

Using Role-Based Modeling Language (RBML) to Characterize Model Families

Dae-Kyoo Kim, Robert France, Sudipto Ghosh, Eunjee Song
Computer Science Department
Colorado State University
Fort Collins CO 80523
{dkkim, france, ghosh, song}@cs.colostate.edu

Abstract

Cost-effective development of large, integrated computer-based systems can be realized through systematic reuse of development experiences throughout the development process. In this paper we describe a technique for representing reusable modeling experiences. The technique allows developers to express domain-specific design patterns as a sub-language of the modeling language, the UML. Use of the sub-language to build application-specific UML models results in the reuse of the embedded design experiences. We use a notation called the (meta-)Role-Based Modeling Language (RBML) to define UML sub-languages. A (meta-)Role Model is a specialization of the UML (Unified Modeling Language) meta-model, that is, it determines a sub-language of the UML. We show how RBML can be used to define domain-specific design patterns.

Keywords: Object-oriented design models, role-based modeling language, software reuse, UML

1. Introduction

There is a growing realization that development cycle time can be significantly shortened if reuse opportunities are exploited in all phases of software development, not just in the coding phase [2, 15, 16]. 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 emergence of the Unified Modeling Language (UML) [20] as a de-facto industry object-oriented (OO) modeling standard has the potential to remove this barrier.

Our reuse-related research focuses on developing mechanisms that facilitate timely development of UML models in well-defined domains through reuse of domain-specific modeling experiences. Arango and Prieto-Diaz [1] show that reuse of domain-specific experiences can significantly

enhance development productivity and quality. In this paper we present a technique for representing domain-specific design patterns (henceforth referred to as domain patterns) as sub-languages of the UML. Application developers that use the sub-languages to develop application-specific models are reusing the design experiences embedded in the domain patterns. The domain patterns can be incorporated into UML modeling tools that allow users to specialize the UML notation. Such tools must provide access to their internal representations of the UML metamodel. One can envisage a development environment in which *domain engineers* are responsible for defining domain patterns and incorporating them into UML modeling tools that *application engineers* use to develop application-specific models. The techniques described in this paper are thus targeted at domain engineers.

The technique uses a notation, called the (meta-)Role-Based Modeling Language (RBML), to describe domain patterns from a variety of perspectives. A (meta-)Role Model characterizes a family of UML diagrams, that is, it defines a sub-language for a particular kind of UML diagram. A domain pattern is defined as a set of Role Models, expressed using the RBML, where each Role Model defines a sub-language for a particular UML model diagram. Domain patterns are thus described from a variety of perspectives using Role Models.

Reuse of design experiences occurs whenever the sub-language defined by a domain pattern is used to build application-specific models. The application-specific models that conform to the rules of a sub-language determined by a Role Model are called *realizations* of the Role Model.

We illustrate our approach by using the RBML to create a domain pattern for a *Checkin-Checkout* (CICO) application domain. Applications within this domain include video-rental applications, car rental applications and library systems. Applications in this domain, provide services for checking in and and checking out items. The applications need to maintain information about the items that can be checked in and out. We use the RBML to define a UML sub-

language for modeling the structural and behavioral properties of such applications.

We describe the RBML in Section 2 and illustrate its use by developing the CICO domain pattern in Section 3. Sample realizations of the CICO Role Models are given in Section 4. We provide an overview of related work in Section 5, and conclude in Section 6.

2. RBML: Role-Based Modeling Language

The work described in this paper is concerned with defining specializations of the UML that incorporate domain-specific patterns. For this reason, the work is based on the UML metamodel. The UML metamodel characterizes valid UML diagrams. The specializations of the UML are defined using Role Models expressed in the RBML.

A Role Model is a structure of meta-roles (henceforth called roles), where a role defines properties that determine a family of UML model elements (e.g., class and generalization constructs). The type of model elements characterized by a role is determined by its base, where a role *base* is a UML metamodel class (e.g., *Class*, *Generalization*). For example, a role with the *Class* base determines a subset of UML class constructs. A UML model element conforms to, or plays (realizes) a role if it is an instance of the role's base and has the properties specified in the role. Such an element is also called a *realization* of the role. A Role Model is thus a characterization of UML diagrams, and a *Role Model realization* is a model (e.g., a static structural diagram, sequence diagram) that consists of realizations of the roles in the Role Model.

