



A UML-based language for specifying domain-specific patterns

Dae-Kyoo Kim, Robert France*, Sudipto Ghosh

Computer Science Department, Colorado State University, Fort Collins, CO 80523, USA

Received 13 June 2003; received in revised form 12 November 2003; accepted 5 January 2004

Abstract

Cost-effective development of large computer-based systems can be realized through systematic reuse of domain-specific design experience. Such experience can be captured by domain-specific design patterns, that is, patterns specifying design solutions for well-defined families of applications. This paper presents a pattern specification notation called the Role-Based Metamodeling Language (RBML) and shows how it can be used to express domain-specific patterns. An RBML pattern defines a domain-specific sub-language of the Unified Modeling Language (UML). Developers can use the sub-language to create UML diagrams for applications in the domain addressed by the pattern. An RBML pattern characterizing check-in and out of items. Applications in the CICO domain manage checking in and out of items. Examples of such applications are car rental, library, and video rental systems. The paper also describes how a UML model of a library system can be obtained from the RBML CICO pattern.

© 2004 Elsevier Ltd. All rights reserved.

Keywords: Object-oriented design models; Patterns; Role-based metamodeling 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

*Corresponding author. Tel.: +1-970-491-6356; fax: +1-970-491-2466.

E-mail addresses: dkkim@cs.colostate.edu (D.-K. Kim), france@cs.colostate.edu (R. France), ghosh@cs.colostate.edu (S. Ghosh).

[1–3]. In the past, a barrier to the reuse of experience 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) [4] as the de facto object-oriented (OO) modeling standard has the potential to remove this barrier.

Reusable software artifacts can be classified as general-purpose or domain-specific. General-purpose artifacts support horizontal reuse, that is, they can be used in a variety of application domains. A reusable domain-specific artifact is created specifically for use in a particular application domain and thus supports vertical reuse. While horizontal reuse is useful, Arango and Prieto-Diaz [5] make the case that reuse of high quality domain-specific experience can result in more significant improvement in development cycle-time and software quality.

This paper presents a technique for expressing domain-specific design patterns (henceforth called domain patterns) that define domain-specific UML sub-languages. Use of the sub-languages to build models of applications in the domains results in reuse of design experience embedded in the domain patterns. The technique uses a notation called the *Role-Based Metamodeling Language (RBML)* to express domain patterns. An RBML specification defines a sub-language for a particular type of UML diagram. For example, an RBML specification that defines a domain-specific UML class diagram sublanguage can be used to create class diagrams for applications in the domain. A domain pattern consists of a number of RBML specifications, each defining a sub-language for a UML diagram type.

Modeling tools that allow users to define and utilize domain patterns are needed to support systematic reuse of domain-specific modeling experience. One can envisage a development environment in which domain engineers develop domain patterns and embed them in modeling tools so that application developers can use them to develop application-specific models. The technique and notation described in this paper can be used as the base for such modeling environments.

To illustrate the technique, the RBML is used to create a domain pattern for a *checkin-checkout (CICO)* application domain. The primary purpose of applications in this domain is to provide services for checking in and out items. Applications within this domain include video rental, car rental, and library systems.

An overview of the UML is given in Section 2. The RBML is described in Section 3, and Section 4 shows how the RBML can be used to develop a CICO domain pattern. The CICO domain pattern is used to build a UML model of a library application in Section 5. An overview of related work is given in Section 6, and Section 7 concludes the paper.

2. Background

The domain pattern approach to defining domain-specific modeling languages utilizes the UML in two ways: (1) the RBML syntax is based on the UML syntax, and (2) a domain pattern defines a UML sub-language by specializing the UML metamodel. This section presents an overview of the UML and its metamodel.

2.1. UML models

A UML design model consists of a number of diagrams, each describing the design from a different perspective. The UML design models considered in this paper consist of class diagrams and sequence diagrams. Examples of the UML diagrams used in this paper are shown in Fig. 1.

In this paper, class diagrams are expressed using notation defined in UML v1.4, but the models are compliant with the recently approved UML 2.0 (see <http://www.omg.org/uml>). The sequence diagram notation used to describe behavior is compliant with UML 2.0, except where we extended the notation to concisely represent repetitive behavior over collections of objects. The extensions will be described when used in the paper.

A UML class diagram consists of classifiers (e.g., classes, interfaces) and relationships between classifiers. An operation in a class can be specified using pre- and postconditions expressed in the Object Constraint Language (OCL) [6]. Associations between classes specify links between class objects. The ends of associations, referred to as *association-ends*, have properties such as multiplicity and navigability.

A sequence diagram describes how instances interact to accomplish a task. An interaction is expressed in terms of *lifelines* and *messages*. A lifeline is a participant in an interaction. In this paper, participants are class objects. A message is a specification of a class of stimuli passed between two objects. A stimulus is a communication and can be a request to invoke a recipient's method or a signal that informs its recipient of the occurrence of an event. In this paper we restrict our attention to messages that represent method calls.

2.2. The UML metamodel

The UML metamodel characterizes the set of all valid UML models. It consists of a class diagram and a set of well-formedness rules that define the abstract syntax of the UML. It also provides an informal description of the UML semantics. The

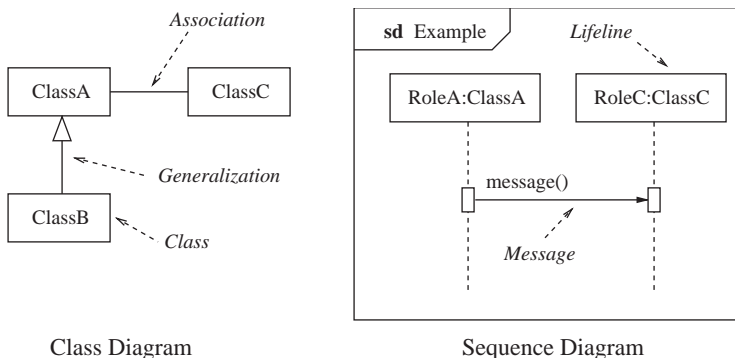


Fig. 1. Overview of UML class and sequence diagrams.

