRealization* must have an interface or a type at its supplier end and a concrete class at its client end:

  **context** |ComponentRealization **inv**:

  (self.supplier.oclIsTypeOf(Interface) or

  (self.supplier.oclIsTypeOf(Class) and self.supplier.isAbstract = true) and

  self.client.oclIsTypeOf(Class)

  A similar constraint is defined for *DecoratorRealization*.

- A relationship that conforms to *ComponentGeneralization* must have its parent and child to be the same type:

  **context** |ComponentGeneralization **inv**:

  self.parent.evaluationType() = self.child.evaluationType()

  A similar constraint is defined for *DecoratorGeneralization*.

- A relationship that conforms to *CompDecReal* must have an interface or a type at its supplier end and a concrete class at its client end:

  **context** |CompDecReal **inv**:

  (self.supplier.oclIsTypeOf(Interface) or

  (self.supplier.oclIsTypeOf(Class) and self.supplier.isAbstract = true) and

  self.client.oclIsTypeOf(Class)

- A relationship that conforms to *CompDecGen* must have its parent and child to be the same type:

  **context** |CompDecGen **inv**:

  self.parent.evaluationType() = self.child.evaluationType()

- An association-end that conforms to *AbsComp* must have a multiplicity of 1..1:

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

- An association-end that conforms to *AbsDec* must have a multiplicity of 1..1:

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

### A.6.3 Conforming Class Diagram

A conforming class diagram (taken from [35]) of the Decorator SPS is shown in Fig. A.12. The model describes a graphical user interface toolkit that lets one decorate a TextView component with additional features like borders or scrolling by enclosing the component in a decorator that add the border or the scroll feature.

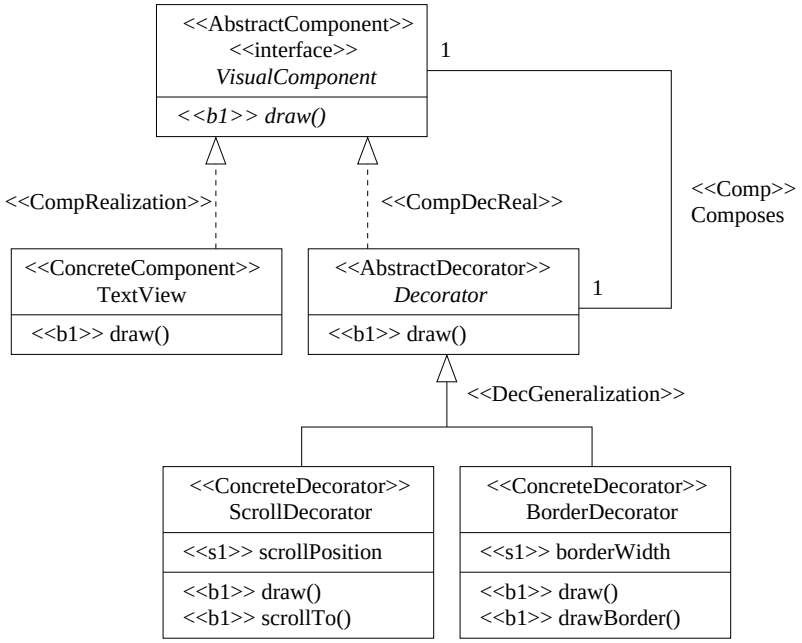



Figure A.12: A Conforming Decorator Class Diagram

### A.6.4 IPS

Fig. A.13 shows an IPS specifying that *ConcreteDecorator* forwards requests to *ConcreteComponent* and may perform additional actions before or after forwarding.

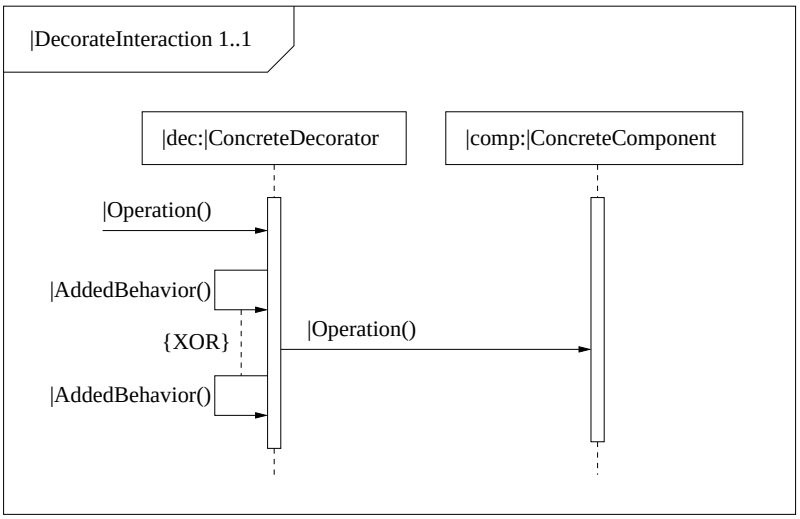



Figure A.13: A Decorator IPS

### A.6.5 Conforming Sequence Diagram

Fig. A.14 shows a conforming sequence diagram of the Decorator IPS. The diagram describes a sequence of adding a scroll to a text view. It first draws the text view and then add a scroll to it. The scroll could be added first before drawing the text view, which still conforms to the IPS.

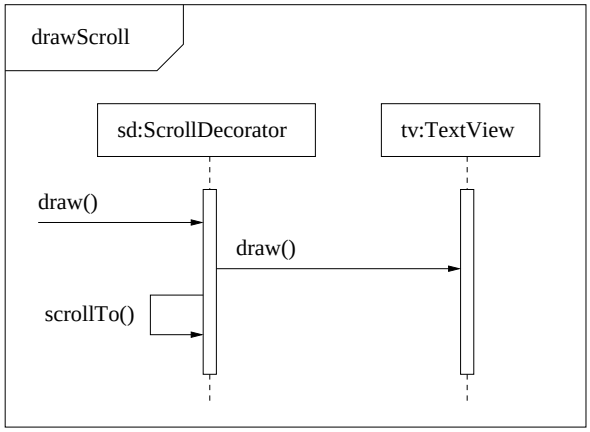



Figure A.14: A Conforming Decorator Sequence Diagram

## A.7 The State Pattern

The *State* design pattern allows an object called *Context* to alter its behavior when its internal behavior changes through objects called *State*. When an operation in the context is called, the context delegates the request to the current state class. The pattern enalbes a separation of abstraction from implemention for state classes to make it easy to change state classes without changing the context.

### A.7.1 SPS

The class diagrams characterized by a State SPS have concrete context classes and state classifiers that are concrete or abstract. Context classes have an attribute to maintain the current state and operations being requested by other objects. State classifiers have a structure that separates abstraction from implementation. State classifiers have operations that handle the delegated requests from the context.