We have developed two types of Role Models:

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 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 UML metamodel class *Classifier* as a base) defines properties that

classifier constructs (e.g., classes, interfaces) must have if they are to realize the role, while a SRM relationship role defines properties that UML relationship constructs (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 that can exist for the role in a realization of the SRM that includes the role. 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, optional, compartment contains feature roles that determine a family of application-specific properties (e.g., properties represented by attributes and operations defined in application-specific classes).

Metamodel-level constraints are well-formedness rules, expressed in the Object Constraint Language (OCL) [20], 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. Currently, only classifier roles have feature roles. 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 paper, 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 expressed in terms of class attributes and operations. 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:

```
/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 attribute (or value-returning function in the case of a calculated attribute value) that plays this feature role in class that conforms to the SRM role. The constraint template enclosed in the brackets $\{, \}$ (parameters are surrounded by $[[,]]$) states that realizations of *CurrentValue* must be associated with a constraint that restricts its value to less than or equal to the value of a realization of

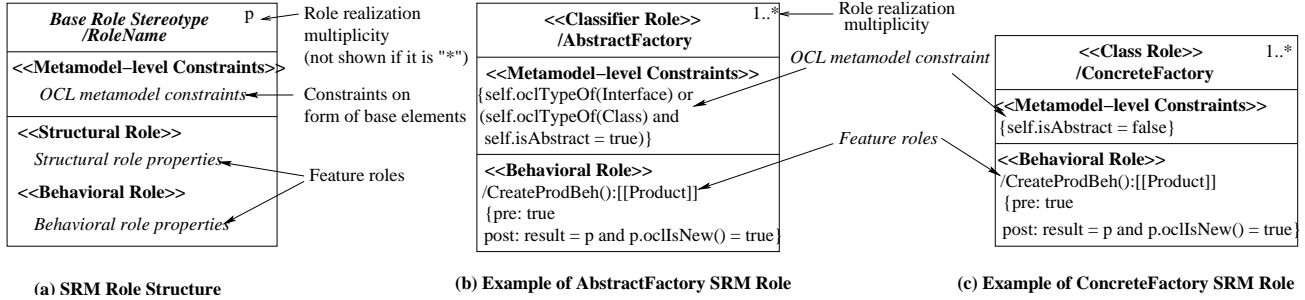


Figure 1. Structure of a SRM Role

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 *CreateProdBeh* shown in Fig. 1(b) and (c). A realization of *CreateProdBeh* is a behavior that creates a new instance of a realization of *Product*.

Substituting the names of realizations for the role names enclosed in the double square brackets ([[,]]) of constraint templates results in an application-specific property, called a *model-level constraint*, expressed in the OCL. For example, substituting *Wall*, the name of a *Product* role realization, for *Product* in the *CreateProdBeh* constraint template results in the following model-level constraint:

```
makeWall():Wall
{pre: true
 post: result = p and p.oclIsNew() = true }
```

Establishing that a model element realizes a SRM role involves proving that the constraints associated with the model element imply the model-level constraints obtained by suitably instantiating the role’s constraint templates, and determining that the realization multiplicities associated with the feature roles are not violated. Feature roles are detailed in [8] and [9].

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. We use the UML form of association to represent relationships between role. Role associations can be named and can have multiplicities associated with their ends. Examples of role relationships are illustrated in Fig. 2. The relationship shown between the

Subject and *Observer* roles in Fig. 2(b) is an abbreviated form of the role structure shown in Fig. 2(a) [8, 9]. The relationship states that a realization of *Subject* can have one or more (1..*) associations to realizations of *Observer* that play the *Observes* association role, and a realization of *Observer* can have one or more associations to *Subject* realizations that realize *Observes*.

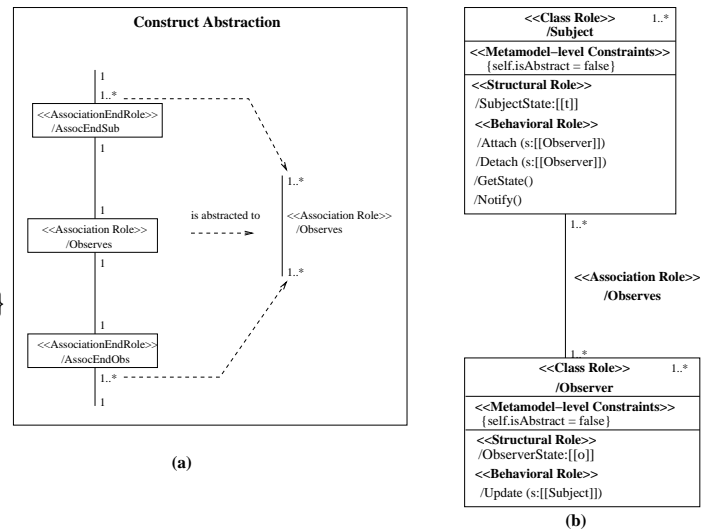


Figure 2. Relationships between roles in a SRM

A class diagram that consists of a class realizing *Subject* that is associated, via realizations of *Observes*, to three classes, each realizing *Observer* conforms to this SRM.

The expanded form is used when one wants to specify constraints on the multiplicities that can appear at multiplicity ends (these are specified as metamodel-level constraints in the *AssociationEnd* roles). In this paper we hide such detail and thus use the abbreviated form. Note that the multiplicities on the abbreviated form do *not* determine the multiplicities that will appear on realizations of the association

role.

2.1.3 Realizing SRMs

An SRM determines a sub-language for UML static models. Each role in a SRM specifies a specialization of its base class in the UML metamodel. A UML static model (e.g., a Class Diagram) conforms to a SRM if static model elements that are intended to be instances of the UML metamodel class specializations defined by the roles in the SRMs have the properties defined in the SRM. This involves establishing that the following:

- (1) For each model element that is intended to play a role (i.e., is an intended instance of the UML metamodel class specialization defined by the role) the model element (i) satisfies the metamodel-level constraint (the constraint evaluates to true for the model element) and (ii) for classifier constructs, the feature roles are realized by attributes and behaviors.