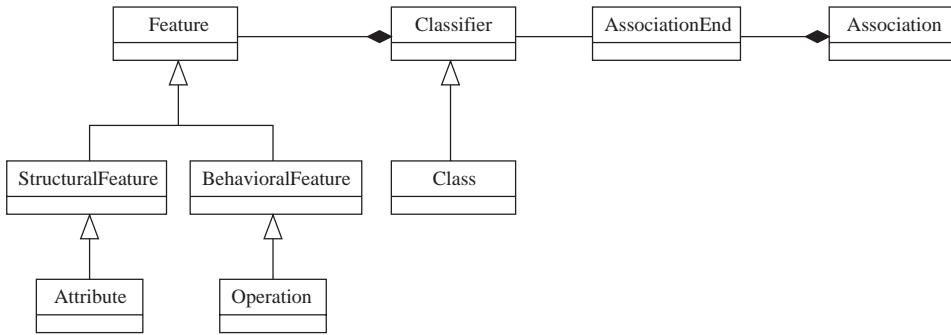


Fig. 2. A fragment of the UML metamodel.

metamodel class diagram consists of classes (referred to as *metaclasses*) describing UML abstract syntactic elements. For example, instances of the *Association* metaclass in the metamodel are UML associations. Well-formedness rules that are not expressible in the metamodel class diagram are expressed using the OCL where possible, and in natural language otherwise [4].

A fragment of the UML metamodel class diagram (from UML v1.4) is shown in Fig. 2. The fragment shows the relationships between UML classifiers (instances of *Classifier*) and their features (instances of *Attribute* and *Operation*), and the relationships between classifiers and associations: an association (instance of *Association*) has association-ends (instances of *AssociationEnd*) that are connected to classifiers.

A domain pattern defines a UML sub-language by introducing specializations (subclasses) of the metaclasses into the UML metamodel. The specializations define abstract syntactic elements that represent domain-specific concepts.

3. The RBML: role-based metamodeling language

The RBML is a language for specifying families of UML models. An RBML specification is a structure of roles [7], where a role specifies properties that model elements must have if they are to play (conform to) the role. Each role is associated with a UML metamodel class (a metaclass) that is called its *base*. The properties expressed in a role define a subset of the base metaclass instances. For example, a role whose base is the *Association* metaclass expresses properties that define a subset of UML associations (instances of *Association*). Formally, a role defines a specialization (subclass) of its base metaclass.

In this paper, a UML model consists of a class diagram and a set of sequence diagrams. The domain patterns described in this paper consist of the following RBML specifications: (1) A *Static Pattern Specification* (SPS) that characterizes conforming class diagrams, and (2) a set of *Interaction Pattern Specifications* (IPSs) that characterize conforming interaction diagrams. A domain pattern's SPS defines a

UML sub-language for class diagrams and its IPSs collectively define a UML sub-language for sequence diagrams.

3.1. Static pattern specifications (SPSs)

Class diagrams that conform to a domain pattern must have the properties specified in the pattern's SPS. An SPS consists of *classifier* and *relationship* roles, where a classifier role is connected to other classifier roles by relationship roles. The base of a classifier role is one of the metaclasses in the *Classifier* generalization hierarchy defined in the UML metamodel. Similarly, the base of a relationship role is one of the metaclasses in the *Relationship* generalization hierarchy.

Properties in a classifier role can be expressed in three forms:

StructuralFeature roles specify structural features of conforming classifiers. A structural feature can be either an attribute or a query (i.e., a value-returning function with no side-effects). *StructuralFeature* roles can be associated with *constraint templates* that are used to produce OCL constraints associated with conforming structural features.

BehavioralFeature roles specify behavioral features of conforming classifiers. A behavioral feature can be implemented by one or more operations. *BehavioralFeature* roles can also be associated with constraint templates that are used to produce operation specifications associated with conforming operations.

Metamodel-level constraints are well-formedness rules that restrict the form of conforming model elements. For example, a metamodel-level constraint can restrict conforming classifiers to be abstract classes. Metamodel-level constraints are expressed in the OCL.

StructuralFeature roles and *BehavioralFeature* roles are referred to as child roles of the parent classifier roles in which they are contained. Fig. 3(a) shows part of an SPS that characterizes class diagrams that conform to a variant of the *Observer* pattern [8]. The variant is referred to as the *RestrictedObserver* pattern. In this pattern, there are one or more observer classes and one or more subject classes in a conforming class diagram. Each observer class must be associated with exactly one subject class, and each subject class must be associated with exactly one observer class.

The symbol “|” is used to indicate roles in RBML specifications. The SPS in Fig. 3(a) consists of two class roles, *Subject* and *Observer*, that are connected by an association role *Observes*. Each role in an SPS can be associated with a *binding multiplicity* that restricts the number of conforming elements that can be bound to the role in a conforming model. In the case of a child role contained in a parent role (e.g., a *StructuralFeature* role contained in a parent classifier role), the binding multiplicity restricts the number of elements in a parent element that can be bound to the child role. For example, the *SubjectState* role is associated with a binding multiplicity 1..1, indicating that a conforming *Subject* class must have exactly one structural feature that plays (is bound to) the *SubjectState* role.

