

Using Role Models as Precise Characterizations of Model Families

Sudipto Ghosh, Dae-Kyoo Kim, Robert France, Eunjee Song
Computer Science Department
Colorado State University
Fort Collins, CO 80523, USA

{ghosh, dkkim, france, song}@cs.colostate.edu

ABSTRACT

Cost-effective development of large, integrated computer-based systems can be realized through systematic reuse practices which require the creation of reusable artifacts. Such artifacts can be obtained from all phases of software development. In our work, we focus on artifacts that describe software design models. The development of reusable design models requires one to characterize families of problems and their solutions. We use a notation that we call Role Models to characterize families of UML design models. A Role Model is a specialization of the UML (Unified Modeling Language) meta-model, that is, it is a sub-language of the UML. We show how Role Models can describe structural and behavioral properties of model families. We illustrate the use of our Role Models to describe a family of designs for checkin-checkout systems.

Keywords

Object-oriented design models, role models, software reuse, UML

1. INTRODUCTION

Cost-effective development of large, integrated computer-based systems can be realized through systematic reuse practices. Advantages of incorporating such practices into a development environment are widely known. Arango and Prieto-Diaz [1] show that reuse of domain-specific experiences can significantly enhance development productivity and quality.

Reusable experiences can be classified as vertical or horizontal. Horizontal reuse occurs when reusable artifacts are created for, and used in a variety of application do-

main. The design patterns described in [12, 19] and the software architectures described in [3, 6, 24] are examples of horizontal experiences packaged for reuse.

Vertical reuse occurs when development experiences within an application domain are reused. When compared to horizontal artifacts, domain-specific artifacts for a mature domain¹ are easier to build, and can cover a wider range of development phases, primarily because of their restricted scope. For example, domain-specific programming languages (DSPLs) [27] provide examples of vertical reuse. A DSPL provides a language interface for assembling domain-specific code components into programs.

There is a growing realization that development cycle times can be significantly shortened if reuse opportunities are exploited in all phases of software development, not just in the coding phase [2, 18, 20]. In the past, a barrier to the reuse of experiences above the code level was the lack of widely-accepted notations for representing requirements and design artifacts. The advent of the Unified Modeling Language (UML) [25] as a de-facto industry object-oriented (OO) modeling standard has the potential to remove this barrier. Since the use of the UML in industry is growing, we are interested in specification techniques that characterize UML designs. In order to leverage UML tools, we have based our notation on the UML infrastructure where possible.

In [10] we present a technique for precise characterization of design patterns using *Role Models*. A Role Model determines a specialization of the UML meta-model, that is, it is a sub-language of the UML. Structural and behavioral properties of a pattern are specified using our Role Models. In this paper, we focus on reusable artifacts that characterize families of problems and their solutions. We apply the Role Models to describe families of problems that possess similar characteristics even though they occur in different application domains. Consider the *Checkin-Checkout* (CICO) problem. We see examples of this problem in domains such

¹A mature domain is one in which substantial codifiable experience exists

as video-rental applications, car rental applications and university libraries. In all these applications, items are checked out by users and returned (checked in) after use. The applications need to maintain a collection (or inventory) of the items. In this paper we illustrate the use of Role Models to describe CICO systems. We use the Role Models to describe the structural and behavioral properties of model families. The concepts used in our work relate to concepts defined at the UML meta-model level (M2 level). Since we use UML graphical notation to represent our Role structures, UML tools can be used to build the graphical forms of our pattern specifications.

In addition to the development of reusable artifacts, effective reuse also requires good support for adapting and incorporating the artifacts into project deliverables. We illustrate the task of tailoring and integrating reusable experiences by showing how the CICO Role Model may be realized by different design models. The realizations would depend on the application domain where the CICO system is being used. Different CICO systems may have different requirements. For example, items may be grouped into collections in different ways (books in university libraries, designs and code in software versioning). Role Models not only help software designers to abstract common characteristics of model families, but also enable them to realize variations in the designs to satisfy application requirements.

The remainder of the paper is organized as follows. We describe Role Models in Section 2 and illustrate their use to specify the CICO system in Section 3. Realizations of the CICO Role Model are shown in Section 4. We provide an overview of related work in Section 5. We conclude in Section 6.

2. ROLE MODELS

We define a *role* to be a property-oriented specification that determines a subset of the role's base instances, where a role *base* can be any UML metamodel class (e.g., *Class*, *Generalization*). For example, a role with the *Class* base determines a subset of class constructs. An instance of a role's base that has the properties specified in a role can *play the role*, that is, it is a *realization* of the role. A *Role Model* is a structure of roles. A *Role Model realization* is a model (e.g., a static structural diagram, a sequence diagram) that consists of realizations of the roles in the Role Model.

Role Models can be used to precisely express families of designs. In [10] we used Role Models to precisely express pattern specifications. Role Models characterize UML models, thus they are based on the UML metamodel. The UML metamodel is defined at level M2 of the UML metamodel architecture. Level M1 consists of UML models (i.e., instances of the M2 metamodel) and level M0 consists of instances of the models at level M1. Two types of Role Models can be used in a specification and these are more fully described in [10] and [11].

Static Role Models (SRMs): A SRM is a characterization of a family of UML static structural models, that is, models that depict classifiers (e.g., UML classes and interfaces) and their relationships with each other (e.g., UML associations and generalizations).

Interaction Role Models (IRMs): An IRM is a characterization of a family of interaction diagrams (e.g., collaboration and sequence diagrams).

2.1 Static Role Models (SRMs)

A SRM consists of roles and relationships between roles. In this subsection we describe SRM roles and the relationships (associations and generalization/specialization) that can exist between them.

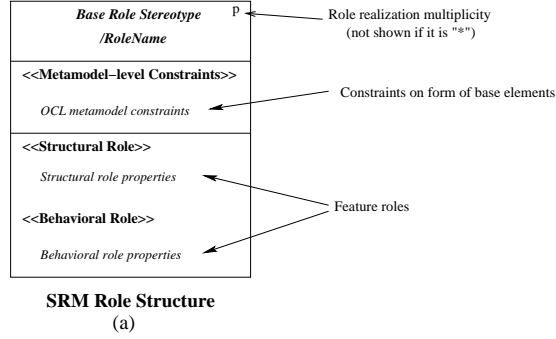
2.1.1 SRM Roles

A SRM role characterizes a set of UML static modeling constructs (e.g., class and association constructs). For example, a SRM classifier role (i.e., a SRM role with the metamodel class *Classifier* as a base) defines properties that classifiers (e.g., classes, interfaces) must have if they are to realize the role, while a SRM relationship role defines properties that UML relationships (e.g., associations, generalizations) must have if they are to realize the role.

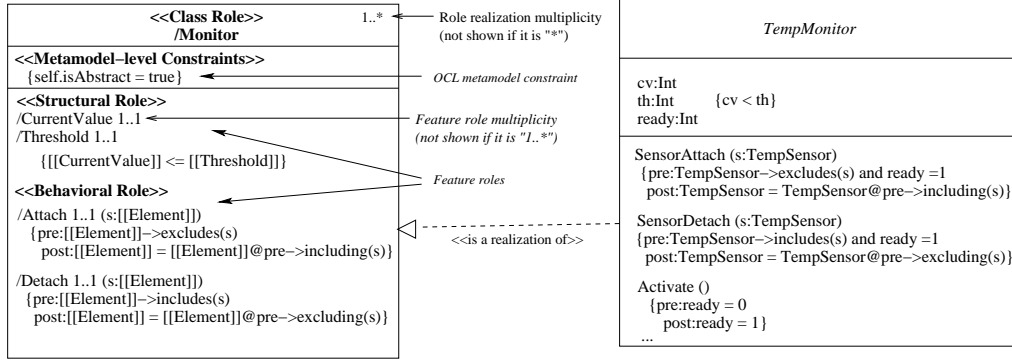
The structure of a SRM role is shown in Fig. 1(a). The top compartment has three parts: a role base declaration of the form $\ll Base\ Role \gg$, where *Base* is the name of the role's base (i.e., the name of a metamodel class); a role name declaration of the form */RoleName*, where *RoleName* is the name of the role; and a *realization multiplicity* that specifies the allowable number of realizations (shown by *P* in Fig. 1(a)) that can exist for the role in a realization of the SRM that includes the role. In this section we do not show role realization multiplicities if they are “*”. The remaining compartments contain specifications of the properties that realizations of the role must possess. The second compartment contains metamodel-level constraints and the third compartment contains properties expressed as feature roles.

Metamodel-level constraints are well-formedness rules, expressed in the Object Constraint Language (OCL) [25], that determine the form of UML metamodel class instances that can realize the role. Specifically, the UML well-formedness rules and the metamodel-level constraints defined in a SRM role determine the form of its realizations.

Feature roles characterize application-specific properties. A feature role consists of a name, a *realization multiplicity*, and a property specification expressed as a *constraint template*. The realization multiplicity specifies the number of realizations a feature role can have in a SRM realization. In this section, we do not show feature role realization multiplicities if they are “1..*”. The *constraint template* of a feature role determines a family of application-specific properties. Features (e.g.,



SRM Role Structure
(a)



Example of a SRM Role and a Realization
(b)

Figure 1: Structure of a SRM Role

attributes, operations) of model constructs that play a SRM role realize feature roles of the SRM role. There are two types of feature roles:

(1) *Structural roles* specify state-related properties that are realized by attributes or value-returning operations in a SRM role realization. An example of a structural role that can be realized by class attributes is given below (see Fig. 1(b)):

```
/CurrentValue 1..1
{[[CurrentValue]] <= [[Threshold]]}
```

In the above example, *CurrentValue* is the feature role name and the realization multiplicity following it (*1..1*) indicates that there must be exactly one realization of this role in a realization of the SRM role containing this feature role. The constraint template enclosed in the braces {,} (parameters are surrounded by [,]) states that realizations of the *CurrentValue* feature role must always have a value less than or equal to the value of a realization of another feature role named *Threshold*.

(2) *Behavioral roles* specify behaviors that are realized by a single operation or method, or by a composition of

operations or methods in a SRM role realization. An example of a behavioral role is given below (see Fig. 1(b)):

```
/Attach 1..1 (s: [[Element]])
{pre: [[Element]] -> excludes(s)
 post: [[Element]] = [[Element]]@pre
 -> including(s)}
```

The name of the above property is *Attach*. The realization multiplicity following the name (*1..1*) indicates that there must be exactly one behavior that realizes this role in a realization of the SRM role containing this feature role. The specification of the behavior references a value *s* that is an instance of a realization of a SRM role called *Element*. The constraint template consists of pre-condition and post-condition templates. The pre-condition template states that before execution of a behavior realizing *Attach*, *s* is not in the set of element objects (instances of an *Element* realization) known to the object that performs the *Attach* behavior. The post-condition template states that the effect of a realization of *Attach* is to include *s* in the set of element objects known to the performing object. The expression *x@pre* in the OCL refers to the value of *x* before execution of an operation.