- (2) The model conforms to constraints expressed across roles in the SRM.

2.1.4 Role Hierarchies

SRMs often contain recurring role structures that can be viewed as role modeling patterns. An example of such a structure is the characterization of class hierarchies. Fig. 3(a) shows a characterization of a classifier hierarchy. The Role Model consists of an abstract role called *User*: an *abstract role* is one that is not intended to be realized - it is a classifier of roles. The specializations of the *User* role are *AbstractUser* and *ConcreteUser*. Since the *User* role is abstract, its realizations must be either realizations of *AbstractUser* or *ConcreteUser*. Realizations of the *User* specialization roles can be connected either by a realization of the *UserRealization* role (a UML $\ll realize \gg$ relationship), or a realization of the *UserGeneralization* role (a UML generalization relationship). This SRM determines a family of model structures that are hierarchies formed by generalization or realization relationships.

The above recurring role structure can be shown in an abstracted form, called a *folded* SRM. Fig. 3(b) shows the folded form of the SRM in Fig. 3(a). 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*.

2.2. 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

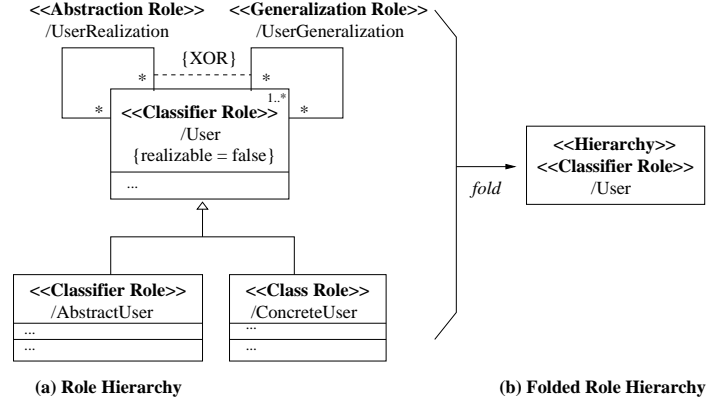


Figure 3. A Role Hierarchy Abstraction

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 (see Fig. 2) is shown in Fig. 4(a). The IRM describes an interaction pattern in which 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 subject state that is returned as a result of the *GetState* behavior.

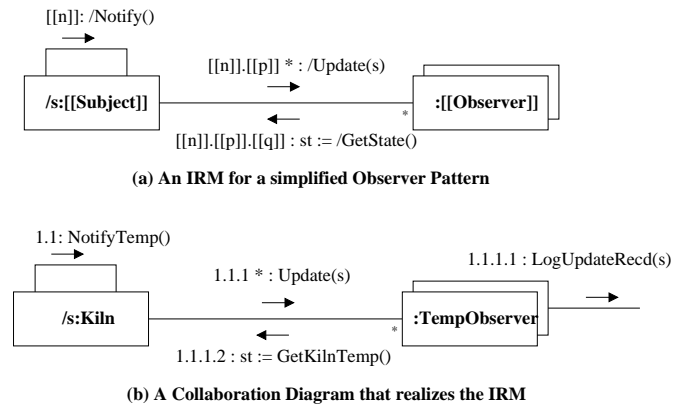


Figure 4. IRM for the simplified Observer pattern

The rectangular boxes in Fig. 4(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) [3]. 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. 4(b). The */s* represents a UML role named *s*; it does not have the same notion of roles in our RBML.

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 called are realizations of the behavioral feature roles that are named in the message roles.

The sequence labels in an IRM are generic (as indicated by surrounding them in *[[,]]*), 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* and the */GetState* interactions as indicated by the sequence expressions.

In the realization of the IRM shown in Fig. 4(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 Models for the CICO Domain Pattern