A class that conforms to the *Subject* role has exactly one structural feature that conforms to the *SubjectState* role and exactly one behavioral feature that conforms

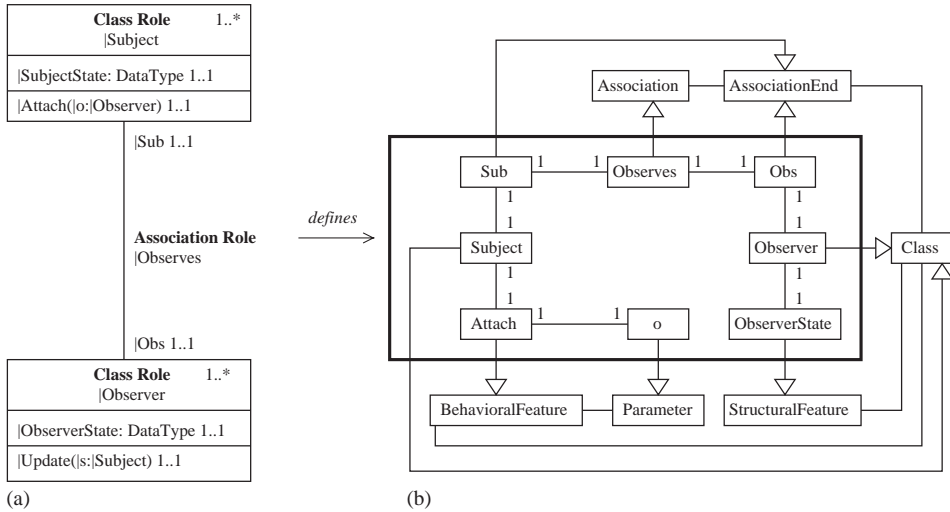


Fig. 3. (a) A partial SPS for the RestrictedObserver pattern and (b) partial view of its specialized UML metamodel.

to the *Attach* role. The *Subject* role is also associated with two behavioral roles not shown in the SPS: a *Notify* role that specifies behavior that is invoked whenever the subject state is changed, and a *Detach* behavioral role specifying behavior that removes an observer from a subject. A conforming *Observer* class must have exactly one structural feature that conforms to the *ObserverState* role and exactly one behavioral feature that conforms to the *Update* role.

The association role *Observes* specifies associations between subject and observer classes. Each end of an association role has an association-end role. The *Observes* role has two association-end roles: *Sub* and *Obs*. The binding multiplicity on the *Sub* role (1..1) specifies that a conforming *Subject* class must be part of only one *Observes* association. Similarly, a conforming *Observer* class must be part of only one *Observes* association. An association-end role contains metamodel-level constraints that express properties associated with the ends of conforming associations (e.g., multiplicity properties). The properties specified in the *Sub* and *Obs* role restrict the multiplicities associated with conforming association-ends. These properties are described in Section 3.1.1.

Each role defines a subclass (specialization) of its base metaclass. Fig. 3(b) shows some of the subclasses determined by the roles in the *RestrictedObserver* SPS. For example, the class role *Observer* in Fig. 3(a) defines a subtype of the metaclass *Class* called *Observer*. An SPS thus defines a specialization of the UML class diagram metamodel.

3.1.1. Metamodel-level constraints

Metamodel-level constraints are well-formedness rules associated with the metaclass specializations defined by RBML specifications. The following are some

of the metamodel-level OCL constraints associated with roles in the *Restricted Observer* pattern:

- (1) *Subject* role: Classes that conform to the *Subject* role must be concrete.

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

In the above, *isAbstract* is an attribute of the metaclass *Class* and *self* refers to a class that conforms to the *Subject* role.

- (2) *Sub* role: An association-end that conforms to *Sub* must be navigable and must have a multiplicity of 0..1 or 1..1. Note that an association-end multiplicity is not the same as a binding multiplicity for an association-end role. For example, the binding multiplicity (1..1) associated with the *Sub* association-end role in Fig. 3(a) restricts the number of conforming association-ends that can be bound to the role, and an association-end multiplicity is a property of an association-end.

context Sub **inv:**
self.multiplicity.lower \geq 0 and self.multiplicity.upper = 1
and isNavigable = true

- (3) An association-end that conforms to *Obs* must be navigable and must have a multiplicity of 0 or more (i.e., 0..*):

context Obs **inv:**
self.multiplicity.lower = 0 and self.multiplicity.upper = *
and isNavigable = true

To reduce diagram clutter, metamodel-level constraints expressed in the OCL are not shown in the RBML diagrams presented in this paper.

3.1.2. Constraint templates

Constraint templates are used to specify semantic properties associated with features that conform to structural and behavioral feature roles. For example, a behavioral feature that conforms to the *Attach* role must define a behavior in which an observer is attached to a subject. A constraint template associated with the *Attach* role specifies conforming behavior in terms of pre- and postconditions. Such a template is referred to as an *operation template*. The operation template for the *Attach* feature role is given below:

context |Attach(|o:|Observer)
pre: true
post: self.|Observes = self.|Observes@pre \rightarrow including(|o)

The template specifies behavior in which an observer (an object that plays the role *o*) is attached to the subject. The observer that plays the parameter role *o* has a

type that is a conforming *Observer* class. The expression *self*.|*Observes* evaluates to the set of observers linked to the subject via an association that conforms to the *Observes* role. The expression *self*.|*Observes@pre* evaluates to the set of observers attached to the subject before execution of an *Attach* behavior.

Instantiating an operation template produces a pre- and postcondition specification for an operation expressed in the OCL. Template instantiation is described in Section 3.3. Structural feature roles can also be associated with constraint templates that can be used to obtain constraints associated with conforming structural features.

3.1.3. Role hierarchy

Many of the RBML specifications we have created include descriptions of classifier hierarchies formed using generalization or realization relationships. The role structure that specifies classifier hierarchies is shown in Fig. 4.