Substituting the names of realizations for the (feature or SRM) role names enclosed in the double square brackets ($\llbracket \cdot \rrbracket$) results in an application-specific property, called a *model-level constraint*, expressed in the OCL. For example, substituting *Sensor* (a realization of *Element*) for *Element* in the *Attach* constraint template results in the following model-level constraint:

```
Attach(s:Sensor)
{pre:Sensor -> excludes(s)
 post: Sensor = Sensor@pre -> including(s)}
```

The metamodel-level constraint, *self.isAbstract = true*, and the base role stereotype, $\ll ClassRole \gg$, indicate that the realizations of the *Monitor* role are abstract classes. The realization multiplicity of the *Monitor* role indicates that there must be one or more realizations of the *Monitor* role in a realization of the SRM that includes the role. The realizing classes must have one realization for each of the *CurrentValue* and *Threshold* (as indicated by the multiplicities “1..1”), and one realization for each of the behaviors, *Attach* and *Detach*.

The abstract class *TempMonitor* shown in Fig. 1(b) is a realization of *Monitor* under the following role realization mapping:

- *cv* realizes *CurrentValue*
- *th* realizes *Threshold*
- *SensorAttach* realizes *Attach*
- *SensorDetach* realizes *Detach*
- The class *TempSensor* realizes *Element*

Establishing that *TempMonitor* is a realization of *Monitor* requires proving that the constraints associated with *TempMonitor* imply the model-level constraints obtained by instantiating the parameters of the *Monitor* feature roles. In other words, the following proof obligations must be discharged:

Structural Property Proof Obligation :

$$cv < th \Rightarrow (cv \leq th)$$

Attach Behavior Proof Obligation :

$$\begin{aligned} & (TempSensor@pre \rightarrow excludes(s) \text{ and} \\ & \text{ready}@pre = 1 \text{ and} \\ & TempSensor = TempSensor@pre \rightarrow including(s)) \\ \Rightarrow & \\ & (TempSensor@pre \rightarrow excludes(s) \text{ and} \\ & TempSensor = TempSensor@pre \rightarrow including(s)) \end{aligned}$$

Detach Behavior Proof Obligation :

$$\begin{aligned} & (TempSensor@pre \rightarrow includes(s) \text{ and} \\ & \text{ready}@pre = 1 \text{ and} \\ & TempSensor = TempSensor@pre \rightarrow excluding(s)) \\ \Rightarrow & \\ & (TempSensor@pre \rightarrow includes(s) \text{ and} \\ & TempSensor = TempSensor@pre \rightarrow excluding(s)) \end{aligned}$$

2.1.2 Role Relationships

A role can be associated with another role, indicating that the realizations of the roles are associated in a manner that is consistent with how the bases of the roles are related in the UML metamodel. For example, a *Class* role can be directly associated with an *AssociationEnd* role, but not with an *Association* role because the UML metamodel does not directly associate the *Association* metamodel element with the *Class* metamodel element. Similarly two *Class* roles can be associated with *Generalization* role or *Abstraction* role. We use the UML form of association to represent role associations. Role associations can be named and can have multiplicities associated with their ends.

To explain role relationships, we use the *User* role from our CICO role model (CICO SRMs will be explained later in Section 3). The SRM shown in Fig. 2 contains two associations, *has-child* and *has-parent*, between the roles *UserGeneralization* and *User*. These associations indicate that realizations of the *User* role can be related to each other via generalizations (realizations of *UserGeneralization*). Fig. 2 also shows other two associations, *is-supplier* and *is-client*, between the roles *UserRealization* and *User*. These indicate that realizations of *User* can be related to each other via UML $\ll realize \gg$ relationships (realizations of *UserRealization*).

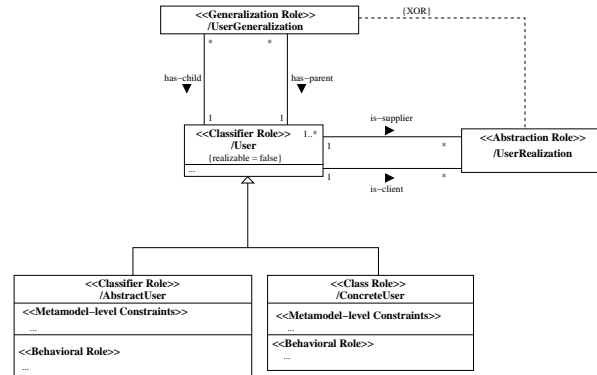


Figure 2: An Example of Role Relationships

Roles with a common set of characteristics can be generalized by a role, called a *role generalization* (or simply generalization when the context is clear), that consists only of the common characteristics. The roles that are generalized are called *role specializations* (or specializations when the context is clear). A role specialization inherits the role associations, the metamodel-level constraints and the feature roles defined in the generalization. Generalizations can be used to restructure a SRM by exploiting commonalities across roles. We use the UML generalization/specialization relationship between roles. A role specialization characterizes a subset of the realiza-

tions characterized by its parent role (currently, a specialization role is restricted to having only one parent role). This can be accomplished by further restricting the properties inherited from the parent role, for example, by restricting the multiplicities of inherited associations between roles, or by adding metamodel-level constraints that further constrain the form of realizations (for details see [10]).

Fig. 2 also illustrates the concept of role generalization. The *User* role (the generalization) captures properties that are common to *AbstractUser* and *ConcreteUser* roles (the specializations). The specializations inherit the associations *has-child*, *has-parent*, *is-supplier*, and *is-client*, as well as the metamodel-level constraints and feature roles (not shown) in the generalization *User*. The constraint $\{XOR\}$ between *UserGeneralization* and *UserRealization* indicates that a relationship between two *User* realizations should be a realization of either *UserGeneralization* or *UserRealization*, but not both. This is an example of a pre-defined constraint across roles.

All realizations of *User* are either realizations of *AbstractUser* or *ConcreteUser*. A role such as *User* is said to be *abstract*. Each role is associated with a tagged value that indicates whether it is abstract or not: the tagged value $\{realizable = false\}$ indicates that the role is abstract while $\{realizable = true\}$ indicates that the role is not abstract. In this paper, the $\{realizable = true\}$ tag is omitted if the role is not abstract.

2.1.3 Abbreviated SRMs

An abbreviated SRM allows one to gain an understanding of a family of designs at a particular level of abstraction without being distracted by detailed information. We illustrate abbreviations for the *AssociationEnd* role and the *Generalization* role structure.

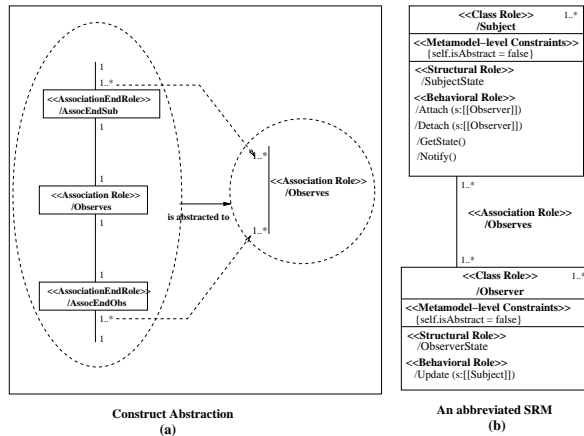


Figure 3: An Example of Abbreviated Association in SRM

An abstraction of the *AssociationEnd* role is illustrated

in Fig. 3(b) which shows the abbreviated SRM for the simple Observer pattern. Details related to the properties of the association ends and the behavioral properties of the class roles are omitted in Fig. 3(b). The detailed association role structure is shown in Fig. 3(a). The association abstraction is possible because each association end role realization is associated with exactly one *Subject* or *Observer* realization and the *Observes* role has a one-to-one relationship with each of its association end roles.

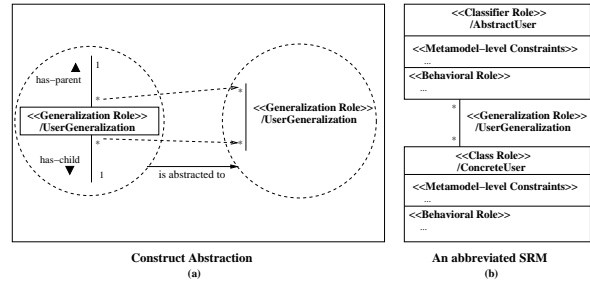


Figure 4: An Example of Abbreviated Generalization in SRM

An abstraction of the *generalization* role structure is illustrated in Fig. 4(b) which shows the abbreviated SRM for the User part of CICO system. The details of the generalization can be seen in Fig. 4(a). We can abbreviate a *realization* role in the same manner.

2.1.4 Role Hierarchies

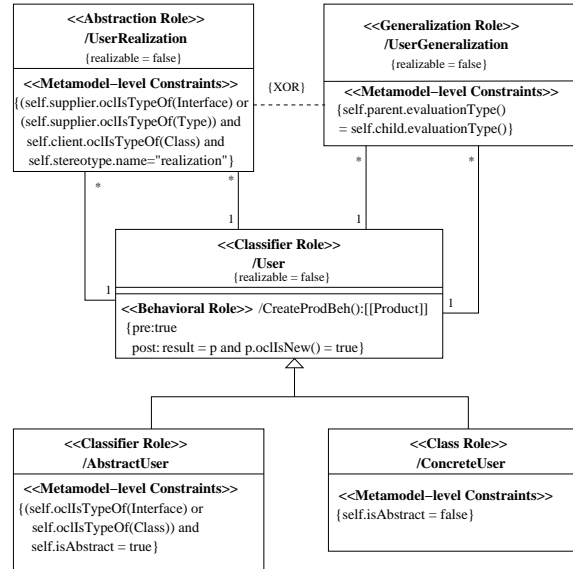


Figure 5: A Role Hierarchy

SRMs often contain recurring structures which can be viewed as a pattern. An example of such a structure is

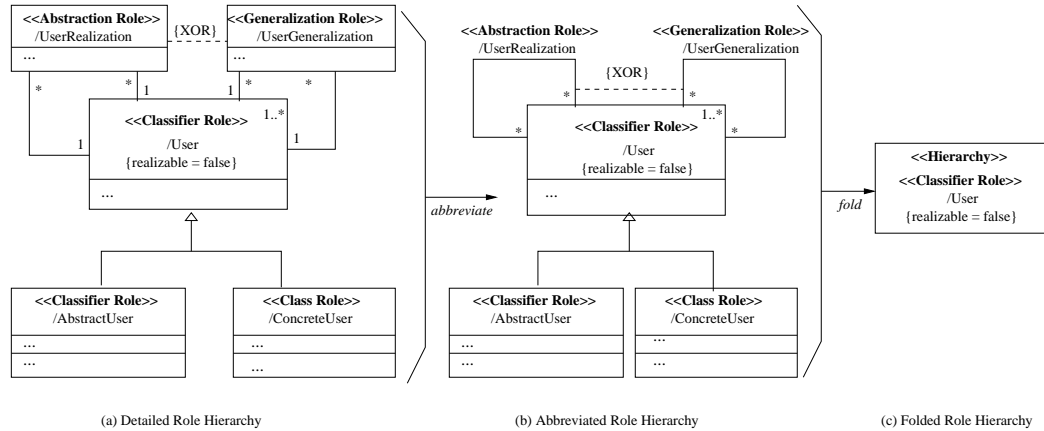


Figure 6: A Role Hierarchy

the hierarchy associated with an abstract role. Fig. 5 shows an abstract role called *User* whose specializations are either *AbstractUser* or *ConcreteUser*. Since the *User* role is abstract, its realizations must be either realizations of *AbstractUser* or *ConcreteUser*, which are both specializations of the *User* role.