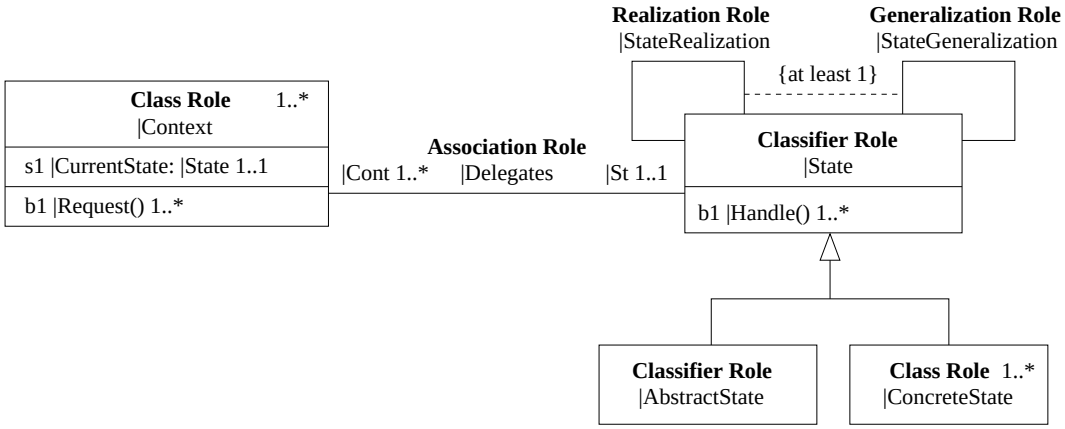



Figure A.15: A State SPS

Fig. A.15 shows an SPS for the State pattern. It specifies that there must be at least one context class that is associated with state classifiers. The context class must have one or more features that play *CurrentState* and *Request* roles. The hierarchy of state classifiers specifies that a structure of state classifiers must have at

least one *ConcreteState* class and relationships that play *StateRealization* or *State-Generalization*. The multiplicities at *Cont* and *St* roles specify that a state classifier can be associated one or more context classes, and a context class can be associated with exactly one state classifier. State classifiers must have at least one feature that plays *Handle*. When an object calls a *Request* operation in the context, the context delegates the request to the state object referenced in the *CurrentState* attribute.

#### A.7.1.1 Well-formedness Rules

The metamodel-level constraints defined on the State SPS are given below:

- A classifier that conforms to *Context* must be a concrete class:

  **context** |Context **inv**: self.isAbstract = false

- A classifier that conforms to *AbstractState* must be an interface or an abstract class:

  **context** |AbstractState **inv**:

  self.oclIsTypeOf(Interface) or

  (self.oclIsTypeOf(Class) and self.isAbstract = true)

- A classifier that conforms to *ConcreteState* must be a concrete class:

  **context** |ConcreteState **inv**: self.isAbstract = false

- A relationship that conforms to *StateRealization* must have an interface or a type at its supplier end and a concrete class at its client end:

  **context** |StateRealization **inv**:

  (self.supplier.oclIsTypeOf(Interface) or

  (self.supplier.oclIsTypeOf(Class) and self.supplier.isAbstract = true) and

  self.client.oclIsTypeOf(Class)

- A relationship that conforms to *StateGeneralization* must have its parent and child to be the same type:

  **context** |StateGeneralization **inv**:

  self.parent.evaluationType() = self.child.evaluationType()

- An association-end that conforms to *Cont* must have a multiplicity of 1..1:

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

- An association-end that conforms to *St* must have a multiplicity of 0..1:

  **context** |St **inv**: self.lowerBound() = 0 and self.upperBound() = 1

#### A.7.1.2 Constraint Templates

A *Request* operation call must invoke a *Handle* operation call to the current state object. This is specified as follows:

**context** |Context :: |Request()

**pre**: true

**post**: |ConcreteStateˆ|Handle()

### A.7.2 Conforming Class Diagram

A class diagram (taken from [35]) that conforms to the State SPS is shown in Fig. A.16. The diagram describes an application for TCP connection. The *TCP-Connection* class represents a network connection. Objects of the class can be in one of the states - Established, Listening, and Closed. The *TCPState* is an interface representing the states of the network connection. The interface defines *open*, *close*, and *acknowledge* operations which are implemented differently in each state. When a TCPConnection object receives a request from other objects, it delegates the request to the current state object.

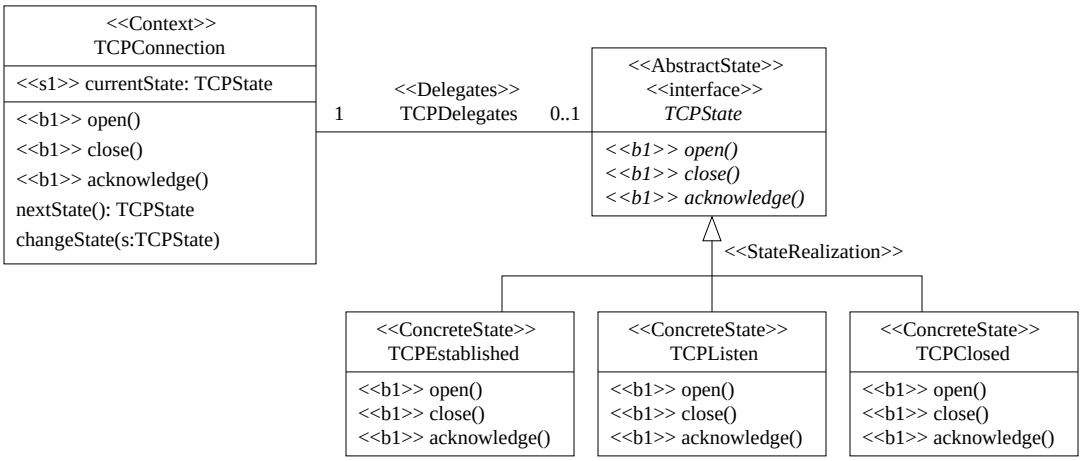



Figure A.16: A Conforming State Class Diagram

### A.7.3 IPS

Fig. A.17 shows a State IPS that specifies the delegation of an operation request from *Context* to *State*.