AbstractRole is a classifier role that is associated with relationship roles *RoleRealization* and *RoleGeneralization*. The classifiers that conform to the *AbstractRole* role are classifiers in a generalization or realization hierarchy. The *AbstractRole* classifier role is abstract, that is, conforming classifiers must conform to at least one of its role specializations (*AbstractClassifier* and *ConcreteClass*). The following are the well-formedness rules associated with role hierarchies:

- (1) A conforming *AbstractClassifier* classifier must either be an interface or an abstract class:

```

context AbstractClassifier inv
    self.oclIsTypeOf(Interface) or
    (self.oclIsTypeOf(Class) and self.isAbstract = true)
    
```

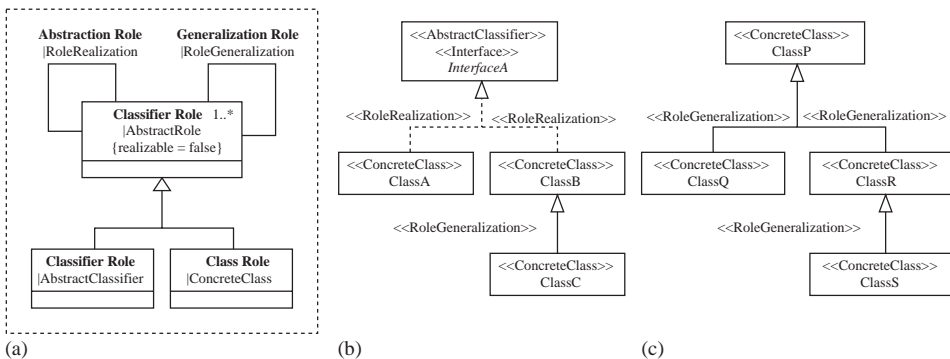


Fig. 4. Role hierarchy and conforming structures: (a) role hierarchy; (b) conforming Model A; (c) conforming Model B.

(2) Conforming *ConcreteClass* classifiers must be concrete classes:

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

A classifier in a conforming *AbstractRole* hierarchy can be an abstract class or an interface (i.e., a classifier conforming to *AbstractClassifier*) or it can be a concrete class (i.e., a class conforming to *ConcreteClass*).

Fig. 4(b) shows a structure that conforms to the role hierarchy model:

- (1) *InterfaceA* is an interface that plays the *AbstractClassifier* role,
- (2) *ClassA*, *ClassB*, and *ClassC* play the *ConcreteClass* role,
- (3) the realization relationship between *InterfaceA* and its class realizations plays the *RoleRealization* role in the hierarchy, and
- (4) the generalization relationship between *ClassB* and *ClassC* plays the *RoleGeneralization* role.

Fig. 4(c) is another conforming model that has a similar structure except that the generalization relationship between *ClassP* and its subclasses play the *RoleGeneralization* role.

A role hierarchy can be concisely represented in the RBML by a single construct called the folded hierarchy role. An example of a folded hierarchy role for the *AbstractRole* hierarchy is shown in Fig. 5.

3.2. Interaction pattern specifications (IPSS)

IPSSs are used to constrain interactions between pattern participants. An IPS consists of an *interaction* role that defines a specialization of the UML meta-model class *Interaction*. An interaction role is a structure of *lifeline* and *message* roles whose bases are *Lifeline* and *Message* metaclasses. Each lifeline role is associated with a classifier role in an SPS: a participant that plays a lifeline role is an instance of a classifier that conforms to the classifier role. A message role is associated with a behavioral feature role in an SPS: a conforming message specifies a call to an operation that conforms to a behavioral feature role.

The IPS that describes the pattern of interactions that takes place as a result of invoking a subject's *Notify* operation is shown in Fig. 6(a). In the figure the lifeline role $|s : |Subject$ represents instances of a class that conforms to the classifier role *Subject* in Fig. 3(a) and the message role *Update* represents asynchronous calls to

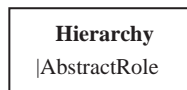


Fig. 5. Folded form of a role hierarchy.

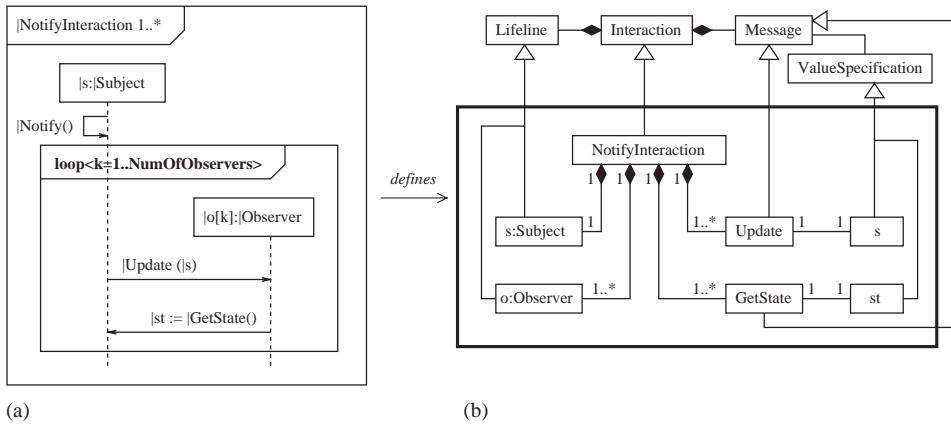


Fig. 6. (a) An IPS for the RestrictedObserver pattern and (b) its specialized UML metamodel.

operations that conform to the feature role *Update*. The *Notify* behavior results in calls to operations that conform to the *Update* role for each observer associated with the subject. Each observer then calls the operation that conforms to *GetState* in the subject.

The UML currently does not have a convenient notation for concisely representing iterative behavior over a collection of objects. The *loop* structure shown in Fig. 6(a) is an extension that allows one to specify such iterative behavior concisely. The lifeline labeled *o[k]* represents the *k*th observer attached to the subject playing the role *s*. The value of *k* ranges from 1 to the number of observers attached to the subject. The function *NumOfObservers* returns the number of observers linked to the subject playing the role *s*. It is defined as follows:

context Subject::NumOfObservers():Integer
post: result = self.Observe → size()

The postcondition states that the function returns the number of elements attached to a subject instance via an association that conforms to *Observe*.

An IPS defines a specialized UML metamodel that specifies a family of sequence diagrams as shown in Fig. 6(b). For example, the interaction role *NotifyInteraction* defines a specialization of the UML metamodel class *Interaction*, and the lifeline role labeled *|s : |Subject* defines a specialization of the UML metamodel class *Lifeline* labeled *s : Subject* in Fig. 6(b).