The *AbstractUser* role characterizes *AbstractUser* realizations which are interface constructs in the form of UML interfaces or abstract classes. The *ConcreteUser* characterizes realizations in the form of concrete user classes. Realizations of the *User* specialization roles can be connected either by a realization of the *UserRealization* role (a UML *realize* relationship), or a realization of the *UserGeneralization* role (a UML generalization relationship).

The notion of hierarchy arises because:

1. a realization of *ConcreteUser* may *realize* a realization of *AbstractUser* (a realization process),
2. a realization of *ConcreteUser* may *specialize* another realization of *ConcreteUser* (a generalization/specialization process),
3. a realization of *ConcreteUser* may *specialize* a realization of *AbstractUser* if both realizations are classes (a generalization/specialization process), or
4. a realization of *AbstractUser* may *specialize* another realization of *AbstractUser* if both realizations are interfaces (a generalization/specialization process).
5. There may be further specializations of either *AbstractUser* or *ConcreteUser* resulting in a deeper hierarchy.

In addition to hierarchical realizations, the role structure also permits non-hierarchical realizations because the role association end multiplicities at the *UserRealization* and *UserGeneralization* ends are “*” (zero or more). The metamodel-level constraints in the *User-*

Realization role indicate that for a *realize* relationship, the supplier must be an interface or a class that can have operations (but no methods), while the client must be a class. In a generalization relationship, the superclass and subclass must both be interfaces or both be classes.

A recurring role hierarchy structure can be shown in a further abstracted form, called a *folded SRM*, in order to help an understanding of a family of designs without being distracted by detailed information. Fig. 6 shows how Fig. 5 can be abstracted to a *folded SRM*. Fig. 6(a) is the same SRM as Fig. 5 except that all metamodel-level constraints and feature roles are not shown. This SRM can be abstracted to Fig. 6(b) by abstractions of generalization and realization. This can be further abstracted by a stereotype structure *Hierarchy* for the *role hierarchy*. This results in a *folded SRM* as shown in Fig. 6(c). All metamodel-level constraints and feature roles of role specializations and their relationships are hidden in a folded SRM. These are revealed when a folded SRM is *unfolded*.

2.1.5 SRM for the Simplified Observer Pattern

We illustrate the use of SRMs to specify the Observer design pattern [12]. A SRM for the simplified pattern is shown in Fig. 7. The SRM consists of two class roles, *Observer* and *Subject*, an association role, *Observes*, and two association end roles, *AssocEndObs* and *AssocEndSub*. The *Observer* role characterizes classes of objects (observers) that monitor other objects called subjects, and the *Subject* role characterizes subject classes.

The association between subject and observer classes in a realization of the Observer pattern is captured in the pattern by *Observes*. The forms of the association ends of a realization of *Observes* are constrained by the two association end roles shown. *AssocEndSub* states that the subject end of an *Observes* realization must have a multiplicity of 0 or more (*), while *AssocEndObs* states

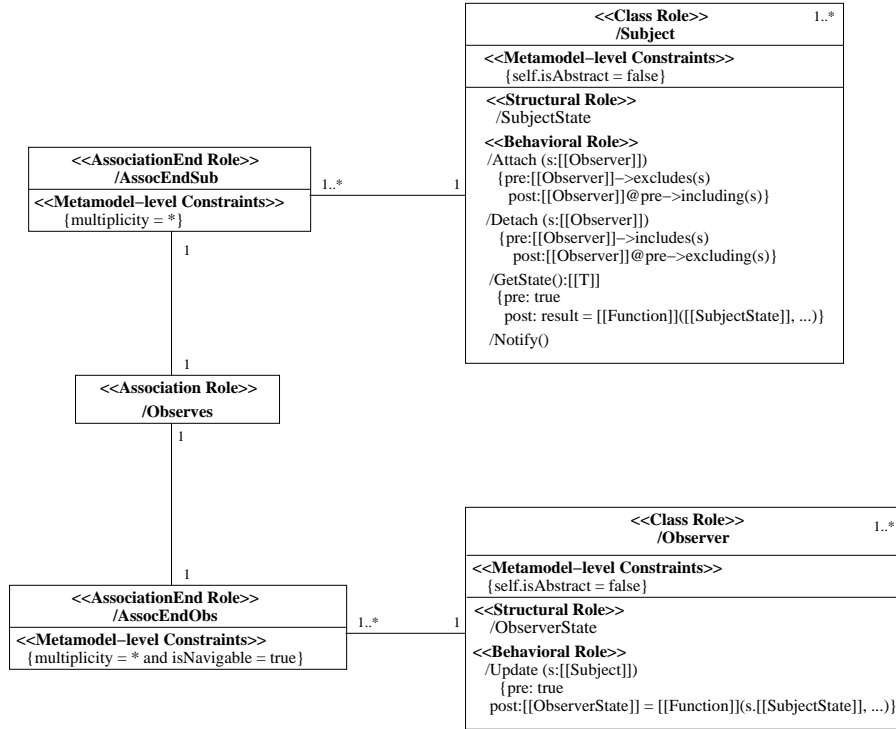


Figure 7: SRM for the Observer Design Pattern

that the observer end must be navigable and must have a multiplicity of 0 or more. The SRM thus characterizes models in which subject objects can attach themselves to one or more observer objects, and an observer object can monitor one or more subject objects.

The multiplicities on the association between the *Subject* role and the *AssocEndSub* role indicate that a realization of the *Subject* role (a class) must have at least one association end that realizes *AssocEndSub*, and that a realization of *AssocEndSub* must be associated with exactly one class that realizes the *Subject* role. The other role associations shown in the SRM are interpreted in a similar manner.

The realization multiplicities shown in the SRM require that a realization of the SRM must have at least one realization of *Subject* and at least one realization of *Observer*, and each realization of *Subject* is associated with at least one *Observer*, and each *Observer* realization is associated with at least one *Subject* realization via the realizations of *Observes*. The smallest models that can realize this SRM consist of a realization of *Subject* and a realization of *Observer* connected by a realization of *Observes*. The “1..*” realization multiplicity for *Observes* (not shown) can be inferred from the realization multiplicities of *Subject* and *Observer* and the multiplicities on the associations between the roles.

Realizations of the *Observer* role must (1) be concrete classes (*self.isAbstract = false*), (2) have at least one realization of *ObserverState*, and (3) have behaviors that realize *Update*.

The model-level post-condition obtained from *Update* requires that the realization of the *ObserverState* be updated with a value that is a defined function of a subject state (a realization of *SubjectState*) and possibly other arguments. The *[[Function]]* parameter must be instantiated with the name of a function that defines how the subject state is used to calculate a new value for the observer state. The bracketed expression following *[[Function]]* shows the parameters that must be passed into the function (the “...” indicates that other parameters not explicitly listed may be present).

Realizations of the *Subject* role must (1) be concrete classes (*self.isAbstract = false*), (2) have at least one realization of *SubjectState*, and (3) have behaviors that realize *Attach*, *Detach*, *GetState*, and *Notify*.

The behavioral model-level constraints that must be satisfied by realizations of *Subject* are obtained by instantiating the parameters of the SRM role’s behavioral role (i.e., *Attach*, *Detach*, *GetState*, *Notify*). A realization of the *Attach* behavior attaches an observer to the subject, and a realization of the *Detach* behavior removes an observer from the subject. A realiza-

tion of *GetState* returns a value (indicated by *result*) of type indicated by the parameter *T*, that is a function of the current value of the state realizing *SubjectState*, and possibly other values. There are no pre- and post-conditions on *Notify* (i.e., there are no restrictions on the effect this operation has on the state of a subject).

A model is said to *realize a SRM* if the following holds:

1. The model constructs that are intended to realize SRM roles do realize the roles. This means that (1) the model elements satisfy the metamodel-level constraints of the role, and (2) the constraints expressed in the realizing model (e.g., pre- and post-conditions for operations, and constraints on attributes) imply the model-level constraints obtained by instantiating the parameters of the feature roles.
2. The model conforms to constraints expressed across roles or their realizations (e.g., realization multiplicities, role association multiplicities).

In this paper, a stereotype with a role name in a model construct is used to indicate that the model construct is an intended realization of the role (e.g., **<< Subject >>**). These stereotypes are printed in bold to distinguish them from other UML- and user-defined stereotypes.

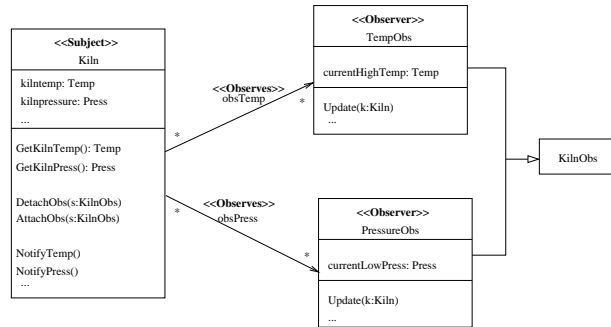


Figure 8: A Realization of the Simplified Observer Pattern

A realization of the simplified Observer pattern is shown in Fig. 8. The class *Kiln* is a realization of the *Subject* role, while *TempObs* and *PressureObs* are realizations of the *Observer* role, as indicated by the stereotypes. The attributes *kilntemp* and *kilnpressure* each play the role of *SubjectState*. *GetKilnTemp* returns the current value of *kilntemp*, and *GetKilnPress* returns the current value of *kilnpressure*. These operations each realize the *GetState* behavior.