In this section, we describe the CICO domain pattern. Some of the common features of CICO applications are listed below:

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 (referred to as *lending* in this paper) if it is available.
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.

3.1. CICO SRM

Fig. 5 shows the SRM of the CICO model family. We make use of the folded $\ll \textit{Hierarchy} \gg$ stereotype for the *CollectionUser*, *User*, *CollectionLending*, *Lending*, *CollectionItem*, *Item*, and *Controller* roles.

3.1.1 User and Item Role Hierarchies

Fig. 6 shows the unfolded form of the *Item* hierarchy. Realizations of *Item* must have one or more realizations of the *verifyItemStatus*, *getLendingID*, *updateStatus*, and *setLendingID* feature roles (these roles are indexed by “b#”, where “b” indicates behavioral role). *verifyItemStatus* verifies whether or not the item has an appropriate status to be checked in or checked out. *getLendingID* is used to find a corresponding lending ID for the item to be checked in. *updateStatus* updates the status of an item, for example, whenever an item is checked out the status of the item is changed to reflect it is no longer available for check out. *setLendingID* provides a lending ID for the item to be checked out. These feature roles are inherited by the *AbstractItem* and *ConcreteItem* specializations of *Item* (*Item* is an abstract role). The *AbstractItem* can be realized either by interfaces or by abstract classes. The *ConcreteItem* must be realized by classes.

The structural roles are indexed by “s#” where “s” indicates structural roles. Realizations of *ConcreteItem* must possess only one realization each of *ItemCode* and *LendingID* (indicated by “1..1”), and one or more *ItemStatus* realizations. The status of the user indicates whether or not the item is available to be checked in or checked out. Details of the *ItemCode*, *ItemStatus*, and *LendingID* are described later in Section 3.1.4.

The *User* role has a similar structure (not shown). It has a *verifyUserStatus* behavioral role and *UserID* and *UserStatus* structural roles. *verifyUserStatus* is required to verify whether or not the user has an appropriate status for performing checkin or checkout. We don’t show the unfolded forms of the other role hierarchies because they have similar structures.

3.1.2 CollectionUser and CollectionItem Role Hierarchies

The *CollectionUser* role characterizes classes whose instances 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.

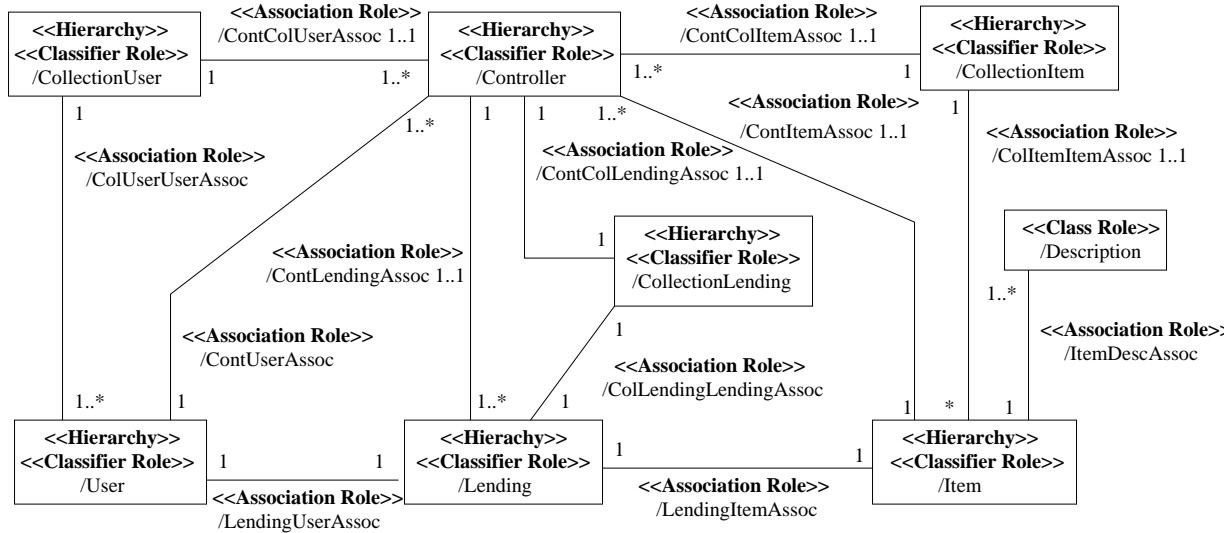


Figure 5. The folded CICO SRM

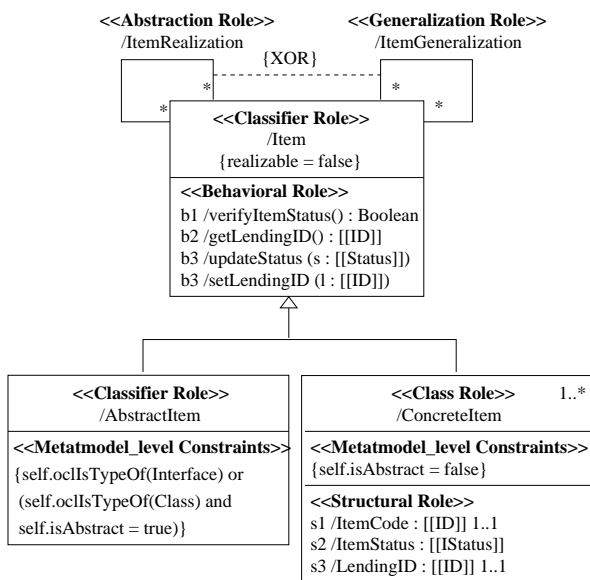