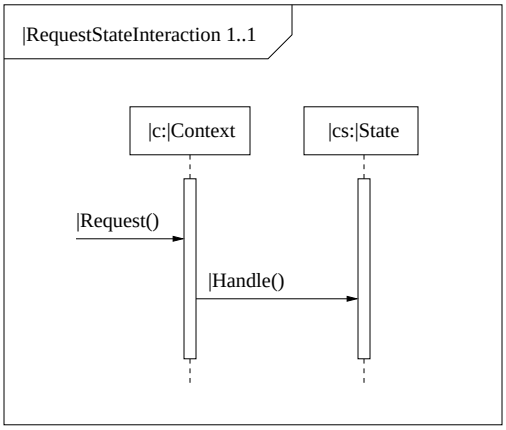



Figure A.17: A State IPS

# Appendix B

# A Domain Pattern and Access Control Patterns

## B.1 The CheckIn-CheckOut Pattern

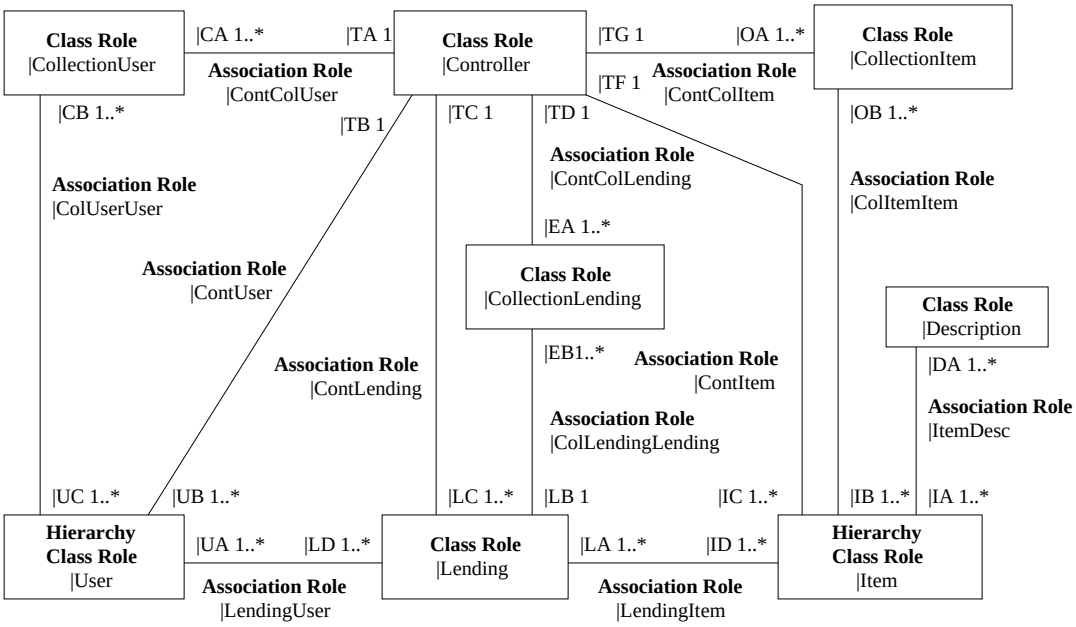



Figure B.1: The CICO SPS

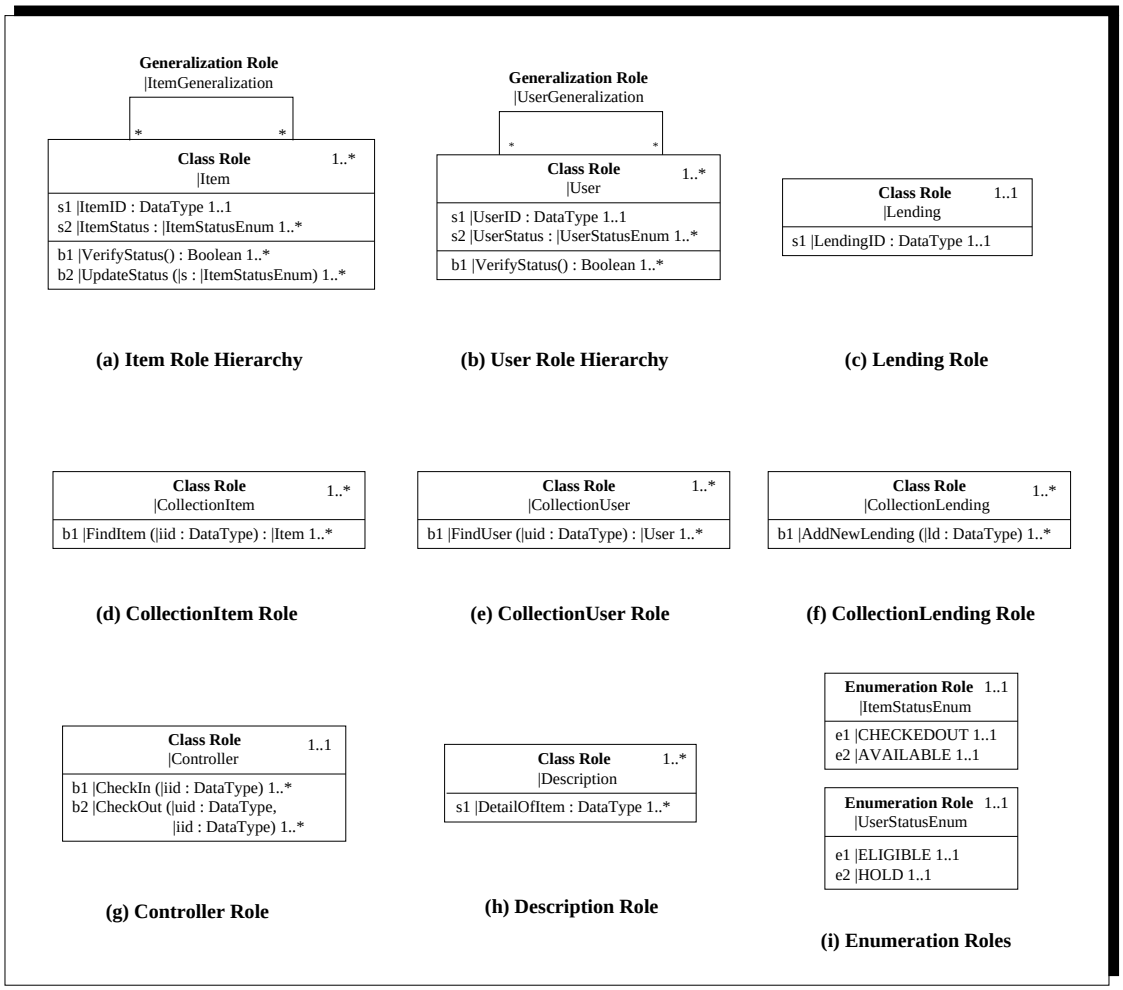
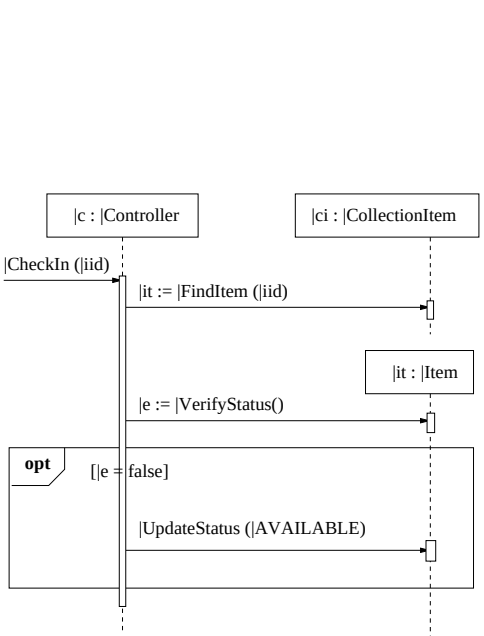
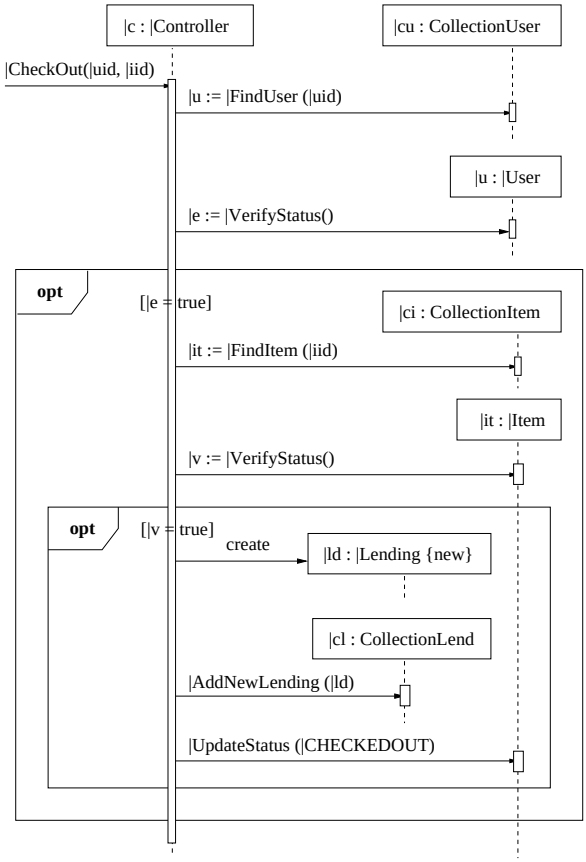