The *DetachObs* and *AttachObs* operations are intended to realize the *Attach* and *Detach* feature roles, respectively. The behaviors specified by the model-level constraints obtained by appropriately instantiating the parameters of the *Attach* and *Detach* constraint templates

are implied when *TempObs* or *PressureObs* objects are passed as parameters to both *DetachObs* and *AttachObs*.

In *TempObs*, the *currentHighTemp* attribute realizes *ObserverState* in the *Observer* role. The *Update* operation compares *k.kilntemp* to *currentHighTemp* and assigns *k.kilntemp* to *currentHighTemp* if the *k.kilntemp* value is greater, else *currentHighTemp* is left unchanged. This behavior implies the following model-level constraint obtained by appropriately instantiating the parameters of the *Update* behavioral role in the *Observer* role:

```
{pre: true
post: let F(t:Temp, c:Temp): Temp =
    if c >= t then c
    else t
    endif
in
currentHighTemp = F(k.kilntemp, currentHighTemp)}
```

2.1.6 SRMs at the UML Metamodel level

Fig. 9 shows how SRMs relate to the UML metamodel. The realizations of the SRM shown in the figure are Class Diagrams. The two class diagrams shown at level M1 are realizations of the SRM. Each role in the SRM determines a subset of UML model constructs, for example, the association role *Assc1* in the SRM shown at level M2 determines a subset of UML association constructs (i.e., it determines a specialization of the metamodel class *Association*). At the model level (M1), *ClassB* and *ClassC* are both realizations of the *R1* SRM role.

Fig. 9 also illustrates the difference between our roles and UML collaboration role: The UML classifier role *RoleA* (at level M1) is a view of *ClassA*, that is, *RoleA* consists of a subset of *ClassA* properties (attributes, operations, associations). Objects of *ClassA* (at level M0) can play the role of *RoleA*.

2.2 Interaction Role Models (IRMs)

Pre- and post-condition templates expressed in a SRM constrain the effects of behaviors. Not all behavioral properties can be expressed in terms of pre- and post-conditions in a SRM, for example, one cannot use pre- and post-condition constraint templates to constrain how objects of realizations interact when performing a particular behavior. In this subsection we describe a type of Role Model, called an *Interaction Role Model* (IRM) that can be used to constrain how objects of realizations interact when carrying out a realization of a behavioral role.

A collaboration-styled IRM for the simplified Observer pattern is shown in Fig. 10(a). The IRM describes a pattern of interaction in which the (realization of the) *Notify* behavior involves invoking the *Update* behavior in each of its observers. During the execution of the *Update* behavior, an observer invokes the *GetState* behavior of the subject. The variable *st* represents the

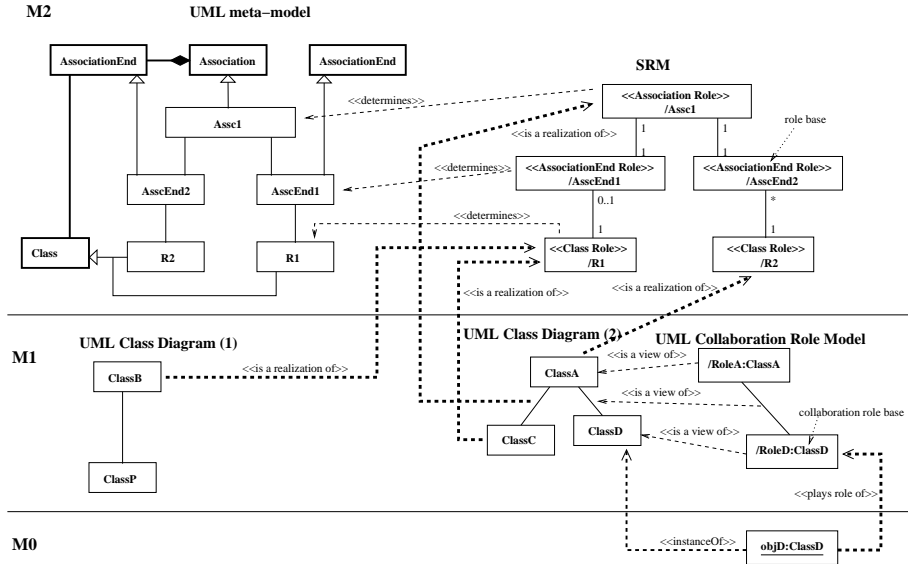


Figure 9: Roles at the Metamodel Level

subject state that is returned as a result of the *GetState* behavior.

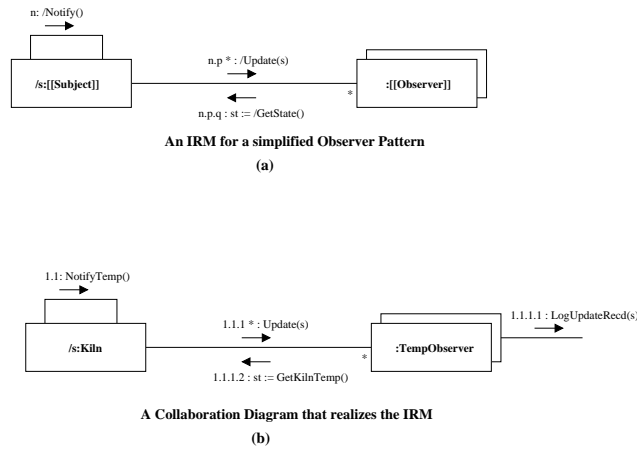


Figure 10: IRM for the simplified Observer pattern

The rectangular boxes in Fig. 10(a) are *collaboration role templates*. A collaboration role template determines a set of UML classifier roles (a UML classifier role is a projection of a UML class). Instantiating the parameters of a collaboration role template results in a UML classifier role (i.e., the realizations of a collaboration role template are UML classifier roles). The classifier roles obtained by instantiating collaboration role templates are projections of the realizations of the SRM roles indicated in the templates. For example, by substituting and *Kiln* for *Subject* in */s : [[Subject]]* we get the UML classifier role */s : Kiln* shown in Fig. 10(b).

The */s* represents a UML role named *s*; it does not have the same notion of roles in our Role Model (e.g. */Subject* in the SRM shown in Fig. 7).

In Fig. 10(a), the box with *[[Observer]]* is a collaboration role template to be instantiated into an unnamed UML role whose base classifier is a realization of the *Observer* role. Fig. 10(b) shows the realization in the form of an unnamed UML role with the base classifier *TempObserver* (the box with *:TempObserver*). In Fig. 10(a), the box with */s : [[Subject]]* is a collaboration role template to be instantiated into a UML role named *s* whose base classifier is a realization of the *Subject* role. Fig. 10(b) shows the realization with a UML role named *s* with the base classifier *Kiln* (box with */s : Kiln*).

Just like in a SRM, we place a “/” in front of the behavioral feature role names (e.g., */Notify*) to indicate they are *message roles* to be realized by actual messages (specifications of stimuli) in an IRM realization. In this paper we restrict our attention to message realizations that are specifications of operation calls, where the operations are realizations of the behavioral feature roles that are named in the message roles.

The sequence labels in an IRM are generic, for example, the expression *n : /Notify()* indicates that the *Notify* stimulus in a realization is the *n*th interaction in the sequence, where a specific value for *n* must be given in a realization. Similarly, the expression *n.p.q : st := /GetState()* indicates that a message realization of *GetState* is the *q*th nested interaction of the *p*th nested interaction of the *n*th outermost interaction. The IRM allows other interactions to occur between the */Update*

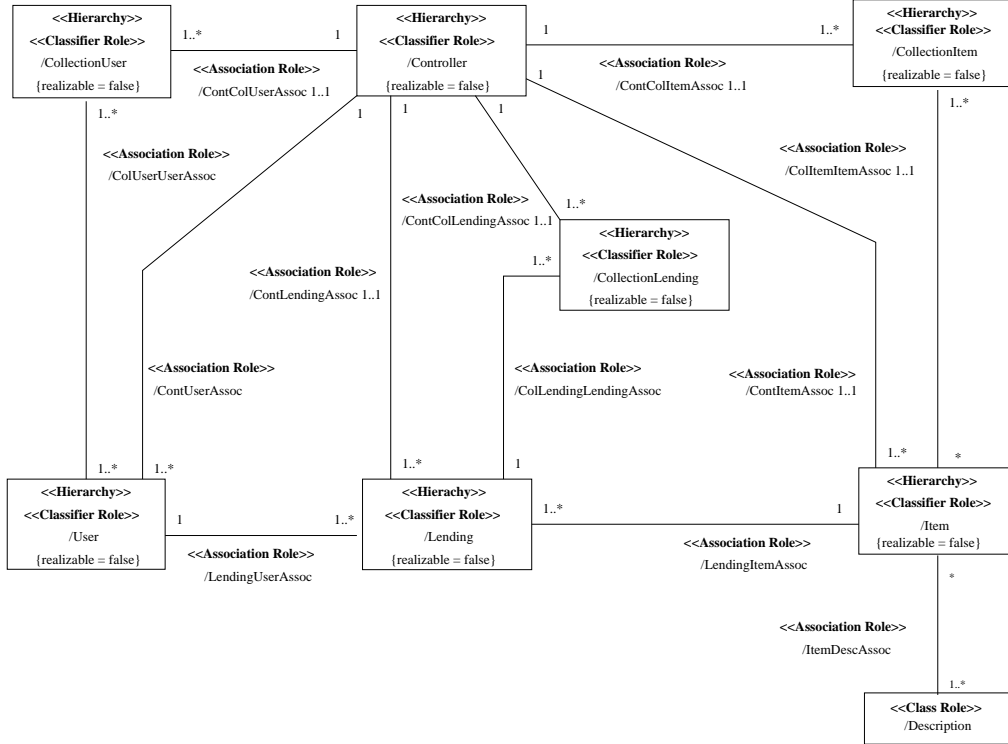


Figure 11: A Folded SRM of CICO System

and the `/GetState` interactions as indicated by the sequence expressions. In the realization of the IRM shown in Fig. 10(b), $n = 1.1$, $p = 1$ and $q = 2$. In this IRM realization, the receipt of the `Update` stimulus is logged before the observer obtains the state of the `Kiln`. There is an interaction (`LogUpdateRecd`) that is not shown in the IRM (i.e., it is not part of the behavior characterized by the pattern). The sequence labels indicate that the `LogUpdateRecd` interaction occurs before the `GetKilnTemp` interaction.