3.3. Obtaining conforming models from role models

Obtaining a conforming model from a Role Model involves binding application-specific model elements to roles. Fig. 7(b) shows part of a model obtained by binding model elements to roles in an SPS for a less restrictive variant of the Observer pattern called the *LooseObserver* pattern (see Fig. 7(a)). The partial *LooseObserver* SPS

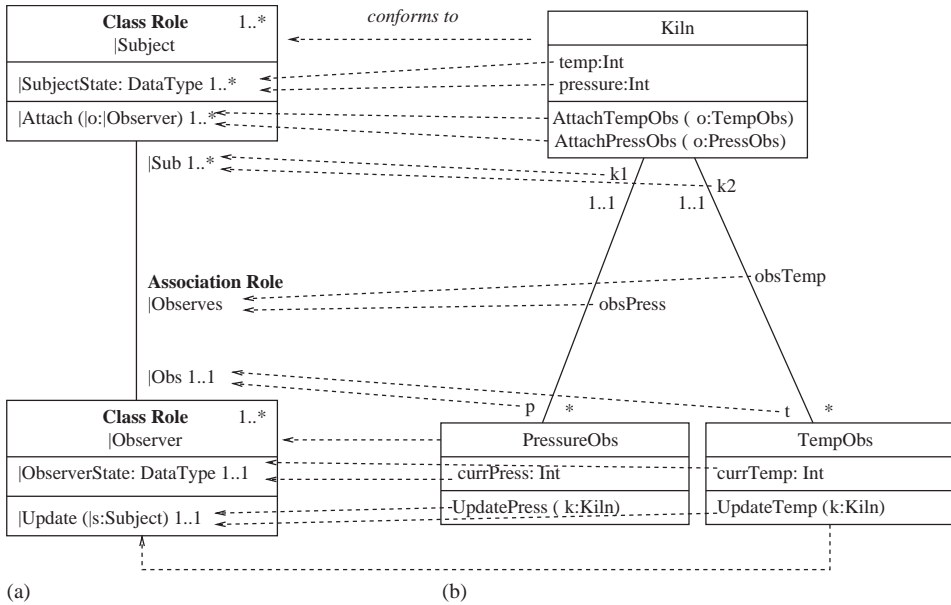


Fig. 7. (a) A partial LooseObserver SPS (in Fig. 3) and (b) a conforming class diagram.

shown in Fig. 7(a) characterizes class diagrams in which subject classes are associated with one or more observer classes. Each subject class can consist of one or more structural features that conform to *SubjectState*, and one or more behavioral features that conform to *Attach*. The *Subject* role is also associated with a *Notify* and a *Detach* behavioral role (not shown in the diagram).

The class diagram shown in Fig. 7(b) describes a system in which a sensor (instance of *Kiln*) records the temperature and pressure in a kiln. The sensor is linked to temperature and pressure observers that monitor kiln temperature and pressure. The bindings used to create the *Kiln* class diagram are indicated by the directed dashed lines between the class diagram and the SPS.

The *SubjectState* role binding multiplicity allows one or more model elements to be bound to it. This role is bound to two attributes, *temp* and *pressure*. The binding multiplicity associated with the *Sub* association-end role indicates that one or more association-ends can be associated with a class that conforms to the *Subject* role. The two association-ends, *k1* and *k2* attached to the *Kiln* class are bound to the *Sub* role. The association-end multiplicities on *k1* and *k2* satisfy the metamodel-level constraints associated with the *Sub* role. The constraint is the same as that given for the *Sub* role in the *RestrictedObserver* pattern (see Section 3.1.1). The binding multiplicity associated with the *Obs* role indicates that an observer class is associated with exactly one subject class. The association-ends *p* and *t* are bound to the association-end role *Obs*.

Specifications for the operations *AttachTempObs* and *AttachPressObs* that are bound to the *Attach* role are obtained by instantiating the *Attach* constraint template

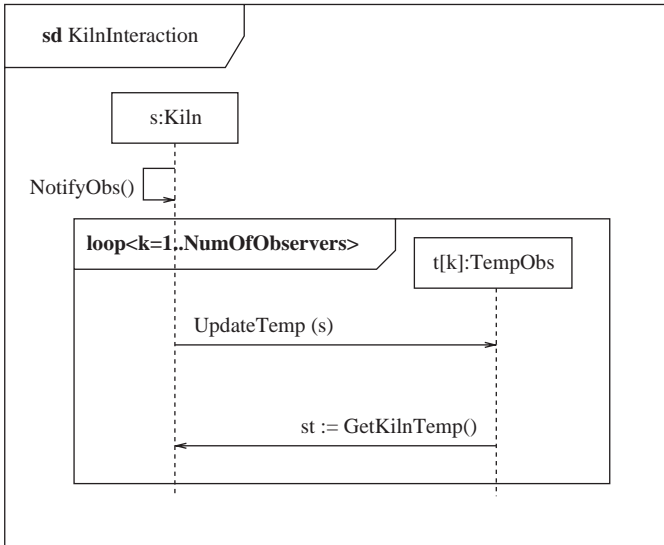


Fig. 8. A conforming NotifyInteraction sequence diagram.

given in Section 3.1.2. For example, the pre- and postconditions for the *AttachTempObs* is given below:

context AttachTempObs(tobs: TempObs)

pre: true

post: self.obsTemp = self.obsTemp@pre \rightarrow including(tobs)

The constraint is obtained by binding role names in the *Attach* operation template to the following elements: *o* is bound to *tobs*, *Observes* is bound to *obsTemp*, and *Observer* is bound to *TempObs*.

The *LooseObserver Notify* IPS is the same as the one given for the *RestrictedObserver* IPS (see Fig. 6). A sequence diagram obtained by binding model elements representing kiln system concepts to roles in the *NotifyInteraction* IPS given in Fig. 6 is shown in Fig. 8.