Figure B.2: CICO Role Hierarchies




(a) An IPS for CheckIn Scenario




(b) An IPS for CheckOut Scenario

Figure B.3: IPSs for CheckIn and CheckOut Scenarios

## B.2 RBAC

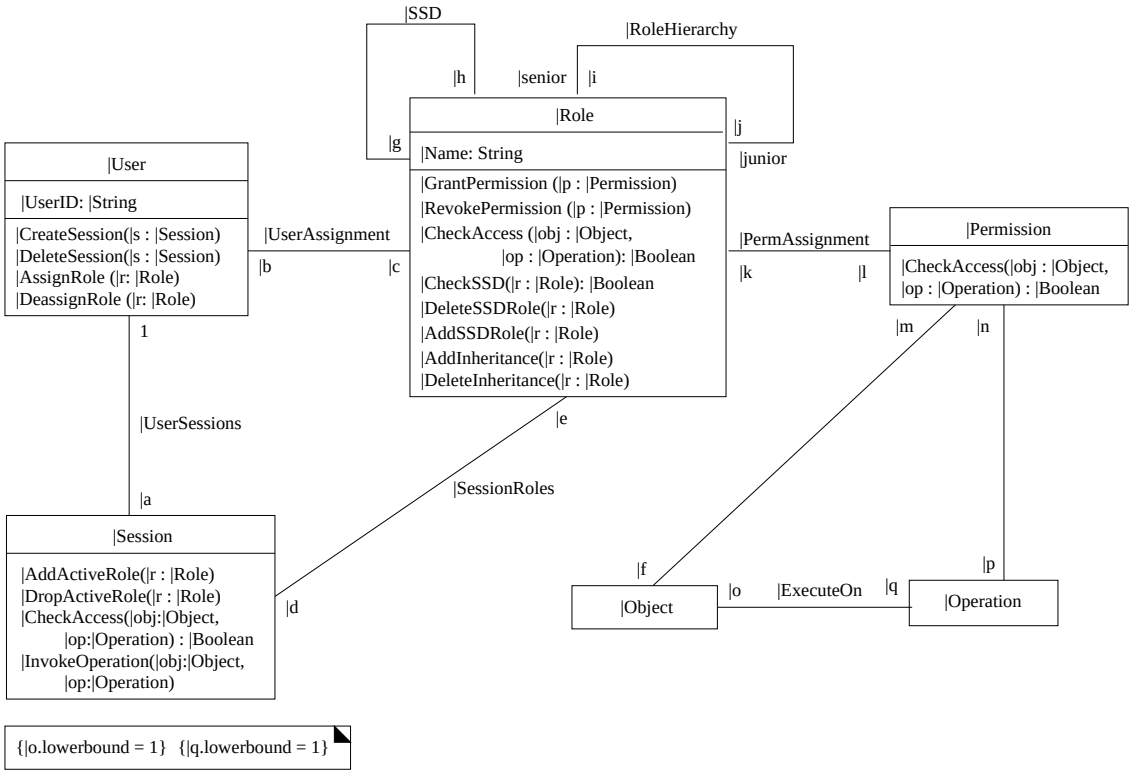



Figure B.4: RBAC Template

## B.3 MAC

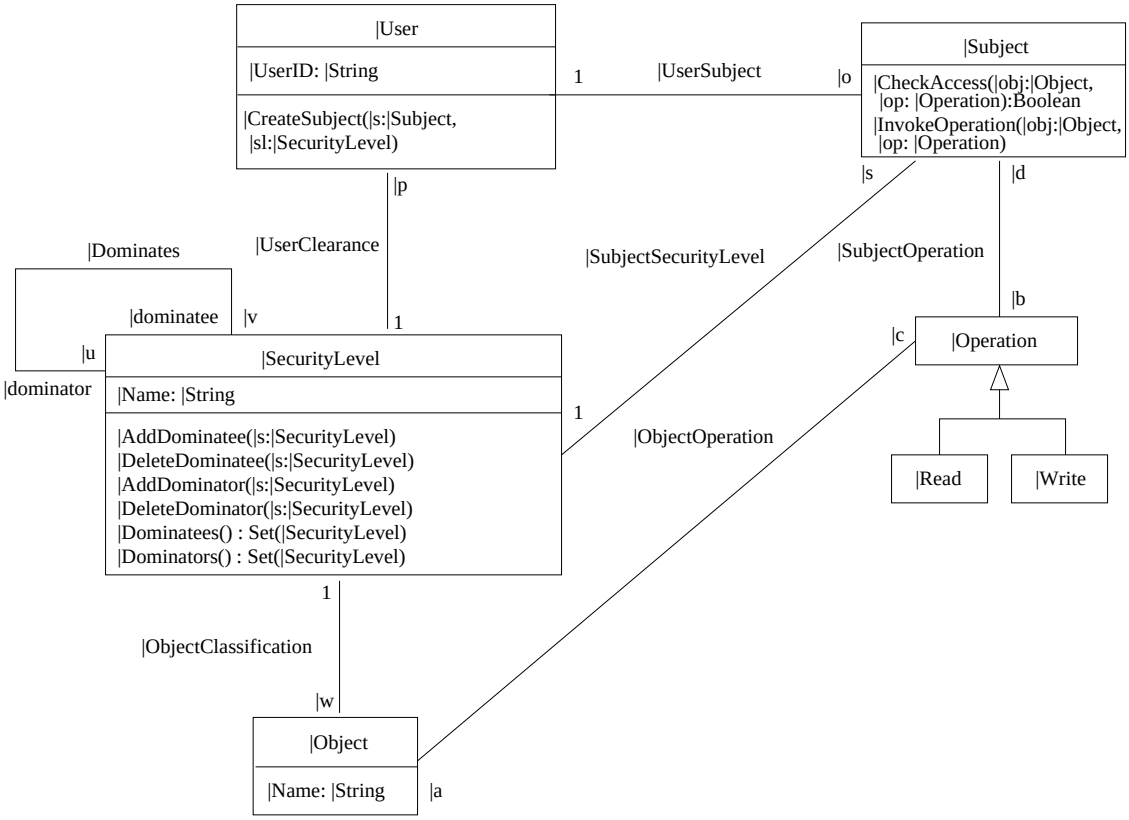



Figure B.5: MAC Template

# B.4 HAC

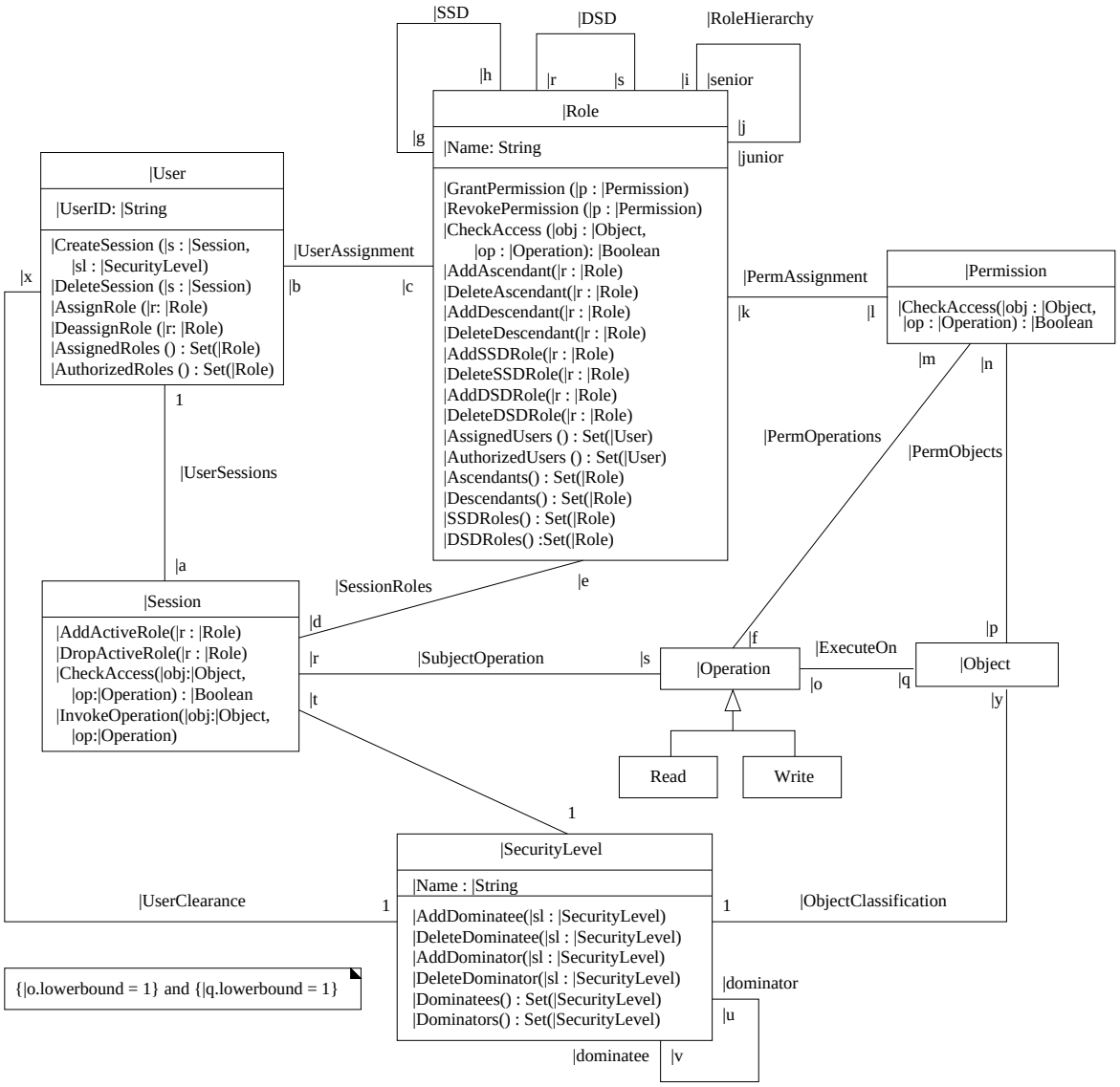



Figure B.6: HAC Template

# Appendix C

# Vehicle Rental System

Fig. C.1 shows a vehicle rental class diagram created using the CICO SPS. The diagram describes a design in which customers rent vehicles. A notable difference from the library system is that the pattern is further restricted to generate separate collection for each type of vehicle and customer. Two types of vehicles can be rented: trucks and leisure vehicles. Unlike the library example, the *Item* hierarchy is bound to a multi-level generalization hierarchy: the *Vehicle* class is specialized by *Truck* and *Leisure* and *Leisure* is further specialized by *Van* and *Sedan*.

The class diagram also includes classifiers and other diagram elements not specified by the CICO SPS. Like the Library system, there is a *Reservation* class. There is also an *InsurancyPolicy* class associated with the *Vehicle* class. An interface class, *CollectionVehicle*, is also added to the model to act as a common interface for the different types of vehicle collections. Vehicle rental scenarios obtained from the CICO IPSs are shown in Fig. C.2 and Fig. C.3.