3. ROLE MODEL FOR CICO SYSTEMS

In this section, we describe the CICO model family using Role Models. We use one SRM and two IRMs, and they are intended to capture minimal properties of the CICO pattern rather than to reflect a “complete” or “correct” characterization. The CICO model family is for applications that require the checkin-checkout type of functionality. Such applications can be found in libraries, movie rental stores and car rental agencies. We find the following common characteristics in these applications:

1. Items in the application domain are assumed to be unique, although several items may have the same description.
2. Items are maintained in a collection. There may be one or more collections.
3. There is a list of authorized users.

4. A user can check out an item (we call this *lending* in our paper) if it is available.
5. A checked out item can be checked in. In our examples, we assume that an item can be checked in only if it was previously checked out. However, in some applications, an item does not need to be checked out before it can be checked in. For example, in a software version control system, the initial version is always checked in first before it can be checked out. Such applications are beyond the scope of this paper.
6. Every time an item is checked out, a record of the lending is made with some associated information. This information is needed when the item is checked back in.

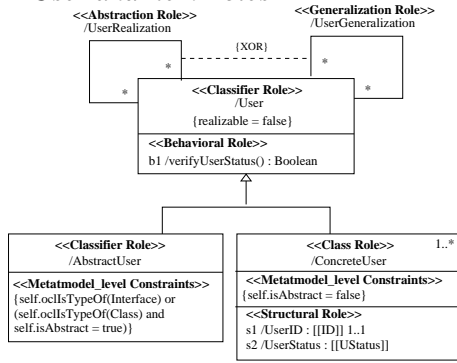
3.1 CICO SRM

Fig. 11 shows the SRM of the CICO model family. We make use of the folded `<< Hierarchy >>` stereotype for the `CollectionUser`, `User`, `CollectionLending`, `Lending`, `CollectionItem`, `Item`, and `Controller` roles. We also have a `Class` role `Description` because only classes can realize this role. Details of these roles are shown in the unfolded form in the following subsections.

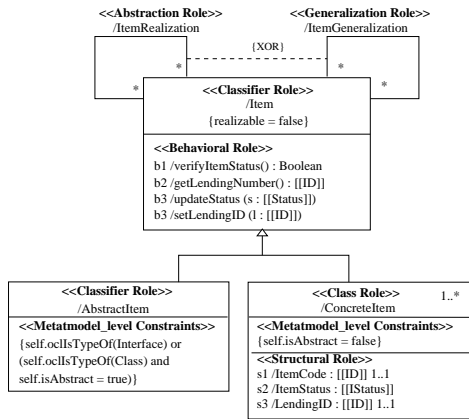
In a folded SRM, one may impose constraints on association roles to restrict the number of realizations. For example, in Fig. 11, the multiplicity “1..1” on the

ColItemItemAssoc association role restricts the number of realizations to only one. SRMs allow us to restrict the multiplicities on association ends when realizing models [10] (not shown in this paper). In this section, we omit showing the multiplicity when it is “*” for classifier and relationship roles, and “1..*” for structural and behavioral roles, these being common for the CICO SRM.

3.1.1 User and Item Roles



(a) User Role Hierarchy



(b) Item Role Hierarchy

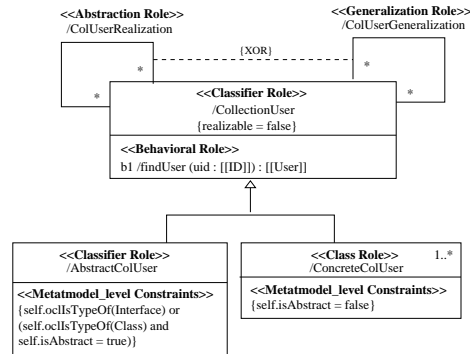
Figure 12: Detailed User and Item Role Hierarchies

Fig. 12 shows the details of *User* and *Item* roles. Realizations of the *User* role must have one or more realizations of the *verifyUserStatus* behavioral role indexed by “b1”, an identifier, where “b” indicates behavioral roles. This role is needed to verify whether or not the user has an appropriate status for performing checkin or checkout. The behavioral role can be realized by a single operation or method, or by a composition of operations or methods. It is also inherited by the *AbstractUser* and *ConcreteUser* specializations. The *AbstractUser* must be realized either by interfaces or by abstract classes. The *ConcreteUser* must be real-

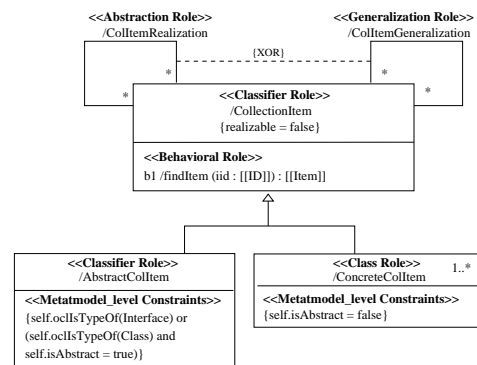
ized by classes. The structural roles are indexed by “s#” (i.e., s1, s2, ...) where “s” indicates structural roles. Realizations of *ConcreteUser* must possess only one *UserID* realization (indicated by “1..1”) and one or more *UserStatus* realizations. The status of the user indicates whether or not the user is eligible to checkin or checkout an item. Details of the *UserID* and *UserStatus* are described later in Section 3.1.4.

The *Item* role can be interpreted in a similar manner. The *updateStatus* role is required to enable updating the status of an item to a new one. For example, whenever an item is checked out (or checked in), the status of the item needs to be changed to `[[CHECKEDOUT]]` (or `[[AVAILABLE]]`). Below, we show the OCL constraints for the *updateStatus* role.

```
context[[Item]] :: /updateStatus (s : [[Status]])
pre : True
post:
  -- Update the status of [[ItemStatus]] to s
  [[ItemStatus]] = s
```



(a) CollectionUser Role Hierarchy



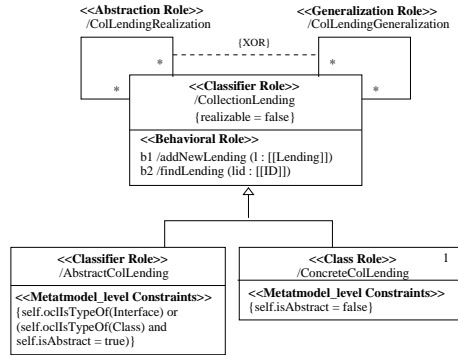
(b) CollectionItem Role Hierarchy

Figure 13: Detailed CollectionUser and CollectionItem Role Hierarchies

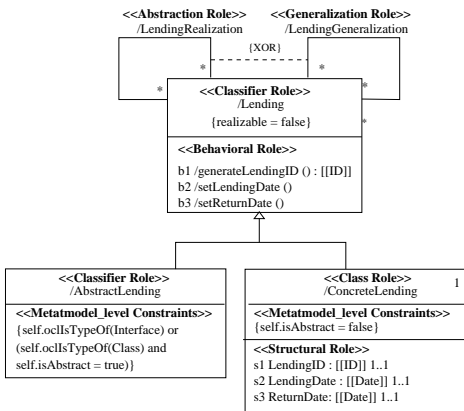
3.1.2 CollectionUser and CollectionItem Roles

Fig. 13 shows the details of *CollectionUser* and *CollectionItem* roles. The *CollectionUser* role is required to maintain a collection of users, and it includes a behavioral role *findUser* which is required to locate a user given the user's `[[ID]]`. Similarly, the *CollectionItem* role is required to maintain a collection of items, and it includes a behavioral role *findItem* to locate an item given the item's `[[ID]]`.

3.1.3 CollectionLending and Lending Roles



(a) CollectionLending Role Hierarchy

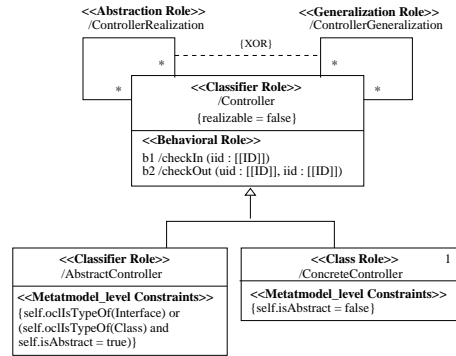


(b) Lending Role Hierarchy

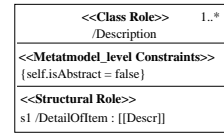
Figure 14: Detailed CollectionLending and Lending Role Hierarchies

Fig. 14 shows *CollectionLending* and *Lending* roles in detail. The *CollectionLending* role is a collection for maintaining all the *Lending* information. The *CollectionLending* role has two behavioral roles, (1) to add new lending information (*addNewLending*), and (2) to find lending information based on some *ID*. There can be only one realization of the *ConcreteColLending* role (as indicated by the “1” in the first compartment of the *ConcreteColLending* role description). The *ConcreteLending* role has three structural roles for the *ID* of

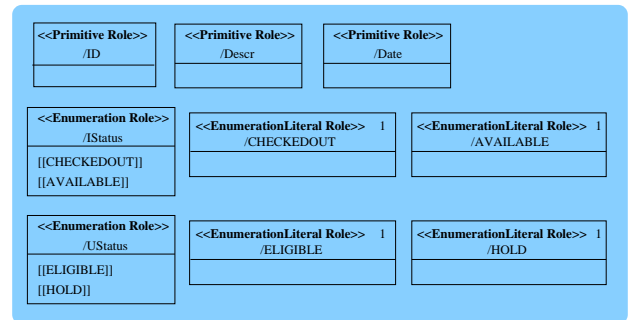
the lending (*LendingID*), and the dates of the lending (*LendingDate*) and return (*ReturnDate*).



(a) Controller Role Hierarchy



(b) Description Role



(c) Data Type Roles

Figure 15: Detailed Controller, Description and Data Type Roles

A *Lending* role is required to maintain information about a particular checkin or checkout scenario. The *Lending* role provides behavioral roles for initializing the lending information (*InitializeLending*), generating a new lending information *ID* (*generateLendingID*), and obtaining the dates for lending (*getLendingDate*) and return (*getReturnDate*). OCL constraints on the *InitializeLending* behavioral roles in the figure are as follows:

```
context[[Lending]] :: /generateLendingID () : [[ID]]
pre : True
post :
-- The generated ID is greater than the
-- number of the instances by one
result = allInstances → size() + 1
```

3.1.4 Controller, Description and Data Type Roles

Fig. 15 depicts the *Controller*, *Description* and *Data Type* roles. Fig. 15(a) shows that there must be only one realization of *ConcreteController* role. OCL constraints on the behavioral roles for *ConcreteController* are as follows:

```

context[[Controller]] :: /checkIn (iid : [[ID]])
pre :
  - - iid is a valid item code
  [[Item]] → exists (i | i.[[ItemCode]] = iid)
post:
  - - Item is checked in
  [[ItemStatus]] = [[AVAILABLE]]

context[[Controller]] ::
/checkOut (uid : [[ID]], iid : [[ID]])
pre :
  - - uid is a valid user id
  - - iid is a valid item code
  [[User]] → exists (u | u.[[UserID]] = uid)
  [[item]] → exists (i | i.[[ItemCode]] = iid)
post:
  - - Item is checked out
  [[ItemStatus]] = [[CHECKEDOUT]]

```

The *Description* role is described in Fig. 15(b). It shows that the *Description* role must contain one or more realizations of the *DetailOfItem* role which is a primitive role described below.