4. Pattern specifications for the CICO domain pattern

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

- (1) Items that can be checked out and in have unique identifiers.
- (2) Items are maintained in one or more collections (e.g., a library system can have a collection of journals, a collection of references, and a collection of general books).

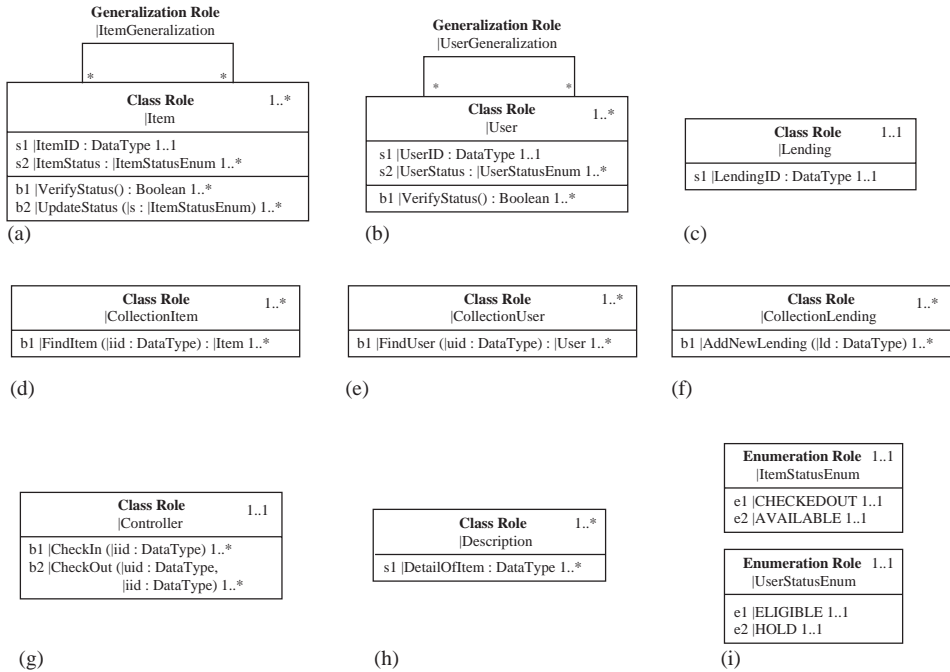


Fig. 10. CICO role hierarchies: (a) item role hierarchy; (b) user role hierarchy; (c) lending role; (d) collectionItem role; (e) collectionUser role; (f) collectionLending role; (g) controller role; (h) description role; (i) enumeration roles.

hierarchies. Fig. 10 also shows the CICO domain pattern classifier roles with their feature roles. The structural roles are labeled using strings of the form “s#” (e.g., s1, s2,...), where “s” denotes structural feature roles. The behavioral roles are labeled “b#”. These labels can be used in application models to mark features that are bound to the roles.

The following is an overview of the properties defined in each classifier and hierarchy role shown in Fig. 9. Constraint templates for some of the more interesting feature roles are also given. Multiplicity and other properties specified in association-ends roles are not given in this paper.

Item role hierarchy (Fig. 10(a)). *Item* role has *ItemID*, *ItemStatus*, *VerifyStatus* and *UpdateStatus* feature roles. The *ItemID* role specifies a structural feature that uniquely identifies items, and the *ItemStatus* role specifies a structural feature that is used to indicate whether an item is checked in or checked out. The types associated with structural features that conform to *ItemID* must be instances of the *DataType* metaclass. The types associated with structural features that conform to *ItemStatus* must conform to *ItemStatusEnum*.

The *VerifyStatus* behavioral role specifies behavior that returns true if the item is available for checkout and false otherwise.

context |Item :: |VerifyStatus (): Boolean
pre : true
post: if |ItemStatus = |AVAILABLE then result = true
 else result = false

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

context|Item :: |UpdateStatus (|s : |ItemStatusEnum)
pre : true
post: |ItemStatus = |s

User role hierarchy (Fig. 10(b)): The *User* role has *UserID*, *UserStatus*, and *VerifyStatus* feature roles where *VerifyStatus* specifies behavior that returns true if a user can checkout an item and false otherwise.

Lending role (Fig. 10(c)): The *Lending* role characterizes classes describing objects that record item checkout details.

CollectionItem role (Fig. 10(d)): The *CollectionItem* characterizes classes representing groups of items. It includes a behavioral role *FindItem* specifying behavior that locates an item given the item's ID.

CollectionUser role (Fig. 10(e)): The *CollectionUser* role characterizes classes representing collections of users. It includes a behavioral role *FindUser* specifying behavior that locates a user given the user's ID.

CollectionLending role (Fig. 10(f)): The *CollectionLending* role characterizes classes describing objects that maintain a collection of checkout records. It has a *AddNewLending* behavior role that specifies behavior that adds new lending information to the collection.

Controller role (Fig. 10(g)): The *Controller* characterizes classes that manage the checkin and checkout of items. The role includes two behavioral roles, *CheckIn* and *CheckOut* representing checkin and checkout behaviors, respectively. Constraint templates for the *CheckIn* and *CheckOut* are given below:

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

CheckIn Postcondition: The *FindItem* operation is called. If the operation returns the item and if the item's status indicates that the item has been checked out, the

item's status is changed to indicate it is now available for checkout by calling the item's *UpdateStatus()* operation.

```

context |Controller :: |CheckOut (|uid:|ID, |iid:|ID)
  pre : true
  post: let itemMessage: OclMessage =
    |CollectionItem^^|FindItem(|iid) → any(true),
    userMessage: OclMessage = |CollectionUser^^|FindUser(|uid)
    → any(true)
  in
    userMessage.hasReturned() and userMessage.result() = user
    and user@pre.|VerifyStatus() = true
    and itemMessage.hasReturned() and itemMessage.result() = item
    and item@pre.|VerifyStatus() = true and |CollectionLending → exists
    (lendinfo|lendInfo.oclIsNew()
    and lendinfo.|User = user and lendinfo.|Item = item)

    and item^|UpdateStatus(|CHECKEDOUT)

```

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

It is important to keep in mind that the above specifications describe patterns and no attempt is made to specify complete CICO behavior. The above templates can be used to create initial specifications for checkin and checkout operations. These specifications can then be extended by designers to meet application-specific requirements not captured by the pattern. This is further discussed and illustrated in Section 5.