Figure 6. Unfolded Item SRM

3.1.3 CollectionLending and Lending Role Hierarchies

The *CollectionLending* role characterizes classes whose instances maintain *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* (*findLending*). There can be only one realization of the *ConcreteColLending* role.

A *Lending* role characterizes classes that maintain in-

formation about a particular checkin or checkout scenario. The *Lending* role provides behavioral roles for generating a new lending ID (*generateLendingID*), and obtaining the dates for lending (*setLendingDate*) and return (*setReturnDate*). The *ConcreteLending* role possesses structural roles *LendingID*, *LendingDate*, and *ReturnDate*. *ConcreteLending* role must have only one realization.

3.1.4 Controller, Description and Data Type Role Structures

Controller provides two behavioral roles, *checkIn* and *checkOut*. There must be only one realization of *ConcreteController* role in the *Controller* role hierarchy. The *Description* role must contain one or more realizations of the *DetailOfItem* role.

There are data type roles in the CICO SRM, for example, *ID*, *Descr* and *Date* (for the identity of a user or item, description and date, respectively). There are also *Enumeration* data type roles, for example, *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

Fig. 7 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* in-

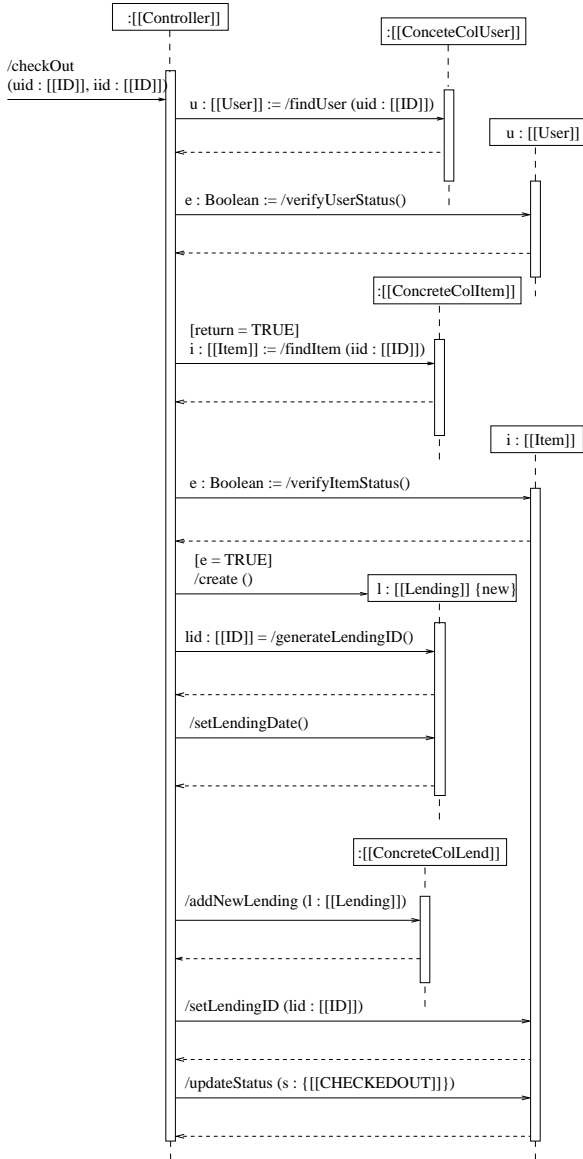


Figure 7. An IRM for a CheckOut Scenario

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 allowed, 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.

4. Realizations of the CICO metaRole Models

Role models can be realized to obtain design models for specific applications. We have developed three different realizations (library system, car rental system, revision control system) of the CICO domain pattern. In this section, we present part of a library system realization. We use bold stereotypes in the model constructs to indicate that the constructs are realizations of the roles named in the stereotypes. Feature properties that are printed in bold realize feature roles in the CICO Role Models.

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. 8 shows the class diagram of the library as a realization of the CICO SRM. We describe a few important aspects of the CICO SRM realization below:

- A *hierarchy* is illustrated in the realization of *Item* by *Copy* which is specialized by *Multimedia* and *Book*.
- *Multimedia* is a realization of *ConcreteItem*, as 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. 6).