The shaded box in Fig. 15(c) defines data type roles being used in the SRM. There are three *primitive* roles *ID*, *Descr* and *Date* for the identity of a user or item, description and date. There are a couple of *Enumeration* roles, such as *IStatus* (Item status) and *UStatus* (User status). These contain the *EnumerationLiteral* roles, such as *CHECKEDOUT*, *AVAILABLE*, *ELIGIBLE* and *HOLD*. The first two describe the status of the item and the last two describe the status of the user.

3.2 CICO IRMs

We developed two IRMs to illustrate the scenarios for checkOut and checkIn respectively. The IRMs are expressed in the style of Sequence Diagrams.

Fig. 16 shows an IRM for a checkOut scenario. The *checkOut* behavior requires the *IDs* of the user and the item. The instance of a realization of the *Controller* invokes the *findUser* behavior with the user *ID* to obtain the user “u” with a matching *ID* from the instance of the class playing the *ConcreteColUser* role. The status of this user is verified and if allowable, the appropriate item is found using the item *ID*. The status of the item is verified. If the status is appropriate, a new instance of the class playing the *Lending* role is created and a new *ID* is generated accordingly followed by setting the date of lending. This instance is added to the Lending Collection (*ConcreteColLend*). The status of

the item is updated to indicate that it is checked out. The *LendingID* in the item is set to the *ID* generated for the lending information. For lack of space, we do not show the alternative courses of events in case a user or item does not have the correct status.

Fig. 17 shows an IRM for a checkIn scenario. The *checkIn* behavior requires the use of the item’s *ID*. The item is first located in the item collection and its status verified (for instance, to verify that it was indeed checked out). If the item has the appropriate status, the *LendingID* is obtained from the item and is used to locate the lending information in the collection. The return date is set in the lending information.

4. REALIZATIONS OF THE CICO ROLE MODEL

Role Models can be realized to obtain design models for specific applications. Designers have the flexibility to add domain-specific model constructs or customized properties unless the Role Model explicitly restricts them. In this section, we demonstrate two different realizations of the CICO Role Model. They are the library model and the car rental model. We use bold stereotypes in the model constructs to indicate that the constructs are intended to be realizations of the roles named in the stereotypes. Feature properties that are not bold are additional features not specified in the Role Model. We describe the two models by realizing the SRM and IRMs for the CICO system.

4.1 Library Model

The library has a collection of items. An item, called copy, can be a book or multimedia item. Members of the library can check in or check out items. They are also allowed to reserve items. Fig. 18 shows the class diagram of the library as a realization of the CICO SRM. Table 1 shows the mapping.

Table 1: Mapping of CICO SRM with Library Model.

SRM Role	Class
<i>Controller</i>	<i>Controller</i>
<i>CollectionUser</i>	<i>CollectionMember</i>
<i>User</i>	<i>Member</i>
<i>CollectionLending</i>	<i>CollectionLoanInfo</i>
<i>Lending</i>	<i>LoanInfo</i>
<i>CollectionItem</i>	<i>CollectionCopy</i>
<i>Item</i>	<i>Copy, Multimedia, Book</i>
<i>Description</i>	<i>Description</i>

We describe a few important aspects of the CICO SRM realization below:

- The *hierarchy* is illustrated in the realization of the *Item* Role by *Copy* which is specialized by

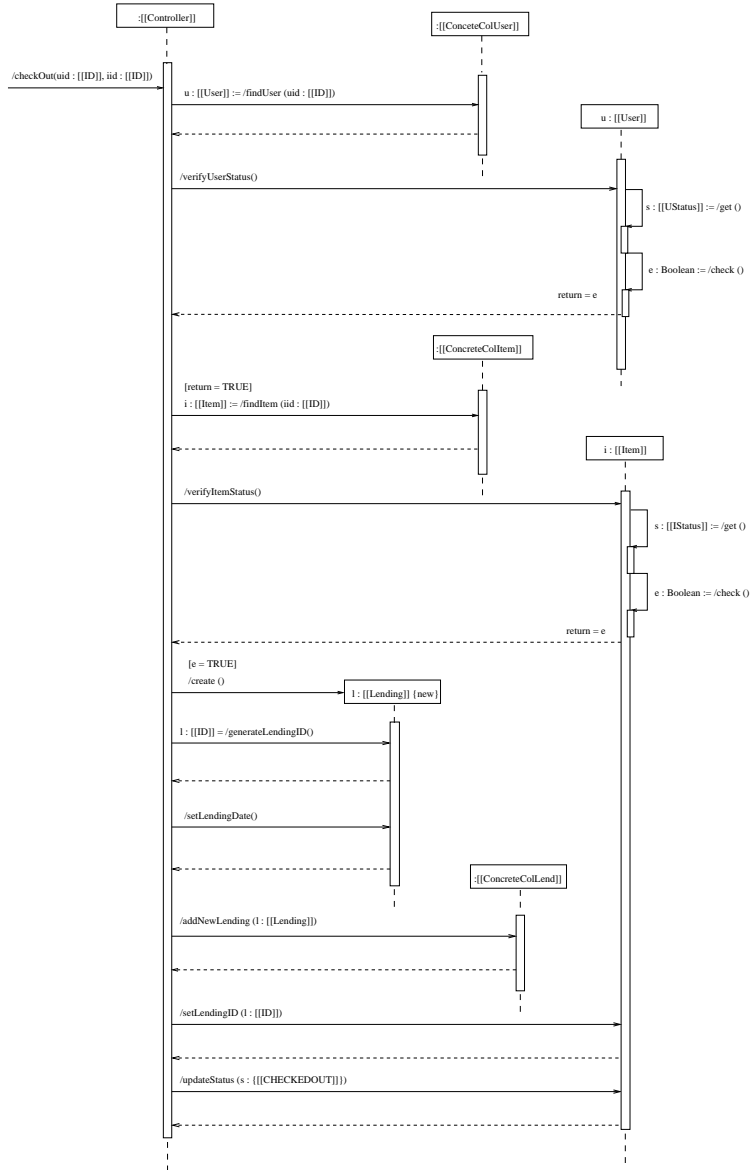


Figure 16: An IRM for a CheckOut Scenario

Multimedia and *Book*. The realization is valid because of the property of the *Item* role hierarchy in Fig. 12.

- *Multimedia* is a realization of the *ConcreteItem* role and this is indicated by the stereotype $\ll \text{ConcreteItem} \gg$. The notation $\ll s1 \gg$ indicates that *CopyID* is a realization of the structural role *ItemCode*, and is the only realization because of the restriction in the multiplicity (“1..1” in Fig. 12).
- The *Book* and *Multimedia* both play the role of *ConcreteItem*. However, the *verifyItemStatus*

behavioral role in *Book* is played by both *verifyHoldStatus* and *verifyBorrowStatus* operations (indicated by the $\ll b1 \gg$ next to both operations), but only by *verifyHoldStatus* in *MultiMedia*. This can be considered a customization involving domain information.

- The $\ll b3 \gg$ next to the *setStatus* operation denotes that it is a realization of the *updateItemStatus* behavioral role.
- The “has” association in the model is the only realization of the association role between the *CollectionCopy* and the *Copy* in the SRM. This re-

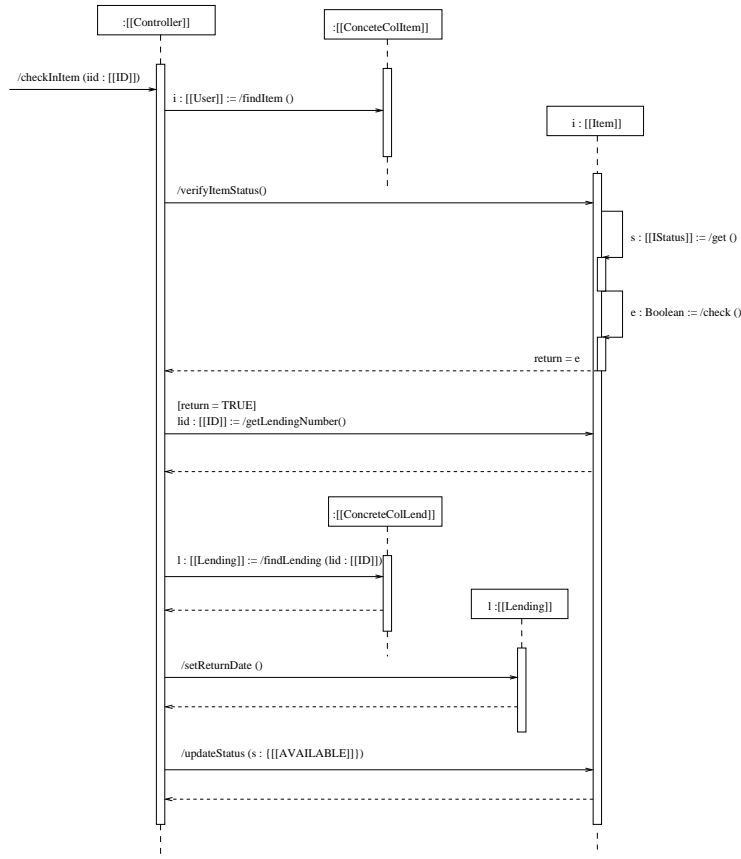


Figure 17: An IRM for a CheckIn Scenario

striction was expressed by “1..1” in Fig. 11.

- There are two associations between *LoanInfo* and *Copy* (*iscurrentlyloaned* and *isloanedpast*), and also between *LoanInfo* and *Member* (*currentlyhas* and *haspast*). This illustrates that the association roles *LendingItemAssoc* and *LendingUserAssoc* can be realized with two associations each.
- The “1” shown in the *ConcreteLending* role in Fig. 14 indicates that there must be only one realization of *Lending*. In the library model the *LoanInfo* is the sole realization of the *Lending* role.
- The status types *CHO* and *AVAL* are realizations of the enumeration role literals *CHECKEDOUT* and *AVAILABLE*.
- Although it may appear that *Member* and *Description* are both realizing single roles, actually, *Member* is realizing a role hierarchy which *Description* is merely realizing a *Class* role.