4.2. CICO IPSs

Fig. 11(a) shows an IPS for a checkin scenario. The item is found in the item collection, and its status is checked to determine whether it is checked out. If the item is checked out the status of the item is updated.

Fig. 11(b) shows an IPS for a checkout scenario. An instance of a classifier that conforms to *Controller* invokes *FindUser* with the user ID *uid* to obtain a matching user “*u*” from a collection of users. The status of the user is verified, and if the user is allowed to checkout the item the requested item is retrieved. The status of the item is queried to determine if it can be checked out. If the item can be checked out, a lending record is created. The record is then added to a collection of lending records (*CollectionLending*). The status of the item is updated to indicate that it has been checked out.

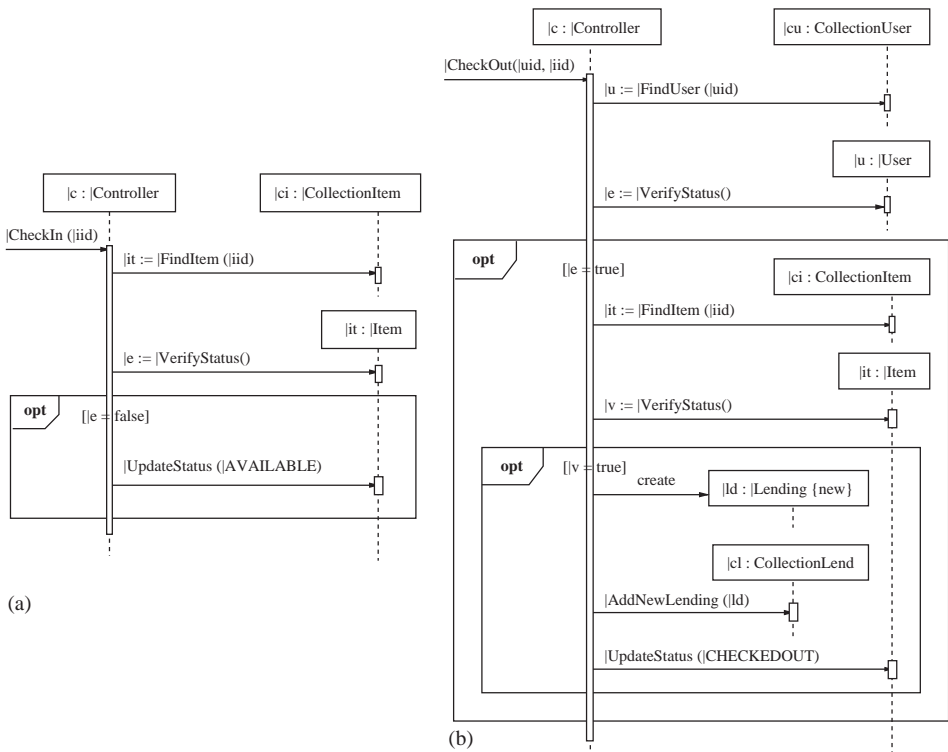


Fig. 11. IPSs for (a) checkin and (b) checkout scenarios.

5. Applying the CICO domain pattern

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

Appendix provides another example of an application model obtained from the CICO pattern.

5.1. The library system: class diagram

The library system described in this section has a collection of items referred to as *copies*. A copy can be a book or a multimedia item. Users that can checkin and checkout copies are referred to as *members*. Fig. 12 shows a class diagram obtained

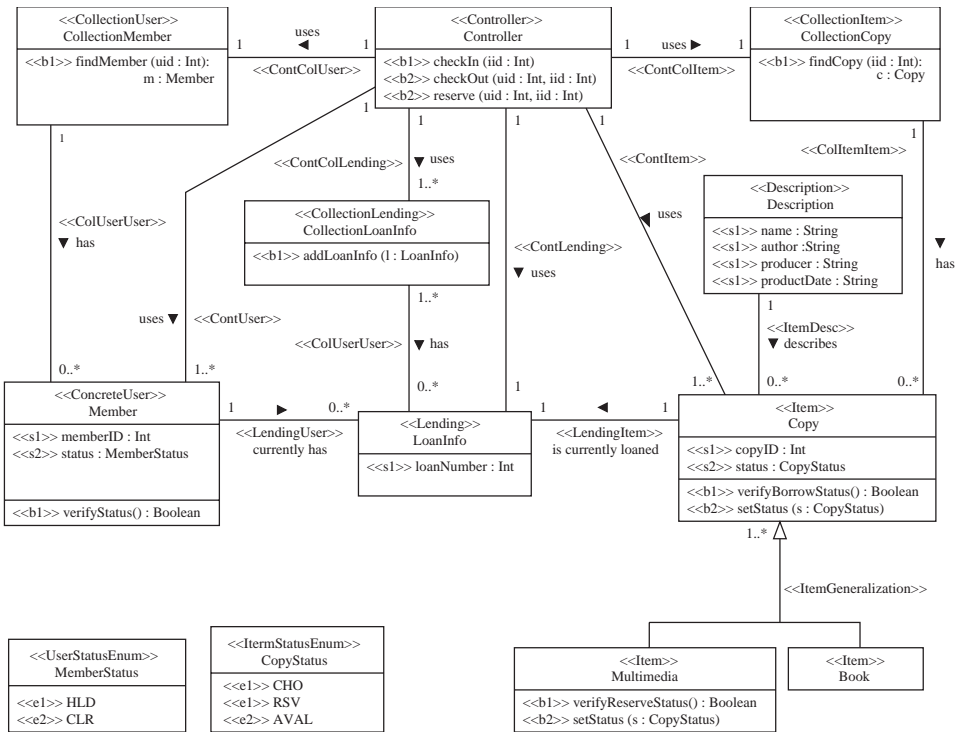


Fig. 12. A CICO conformant library class diagram.