- *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 a realization of the association role between the *CollectionCopy* and the *Copy* in the SRM.

• 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.

• *ConcreteLending* Lending role is constrained to have only one realization in Section 3.1.3. 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*.

Note that the realization also contains additional features that are not determined by the CICO domain pattern. These additional features are listed below:

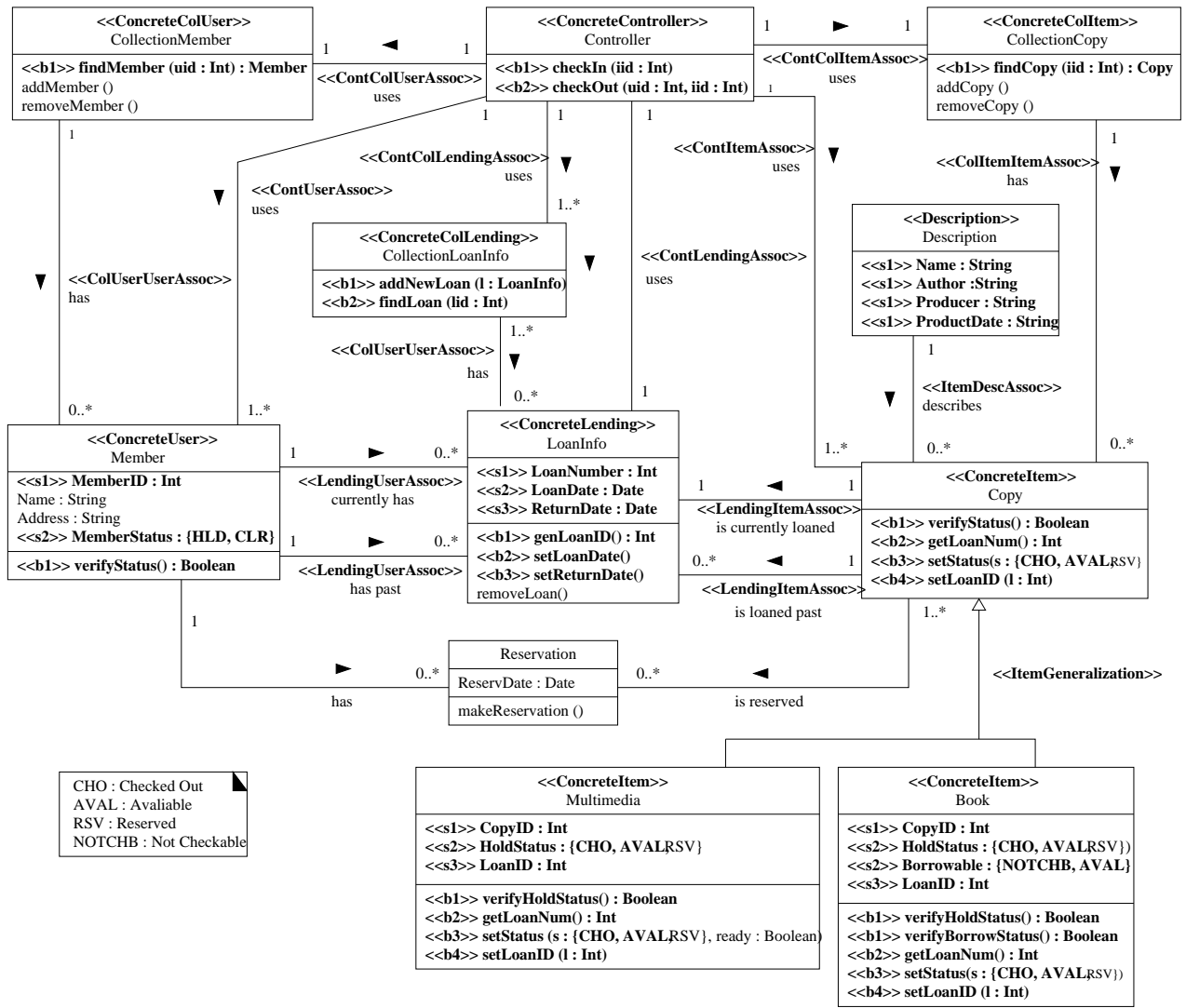


Figure 8. A Realization of the CICO SRM: Library Model

• *The capability to reserve items in the library:* The *Reservation* class allows members to reserve copies.

• *Extra attribute features and operations:* The attributes *Name* and *Address* in *Member* are not part of the CICO domain pattern. 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.

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. Figure 9 shows a realization of the CICO IRM for *checkOut*.

5. Related Work

Work on domain-specific languages [22] 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

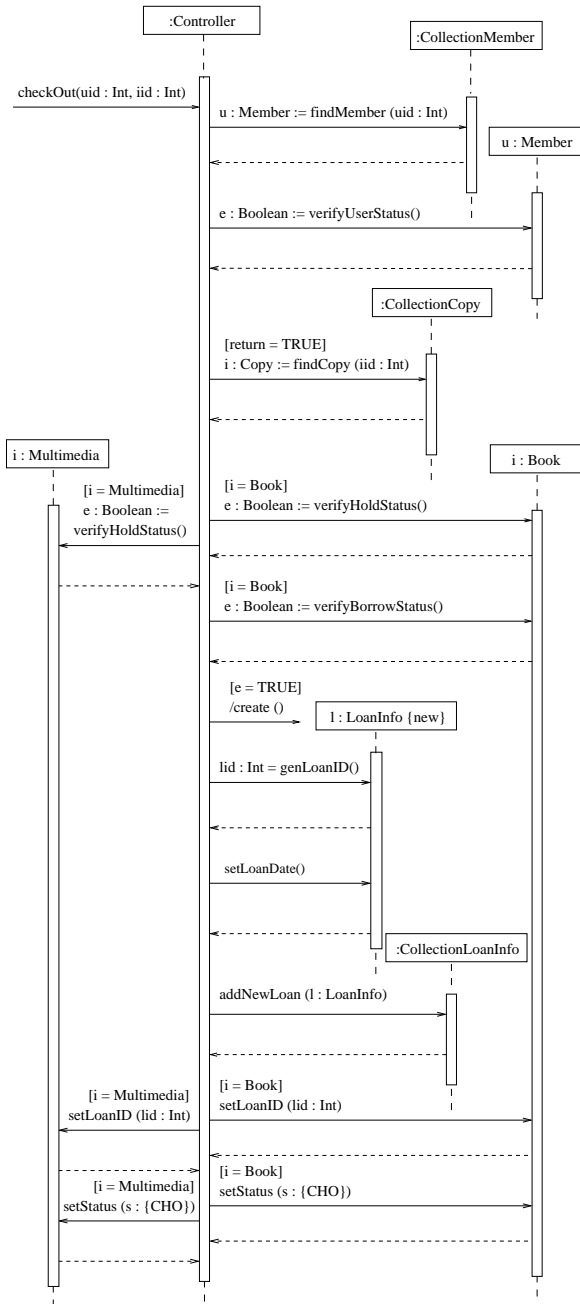


Figure 9. A Realization of the Library Check-Out IRM

programming languages (DSPLs). Our work focuses on incorporating reusable experiences into modeling languages.

Other forms of reusable experiences packaged for vertical reuse are frameworks [19] and domain-specific architectures (e.g., see [7, 10, 14, 21]). There is a considerable body of work on domain engineering processes and

domain modeling notations (e.g. see [1, 10, 12, 15, 21]). Our approach can complement the above efforts by providing mechanisms for representing and incorporating patterns into the UML to create specializations of UML constructs that reflect domain-specific patterns.

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, with limited support for specifying constraint patterns.

Other work on precisely defining pattern properties include those of Lauder and Kent [13], and Guennec et al. [11]. Lauder and Kent [13] use graphical constraint diagrams for precise visual presentation of patterns. Guennec et al. [11] 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 proposed by Eden [6]. 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 [17, 18]). An object-based role specifies properties that objects in the run-time environment must have if they have to play the role. Our RBML requires that roles be played by model elements (e.g., classes and associations), and not by objects.

6. Conclusions