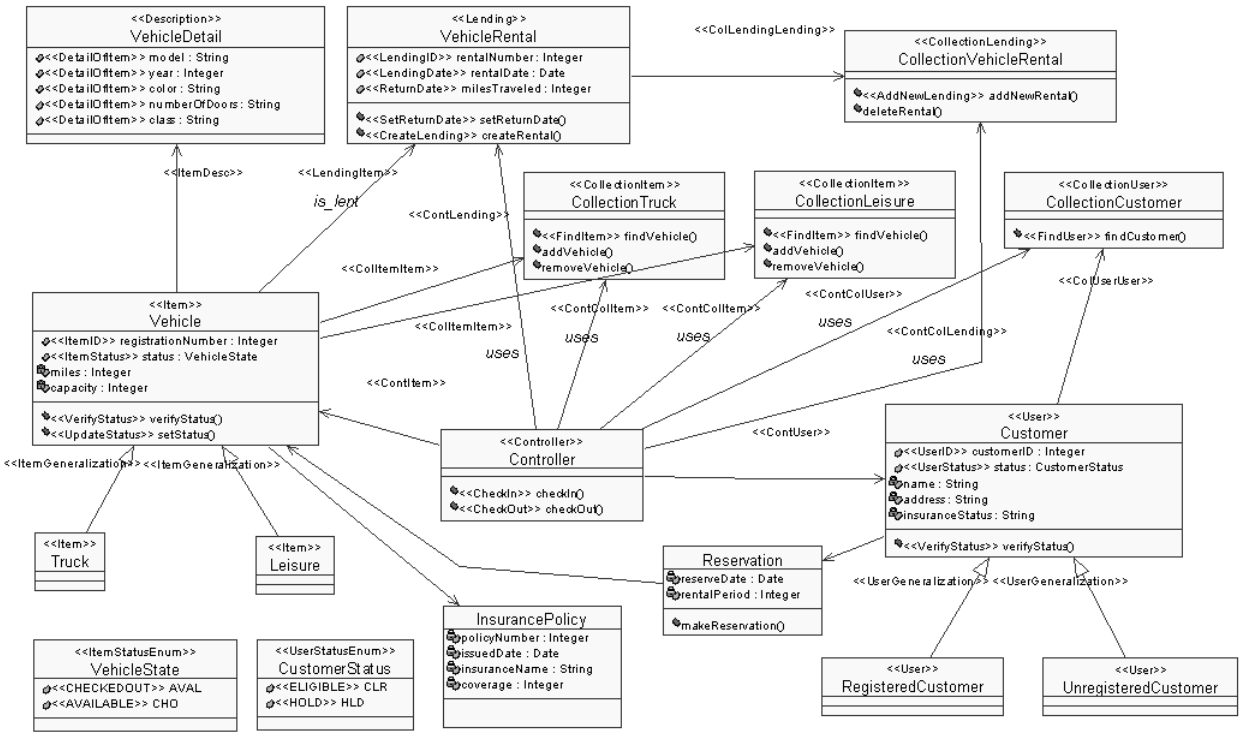



Figure C.1: A Vehicle Rental Class Diagram

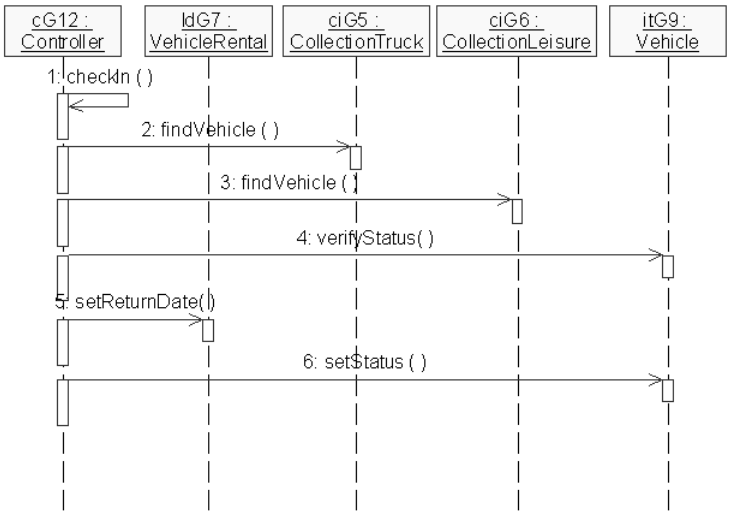



Figure C.2: A Vehicle Rental CheckIn Sequence Diagram

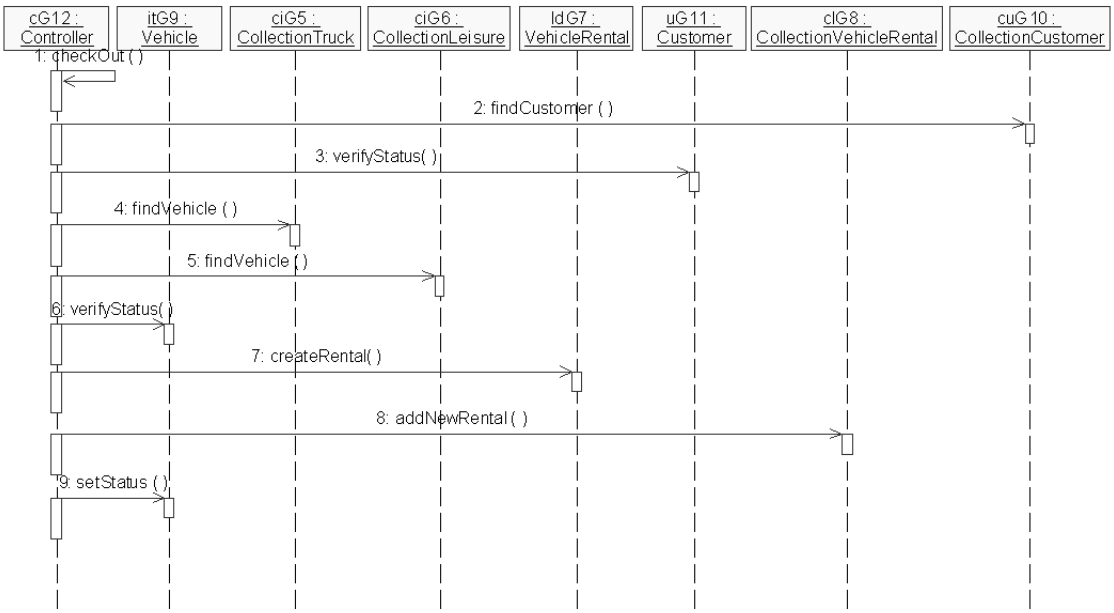



Figure C.3: A Vehicle Rental CheckOut Sequence Diagram

# REFERENCES


[1] E. Agerbo and A. Cornils. How to preserve the benefits of design patterns. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 134 –143, Vancouver, Canada, 1998.

[2] A. Albano, G. Ghelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[3] H. Albin-Amiot and Y. G. Gueheneuc. Meta-Modelling Design Patterns: Application to Pattern Detection and Code Synthesis. In *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[4] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[5] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Constructions*. Oxford University Press, 1977.

[6] G. Arango and R. Prieto-Diaz. Introduction and Overview: Domain Analysis Concepts and Research Directions. In R. Prieto-Diaz and G. Arango, editors, *Domain Analysis and Software Systems Modeling*, pages 9–32. IEEE Computer Society Press, 1991.

[7] J. Araujo, J. Whittle, and D. Kim. Modeling, Composing and Validating Scenario-Based Requirements with Aspects. In *Proceedings of 12th IEEE International Requirements Engineering Conference (RE)*, Kyoto, Japan, September 2004.

[8] C. W. Bachman and M. Daya. The Role Concept in Data Models. In *International Conference on Very Large Databases*, pages 464–476, 1977.

[9] S. Barker. Security Policy Specification in Logic. In *Proceedings of the International Conference on Artificial Intelligence*, pages 143–148, Las Vegas, NV, 2000.

[10] S. Barker and A. Rosenthal. Flexible Security Policies in SQL. In *Proceedings of the 15th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Niagara-on-the-Lake, Canada, 2001.

[11] J. Barwise. An Introduction to First-Order Logic. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 5–46. North-Holland, Amsterdam, 1977.

[12] V. R. Basili and H. D. Rombach. Support for Comprehensive Reuse. Technical Report UMIACS-TR-91-23, CS-TR-2606, Department of Computer Science, University of Maryland at College Park, 1991.

[13] E. Bertino, P. Bonatti, and E. Ferrari. TRBAC: A Temporal Role-Based Access Control Model. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 21–30, Berlin, Germany, 2000.

[14] R. T. V. Braga, F. S. R. Germano, and P. C. Masiero. A Family of Patterns for Business Resource Management. In *Proceedings of the 5th Annual Conference on Pattern Languages of Programs (PLoP'98)*, Monticello, IL, USA, 1998.

[15] R. T. V. Braga, F. S. R. Germano, and P. C. Masiero. A Pattern Language for Business Resource Management. In *Proceedings of the 6th Pattern Languages of Programs Conference (PLoP'99)*, volume 7, pages 1–34, Monticello, IL, USA, 1999.