The following customizations have been made to the

CICO SRM:

- *Introduction of the capability to reserve items in the library:* In Fig. 18 we see that the class *Reservation* was added even though it was not required by the CICO SRM. Adding the *Reservation* class allows members to reserve copies.
- *Extra attribute features and operations:* The attributes *Name* and *Address* in *Member* are not required by the *User* role. The operation *removeLoan* in *LoanInfo* is not required by the *Lending* role.
- *Extra parameters in operations that realize behavioral roles:* The *ready* parameter passed in the *setStatus* operation in the *Multimedia* construct is in addition to the parameters required to realize the behavioral role *updateItemStatus*. Another example is the *RSV* (reserved) status that was added to the initial enumeration of different kinds of status types.

Figures 19 and 20 show realizations of the CICO IRMs for checkIn and checkOut respectively. Because of space

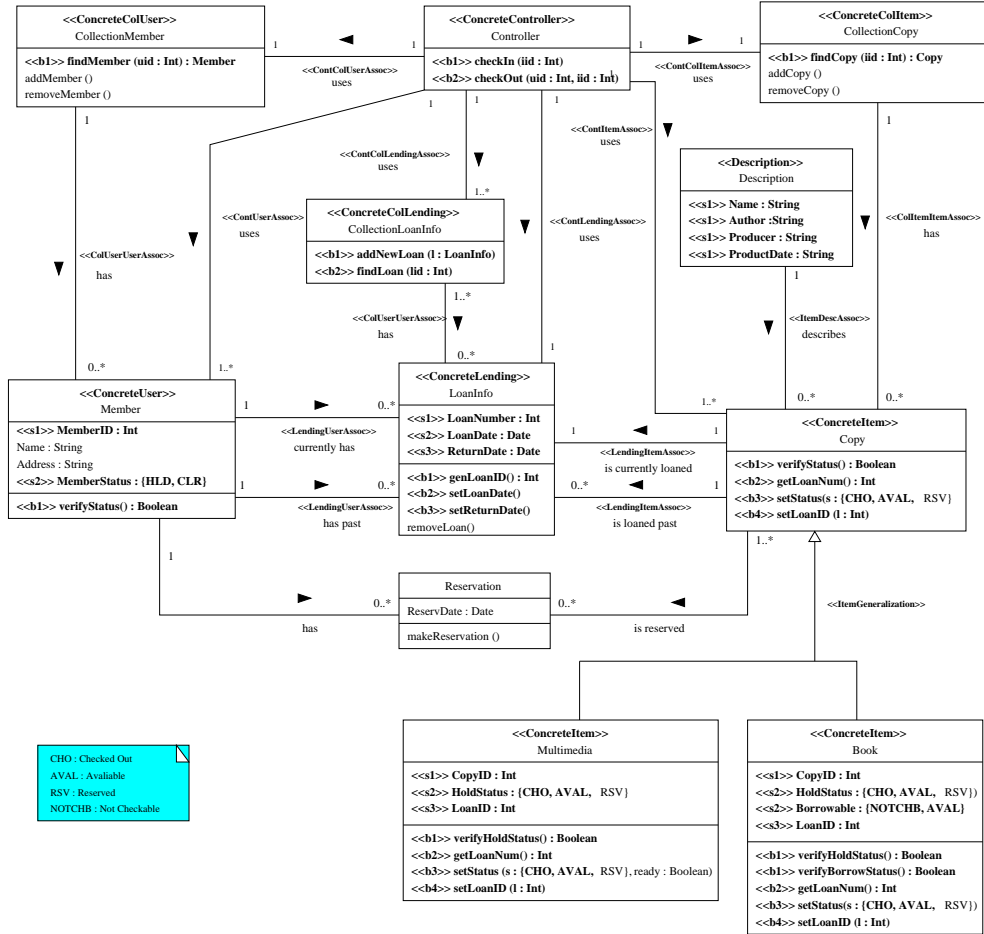


Figure 18: A Realization of the CICO SRM: Library Model

limitations, we do not show additional domain-specific behavioral specifications. For example, we do not show the capability to make reservations.

Even though both *verifyHoldStatus* and *verifyBorrowStatus* play the *verifyItemStatus* behavioral role for Book, they need not both be expressed in the IRM, but at least one needs to be present. The actual realizations depend on the requirements in the scenario. For example, we do not need to verify the “borrowable” status of a book while checking in the book, but do need to verify the hold status of the book. Therefore, in Fig. 19 only *verifyHoldStatus* is used, but in Fig. 20, both *verifyHoldStatus* and *verifyBorrowStatus* are used.

4.2 Car Rental Model

The car rental model is another realization of the CICO Role Model. This model has been designed for use in a car rental application where customers can rent vehicles. There are two types of vehicle collections, one for the *Truck* kind (called *CollectionTruck*), and the other for passenger vehicles (called *CollectionSedan*).

These collections realize the *CollectionCar* which is a UML interface. A *Car* is specialized by *Truck*, *Van* and *Sedan*. The *CollectionTruck* has the *Trucks* and the *CollectionSedan* has the *Van* and *Sedan*. Each *Car* has zero or more insurance policies associated with it. Customers are allowed to make reservations.

The car rental model and library models share several similar aspects of realization. However, there are some aspects not seen in the library model. These are as follows:

- The *CollectionItem* hierarchy is realized using a realize relationship between (1) *CollectionCar* and *CollectionTruck*, and (2) *CollectionCar* and *CollectionSedans*. Both *CollectionTruck* and the *CollectionSedan* realize the *ConcreteColItem*. The realization is valid according to the CICO SRM because the *ConcreteColItem* has “1..*” realization multiplicity. Being the default realization multiplicity, it is not shown in Fig. 12.

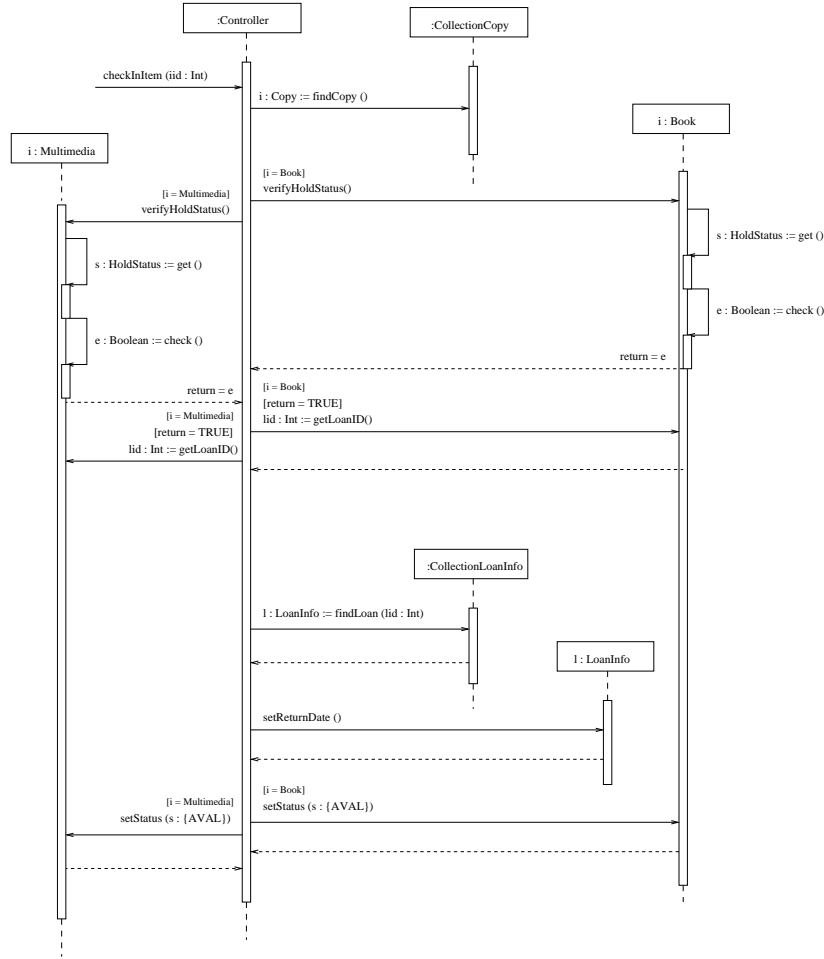


Figure 19: A Realization of the Library CheckIn IRM

- The *CollectionSedan* has two associations with *Van* and *Sedan* respectively. This realization satisfies the constraint “1..1” on the *ColItemItem- Assoc* role in Fig. 11.

5. RELATED WORK

The work on patterns (e.g., see [12, 19]) and architectures (e.g., see [3, 6, 24]) holds much promise, but very little attention has been paid to incorporating such experiences into system development environments (i.e., into notations, tools, and techniques used to develop systems). The *Catalysis* method [7] is notable in that it attempts to incorporate reusable experiences into modeling notation by providing syntactic support for representing architectural, design, and refinement patterns, called *frameworks*, in UML models. A problem with the *Catalysis* method is that the semantic aspects of these structures are described informally.

Work on domain-specific languages [27] focus on providing language interfaces for assembling code components

into programs. These languages focus on downstream development phases (detailed design and implementation in code) and are more appropriately called domain-specific programming languages (DSLs). Other forms of reusable experiences packaged for vertical reuse are frameworks [23] and domain-specific architectures (e.g., see [9, 13, 17, 26]). There is a considerable body of work on domain engineering processes and domain modeling notations (e.g. see [1, 13, 15, 18, 26]).