In this paper we gave an overview of the Role-Based Modeling Language (RBML), and outline how it can be used to define Role Models that determine a specialization of the UML.

We plan to use an existing tool that provides access to the UML metamodel (e.g., the Objecteering tool - see www.softteam.fr) to develop prototype support for incorporating Role Models into UML tools. We are also developing a specialized form of Role Models that will allow stamping out of conforming models. Such models can provide convenient start points for developing application-specific models.

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] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [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] A. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, Israel, 1999.
- [7] S. T. for Adaptable Reliable Systemms (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.
- [8] 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.
- [9] R. B. France, D. K. Kim, E. Song, and S. Ghosh. Using Roles to Characterize Model Families. In *Proceedings of the 10th OOPSLA Workshop on Behavioral Semantics: Back to Basics*, Seattle, Washington, November 2001.
- [10] M. L. Griss. Software reuse: From library to factory. *IBM Systems Journal*, 32(4), 1993.
- [11] A. Guennec, G. Sunye, and J. Jezequel. Precise modeling of design patterns. In *Proceedings of UML'00*, 2000.
- [12] 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.
- [13] A. Lauder and S. Kent. Precise visual specification of design patterns. In *Proceedings of ECOOP'98*, 1998.
- [14] 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.
- [15] J.-M. Morel and J. Faget. The REBOOT environment. In *Advances in Software Reuse*. IEEE Computer Society Press, March 1993.
- [16] R. Prieto-Diaz. Status report: Software reusability. *IEEE Software*, 10(3), 1993.
- [17] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OORAM Software Engineering Method*. Manning/Prentice Hall, 1996.
- [18] D. Riehle and T. Gross. Role Model Based Framework Design and Integration. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 117–133, Vancouver, Canada, October 1998. ACM Press.
- [19] G. F. Rogers. *Framework-based software development in C++*. Prentice Hall, 1997.
- [20] The Object Management Group (OMG). Unified Modeling Language. Version 1.3, OMG, <http://www.omg.org>, June 1999.
- [21] W. Tracz, L. Coglianesi, and P. Young. Domain-specific SW architecture engineering. *Software Engineering Notes*, 18(2), 1993.
- [22] D. S. Wile and J. C. Ramming. Special section: Domain specific languages. In *IEEE Transactions on Software Engineering*, 25(3). IEEE, 1999.