[16] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. Wiley, 1996.

[17] R. Chandramouli. Application of XML Tools for Enterprise-Wide RBAC Implementation Tasks. In *Proceedings of the 5th ACM Workshop on Role-based Access Control*, Berlin, Germany, July 2000.

[18] K.M. Chandy and J. Misra. *Parallel Program Design - A Foundation*. Addison-Wesley, 1988.

[19] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[20] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, 1995.

[21] W. Cunningham and K. Beck. A Diagram for Object-Oriented Programs. In *Proceedings of OOPSLA*, pages 361–367, Portland, Oregon, September 1986.

[22] M. Dahchour, A. Pirotte, and E. Zimanyi. A Generic Role Model for Dynamic Objects. In *Proceedings of the 14th Advanced Information Systems Engineering nternational Conference, CAiSE'02*, pages 643–658, Toronto, Canada, May 2002.

[23] E. Durr and E. Dusink. The Role of VDM ++ in the Development of a Real-Time Tracking and Tracing System. In J. Woodcock and P. Larsen, editors, *Proceedings of Formal Methods Europe (FME'93), Lecture Notes in Computer Science*, pages 64–72. Springer Verlag, 1993.

[24] A. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application.* PhD thesis, University of Tel Aviv, Israel, 1999.

[25] B. A. Farshchian and S. Jakobsson. Coupling MDA and Parlay to increase reuse in telecommunication application development. In *Proceedings of Workshop on Software Model Engineering (WiSME) at UML 2002*, Dresden, Germany, Octorber 2003.

[26] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and Systems Security*, 4(3), August 2001.

[27] J. Fiadeiro and T. Maibaum. Sometimes "Tomorrow" is "Sometime" Action Refinement in a Temporal Logic of Objects. In *Proceedings of the 1st International Conference on Temporal Logic*, volume 827 of *Lecture Notes in Artificial Intelligence*, pages 48–66. Springer-Verlag, 1994.

[28] G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Object-Oriented Patterns. In *Proceedings of the 11th European Conference on Object Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 472–495. Springer-Verlag, 1997.

[29] Software Technology for Adaptable Reliable Systenms (STARS). STARS Conceptual Framework for Reuse Processes, Volume 1: Definition, Version 3.0. Technical Report STARS-VC-A018/001/00, Unisys STARS Technology Center, October, 1993.

[30] R. France, G. Georg, and I. Ray. Supporting Multi-Dimensional Separation of Design Concerns. In *Proceedings of the AOSD Workshop on AOM: Aspect-Oriented Modeling at UML 2003*, March 2003.

[31] R. France, S. Ghosh, E. Song, and D. Kim. A Metamodeling Approach to Pattern-Based Model Refactoring. *IEEE Software*, 20(5), September/October 2003.

[32] R France, D. Kim, S. Ghosh, and E. Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, March 2004.

[33] R. France, D. Kim, E. Song, and S. Ghosh. Using Roles to Characterize Model Families. In Haim Kilov, editor, *Practical foundations of business and system specifications*, pages 179–195. Kluwer Academic Publishers, 2003.

[34] U. Frank. Delegation: An Important Concept for the Appropriate Design of Object Models. *Journal of Object Oriented Programming*, 44:13–17, June 2000.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[36] G. Georg, R. France, and I. Ray. Composing Aspect Models. In *Proceedings of the 4th AOSD Modeling With UML Workshop, UML 2003*, San Francisco, CA, October 2003.

[37] G. Georg, R. France, and I. Ray. Creating Security Mechanism Aspect Models from Abstract Security Aspect Models. In *Proceedings of the Workshop on Critical Systems Development at UML 2003*, San Francisco, CA, October 2003.

[38] Geri Georg, Robert France, and Indrakshi Ray. Designing High Integrity Systems using Aspects. In *Proceedings of the Fifth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2002)*, Bonn, Germany, November 2002.

[39] G. Gnesi, D. Latella, and M. Massink. A Stochastic Extension of a Behavioral Subset of UML Statechart Diagrams. In L. Palagi and R. Bilof, editors, *Proceedings of the 5th IEEE International High-Assurance Systems Engineering Symposium*, pages 55–64. IEEE Computer Society Press, 2000.

[40] G. Gottlob, S. Michael, and B. Rock. Extending Object - Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[41] M. Grand. *Patterns in Java-A catalog of reusable design patterns illustrated with UML*. Wiley, 1999.

[42] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, 44(10):87–93, October 2002.

[43] M. L. Griss. Software Reuse: From Library to Factory. *IBM Systems Journal*, 32(4):1–23, 1993.

[44] A.L. Guennec, G. Sunye, and J. Jezequel. Precise Modeling of Design Patterns. In *Proceedings of UML'00*, pages 482–496, 2000.

[45] R. J. Hayton, J. M. Bacon, and K. Moody. Access Control in Open Distributed Environment. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA, May 1998.

[46] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering*. Addison-Wesley, 2001.

[47] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic Design Pattern Detection. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, Portland, Oregon, May 2003.

[48] M. Hitchens and V. Varadarajan. Tower: A Language for Role-Based Access Control. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.

[49] J. A. Hoagland, R. Pandey, and K. N. Levitt. Security Policy Specification Using a Graphical Approach. Technical Report CSE-98-3, Computer Science Department, University of California Davis, July 1998.

[50] Na Li Indrakshi Ray, Robert France and Geri Georg. An Aspect-Based Approach to Modeling Access Control Concerns. *Information and Software Technology*, 46(9):557–633, July 2004.

[51] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[52] S. Judson and R. France. Model Transformations at the Metamodel Level. In *Proceedings of the Workshop in Software Model Engineering, UML'03 Conference*, October 2003.

[53] J. Jurjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on the Unified Modeling Language*, pages 412–425, Dresden, Germany, October 2002.

[54] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analaysis FODA: Feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, CMU, 1990.

[55] G. Karsai. Tool Support for Design Patterns. In *Proceedings of New Directions in Software Technology (NDIST) 4 Workshop*, December 2001.

[56] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyvaskyla, Finland, June 1997.

[57] D. Kim, R. France, and S. Ghosh. A UML-Based Language for Specifying Domain-Specific Patterns. *Journal of Visual Languages and Computing, Special Issue on Domain Modeling with Visual Languages*, 15(3-4), June 2004.

[58] D. Kim, R. France, S. Ghosh, and E. Song. Using Role-Based Modeling Language (RBML) as Precise Characterizations of Model Families. In *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002)*, Greenbelt, MD, December 2002.

[59] D. Kim, R. France, S. Ghosh, and E. Song. A UML-Based Metamodeling Language to Specify Design Patterns. In *Proceedings of the Workshop on Software Model Engineering (WiSME) at UML 2003*, San Francisco, CA, October 2003.

[60] D. Kim, R. France, S. Ghosh, and E. Song. A Role-Based Metamodeling Approach to Specifying Design Patterns. In *Proceedings of the 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC)*, Dallas, TX, November, 2003.

[61] D. Kim, S. Ghosh, R. France, and E. Song. Software Component Specification Using Role-Based Modeling Language. In *Proceedings of 11th OOPSLA Workshop on Behavioral Semantics: Serving the Customer*, Seattle, Washington, November 2002.