Pattern languages (e.g. see [4, 5]) have been developed to describe Business Resource Management that covers applications including patterns for resource rental, trading and maintenance. In [4], Braga et al. use Class Diagrams to describe three patterns related to resource rental, trade and maintenance. They use the diagrams to stamp out models (which are also Class Diagrams) for various situations, such as library service, medical attendance, video rental, real estate rental and show box office. Their approach is purely syntactic. Apart from the basic structures, it is not clear what aspects

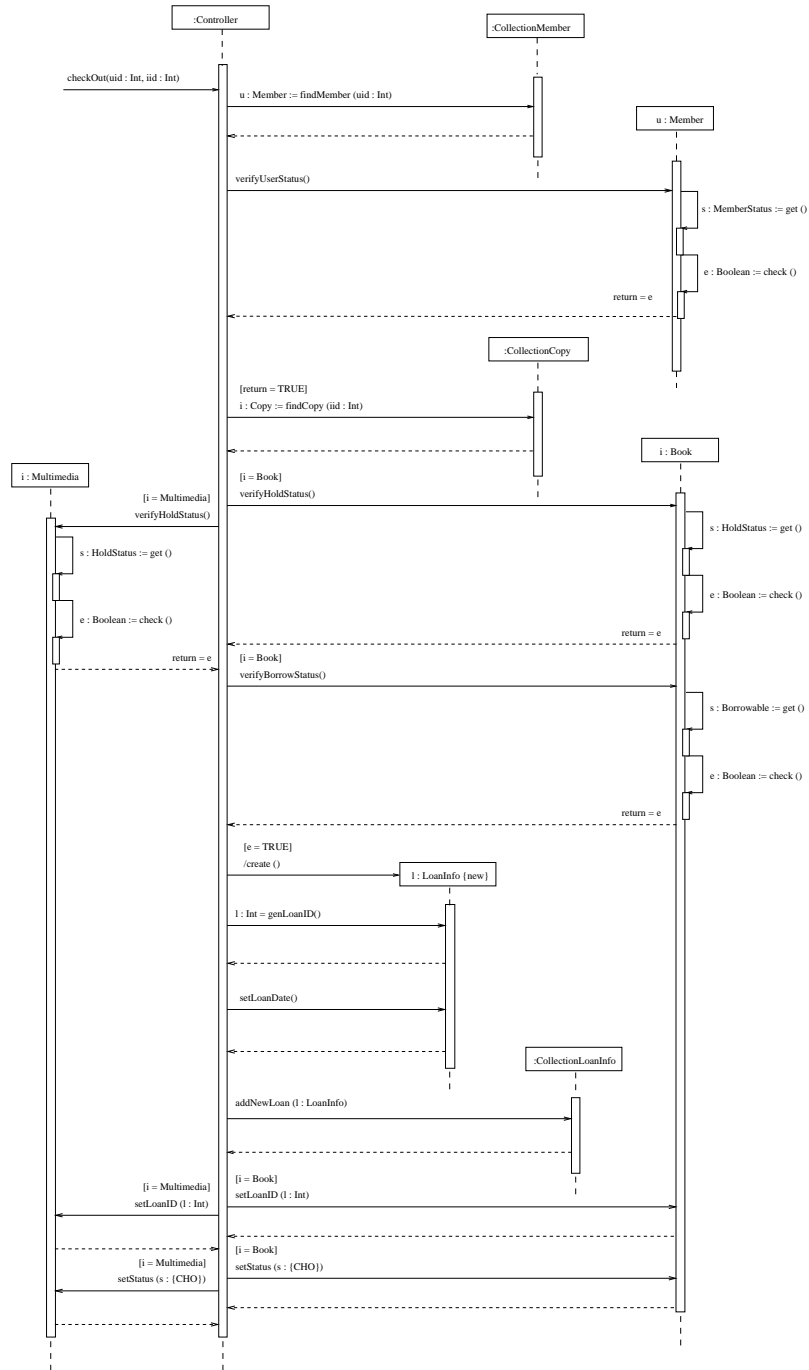


Figure 20: A Realization of the Library CheckOut IRM

of the pattern are described and how the pattern may be realized. Moreover, unlike Braga et al., we describe families of models at a higher level of abstraction.

Other work on precisely defining pattern properties include those of Lauder and Kent [16], and Guennec et

al. [14]. Lauder and Kent [16] use graphical constraint diagrams for precise visual presentation of patterns. Guennec et al. [14] use a metamodeling approach that is based on the UML metamodel. Their approach provides an alternative representation in terms of meta-collaborations that utilize a family of recurring properties initially pro-

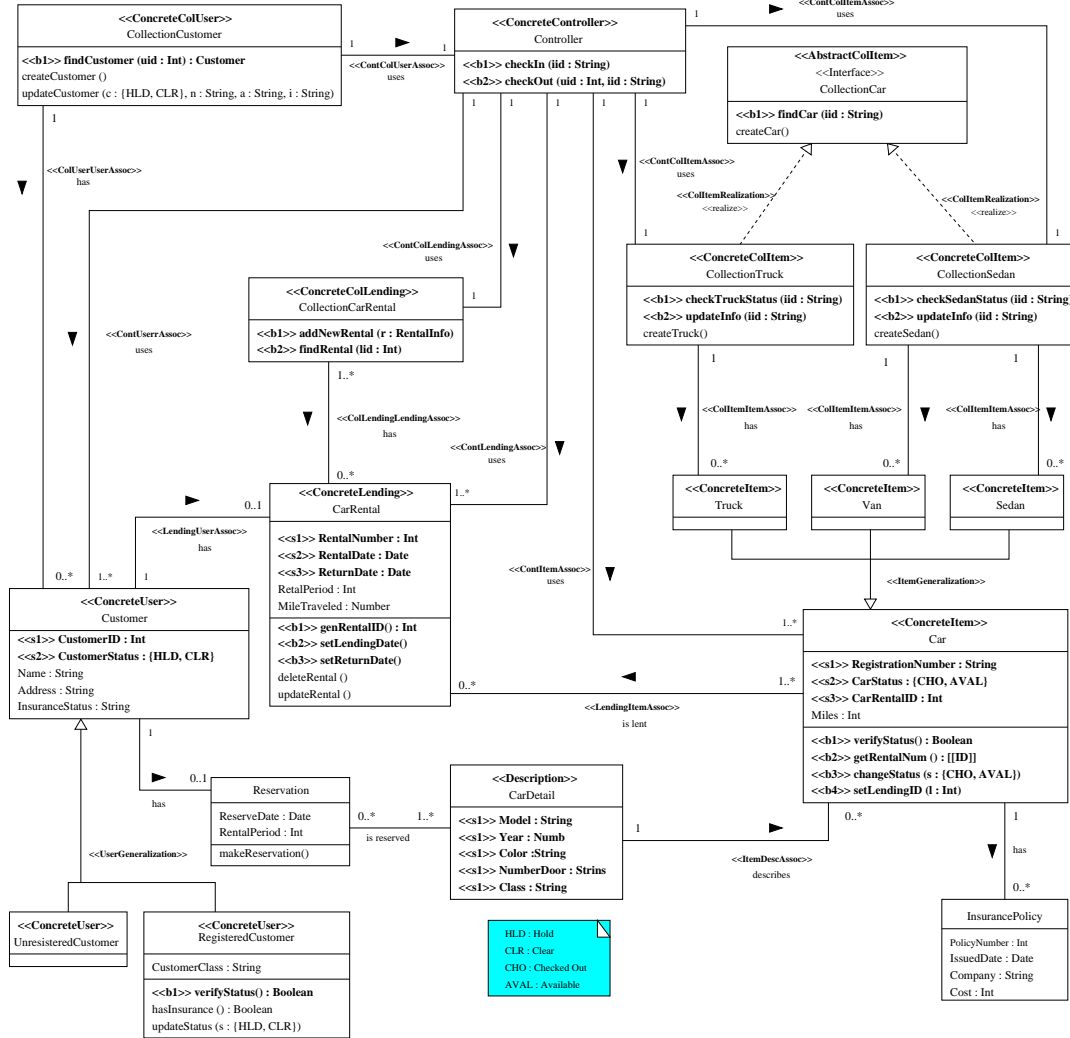


Figure 21: A Realization of the CICO Role Model : Car Rental Model

posed by Eden [8]. Pattern properties are expressed in terms of meta-collaborations that consist of roles played by instances of UML metamodel classes. The paper does not, however, describe how properties other than hierarchical structures of classifiers are specified. Nor is there a clear notion of what it means for a model to realize a role model.

Work has been done on developing object-based notions of roles (e.g., see [21, 22]). An object-based role specifies properties that objects in the run-time environment must have if they have to play the role. Our Role Models require that roles be played by model elements (e.g., classes and associations), and not by objects.

6. CONCLUSIONS

In this paper we introduced the notion of Role Models as a means for precisely characterizing families of

design models. Our notion of roles can be used as the basis for developing reusable models by incorporating role model elements into standard UML models (e.g., class diagrams, activity diagrams, collaboration diagrams). Different design models can be realized from the reusable models for different applications. The creation of reusable models would allow developers to raise the level of abstraction at which they develop systems (see <http://www.omg.org/mda>).

Roles can also be used to create design model frameworks. The roles in a framework characterize the design elements that can be used within the framework. A particular design model can be obtained from such a framework by plugging in realizations of the roles.

7. REFERENCES

- [1] G. Arango and R. Prieto-Diaz. Introduction and overview: Domain analysis concepts and research directions. In *Domain Analysis and Software Systems Modeling*. IEEE Press, 1991.
- [2] 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.
- [3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering, Addison-Wesley, 1998.
- [4] 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.
- [5] 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.
- [6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. Wiley, 1996.
- [7] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
- [8] A. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, Israel, 1999.
- [9] Software Technology for Adaptable Reliable Systems (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.
- [10] R. B. France, D. K. Kim, and E. Song. Patterns as precise characterizations of designs. Technical Report 02-101, Computer Science Department, Colorado State University, 2002.
- [11] R. B. France, D. K. Kim, E. Song, and S. Ghosh. Using roles to characterize model families. In *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics: Back to the Basics*, 2001.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [13] M. L. Griss. Software reuse: From library to factory. *IBM Systems Journal*, 32(4), 1993.
- [14] A. Guennec, G. Sunye, and J. Jezequel. Precise modeling of design patterns. In *Proceedings of UML'00*, 2000.
- [15] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis FODA: Feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, CMU, 1990.
- [16] A. Lauder and S. Kent. Precise visual specification of design patterns. In *Proceedings of ECOOP'98*, 1998.
- [17] 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., 1995.
- [18] J.-M. Morel and J. Faget. The REBOOT environment. In *Advances in Software Reuse*. IEEE Computer Society Press, March 1993.
- [19] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.
- [20] R. Prieto-Diaz. Status report: Software reusability. *IEEE Software*, 10(3), 1993.
- [21] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OORAM Software Engineering Method*. Manning/Prentice Hall, 1996.
- [22] D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*. ACM Press, 1998.
- [23] G. F. Rogers. *Framework-based software development in C++*. Prentice Hall, 1997.
- [24] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [25] The Object Management Group (OMG). Unified Modeling Language. Version 1.3, OMG, <http://www.omg.org>, June 1999.
- [26] W. Tracz, L. Coglianese, and P. Young. Domain-specific SW architecture engineering. *Software Engineering Notes*, 18(2), 1993.
- [27] D. S. Wile and J. C. Ramming. Special section: Domain specific languages. In *IEEE Transactions on Software Engineering*, 25(3). IEEE, 1999.