[62] D. Kim, I. Ray, R. France, and N. Li. Modeling Role-Based Access Control Using Parameterized UML Models. In *Proceedings of Fundamental Approaches to Software Engineering (FASE/ETAPS)*, Barcelona, Spain, March 2004.

[63] C. Kobryn. UML 2001: A Standardization Odyssey. *CACM*, 10(42):29–37, 1999.

[64] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[65] K. Lano, J. Bicarregui, and S Goldsack. Formalising Design Patterns. In *Proceedings of the 1st BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computer Science*. Springer-Verlag, 1996.

[66] A. Lauder and S. Kent. Precise Visual Specification of Design Patterns. In *Proceedings of ECOOP'98*, pages 114–136, 1998.

[67] T. Lewis, L. Rosenstein, W. Pree, A. Weinand, E. Gamma, P. Calder, G. Andert, J. Vlissides, and K. Schmucker. *Object Oriented Application Frameworks*. Manning Publication Co., Greenwich, CT, 1995.

[68] T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the 5th International Conference on the Unified Modeling Language*, pages 426–441, Dresden, Germany, October 2002.

[69] D. Mapelsden, J. Hosking, and J. Grundy. Design Pattern Modelling and Instantiation using DPML. *ACM International Conference Proceeding Series, Proceedings of the 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, 10:3–11, 2002.

[70] T. Mikkonen. Formalizing Design Patterns. In *Proceedings of the 20th International Conference on Software Engineering*, pages 115–124, Kyoto, Japan, April 1998.

[71] H. Mili, A. Mili, S. Yacoub, and E. Addy. *Reuse-Based Software Engineering: Techniques, Organization, and Controls*. John Wiley & Sons, 2002.

[72] J.-M. Morel and J. Faget. The REBOOT Environment. In *Proceedings of the Second International Workshop on Software Resuability: Advances in Software Reuse*, pages 80–88, Lucca, Italy, March 1993. IEEE Computer Society Press.

[73] M. Nyanchama and S.L. Osborn. Modeling Mandatory Access Control in Role-Based Security Systems. In *Proceedings of Database Security IX: Status and Prospects, Spooner, Demurjian and Dobson, eds. Chapman & Hall*, pages 129–144, August 1997.

[74] OASIS. XACML Language Proposal, Version 0.8. Technical report, Organization for the Advancement of Structured Information Standards, January 2002. Available electronically from http://www.oasis-open.org/committees/xacml.

[75] R. Ortalo. A Flexible Method for Information Systems Security Policy Specification. In *Proceedings of the 5th European Symposium on Research in Computer Security*, Louvain-la-Neuve, Belgium, 1998. Springer-Verlag.

[76] S. Osborn. Mandatory Access Control and Role-Based Access Control Revisited. In *Proceedings of the 2nd ACM Workshop on Role-Based Access Control*, Fairfax, Virginia, 1997.

[77] B.-U. Pagel and M. Winter. Towards pattern-based tools. In *Proceedings of EuropLop*, München, 1996.

[78] B. Pernici. Objects with Roles. In *Proceedings of the conference on Office information systems*, pages 25–27, Cambridge, MA, April 1991.

[79] C. E. Phillips, S. A. Demurjian, and T. C. Ting. Toward Information Assurance in Dynamic Coalitions. In *Proceedings of the IEEE Workshop on Information Assurance*, United States Military Academy, West Point, NY, June 2002.

[80] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.

[81] R. S. Pressman. *Software Engineering, A Practitioner's Approach. 5th Edition*. McGraw Hill, 2001.

[82] R. Prieto-Diaz. Status Report: Software Reusability. *IEEE Software*, 10(3):61–66, 1993.

[83] A. Radermacher. Support for Design Patterns through Graph Transformation Tools. In *Proceedings of Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, pages 111–126, 1999.

[84] I. Ray, N. Li, D. Kim, and R. France. Using Parameterized UML to Specify and Compose Access Control Models. In *Proceedings of the 6th IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS)*, Lausanne, Switzerland, November 13-14 2003.

[85] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OORAM Software Engineering Method*. Manning/Prentice Hall, 1996.

[86] C. Ribeiro, A. Zuquete, and P. Ferreira. SPL: An Access Control Language for Security Policies with Complex Constraints. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2001.

[87] RM-ODP, International Standard Organisation (ISO). Information technology - Open Distributed Processing - Reference model: Enterprise Language. Technical Report ISO/IEC, 15414, ITU-T Recommendations X.911, Amendment1: Additional text, ISO, 2001.

[88] G. F. Rogers. *Framework-Based Software Development in C++*. Prentice Hall, 1997.

[89] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[90] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual. 2nd Edition*. Addison-Wesley, 2004, To be published.

[91] R. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communications*, 32(9), September 1994.

[92] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[93] J. Schumann and J. Whittle. Automatic Synthesis of Agent Designs in UML. In *Proceedings of FAABS*, 2000.

[94] Edward Sciore. Object Specilazation. *ACM Transactions on Information Systems*, 7(2):103–122, April 1989.

[95] M. Sefikla, A. Sane, and R. H. Campbell. Monitoring Compliance of a Software System with its High-Level Design Models. In *Proceedings of the International Conference on Software Engineering, 1996*, 1996.

[96] E. Song, R. France, D. Kim, and S. Ghosh. Using Roles for Pattern-Based Model Refactoring. In *Proceedings of the Workshop on Critical Systems Development at UML 2002*, Dresden, Germany, October 2002.

[97] F. Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data and Knowledge Engineering*, 35(1):83–106, 2000.

[98] L. A. Stein and S. B. Zdonik. Clovers: The Dynamic Behavior of Types and Instances. Technical Report CS-89-42, Department of Computer Science, Brown University, Providence, RI, November 1, 1989.

[99] The Object Management Group (OMG). Unified Modeling Language. Version 1.4, OMG, http://www.omg.org, September 2001.

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

[101] W. Tracz, L. Coglianese, and P. Young. Domain-Specific Software Architecture Engineering Process Outline. *ACM SIGSOFT Software Engineering Notes*, 18(2):40–49, 1993.

[102] B. Unger and W. F. Tichy. Do Design Patterns Improve Communication? An Experiment with Pair Design. In George Stark, editor, *Proceedings of International Workshop Empirical Studies of Software Maintenance*, http://members.aol.com/GEShome/wess2000/unger-tichy.pdf, October 2000.

[103] J. Warmer and A. Kleppe. *The Object Constraint Language, Second Edition*. Addison-Wesley, 2003.

[104] J. Whittle, J. Araujo, and D. Kim. Modeling and Validating Interaction Aspects in UML. In *Proceedings of the 4th AOSD Modeling With UML Workshop, UML 2003*, San Francisco, CA, October 2003.

[105] R. Wieringa and W. D. Jonge. The Identification of Objects and Roles: Object Identifier Revisited. Technical Report lR-267, Vrije University, Amsterdam, 1991.

[106] D. S. Wile and J. C. Ramming. Special Section: Domain Specific Languages. *IEEE Transactions on Software Engineering*, 25(3):289–290, 1999.