from the CICO SPS. The stereotypes on the diagram elements indicate the CICO SPS roles they are bound to. Some of these bindings are described below (in the following, the symbol \mapsto is to be read “binds to”):

- The *Controller* role in the SPS is bound to the *Controller* class in the Library class diagram. The *CheckOut* role is bound to an operation that checks out copies (*checkOut*) and an operation that reserves copies (*reserve*). The bindings indicate that the *reserve* operation is intended to behave as specified by the *CheckOut* role. For this application, only multimedia copies can be reserved.
- The *Copy* hierarchy is obtained from the *Item* hierarchy defined in the CICO SPS. For example, $Copy \mapsto Item$, $Multimedia \mapsto Item$, and $Book \mapsto Item$ are bindings that produce the classes in the *Copy* hierarchy. The two generalization relationships shown in Fig. 12 are bound to the *ItemGeneralization* role. The *copyID* attribute is the only one that plays the *ItemID* role as required by the role’s binding multiplicity (1..1).
- The *VerifyStatus* feature role in the *Item* role is bound to *verifyBorrowStatus* in the *Copy* class and to *verifyReserveStatus* in the *Multimedia* class. The

In addition to the properties specified in the CICO pattern, the library system is required to (1) track lending history, (2) record reservations, (3) maintain contact information on members, (4) record the date a copy is checked out, (5) support adding and removing members and copies, and (6) support removal of lending information from the system. Additional class diagram elements are needed to address the above requirements. Fig. 13 shows the completed library system class diagram (additional elements are shown in bold typeface). Some of the elements added to the diagram are described below:

- The “has loan history” and “has loaned” associations are added to support tracking of lending history.
- The *Reservation* class and attached associations are added to support recording of reservations.
- The attributes *name* and *address* in *Member* are added because this information is used in the application to support activities that require contacting the member.

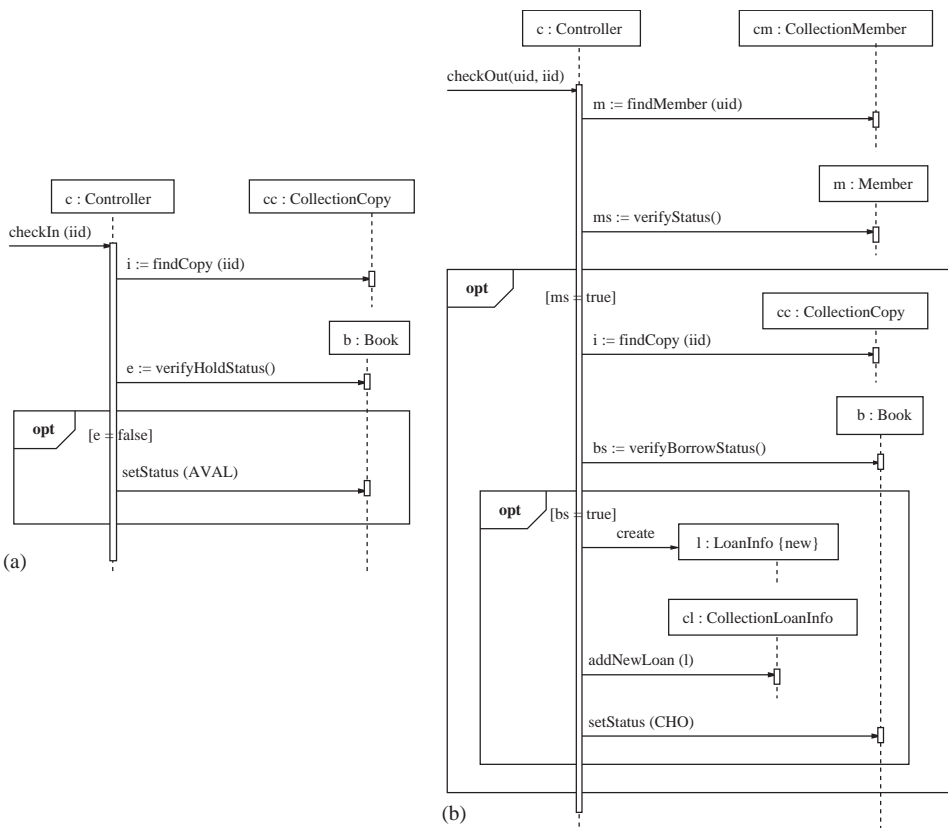


Fig. 14. CICO conformant library scenarios: (a) sequence diagram for the library checkin scenario; (b) sequence diagram for the library checkout scenario.

5.2. The library system: sequence diagrams

Sequence diagrams can also be obtained from the CICO IPSs by binding sequence diagram model elements to roles. The class and feature bindings are the same as those used to produce the class diagram. Fig. 14 shows sequence diagrams obtained from the CICO IPSs in Fig. 11.

6. Related work

Early work on domain-specific languages [9] tended to focus on providing language interfaces for assembling code components into programs. These languages focused on downstream development phases (detailed design specification and implementation in code). Domain-specific design specification and architectural languages have begun to appear (e.g., see [10]). We are not aware of any approaches that allow developers to specify reusable static and behavioral UML models and use them to develop applications.

Other forms of reusable experiences packaged for vertical reuse are frameworks [11] and domain-specific architectures (e.g., see [12–15]). There is a considerable body of work on domain engineering processes and domain modeling notations (e.g., see [2,5,13,15,16]). Our approach can complement the above efforts by providing a notation for defining domain-specific UML sublanguages as patterns.

Pattern languages for specifying Business Resource Management patterns have been developed (e.g., see [17,18]). In [17], Braga et al. use class diagrams to describe three patterns related to resource rental, trade and maintenance activities. The diagrams are used to stamp out class diagrams describing application-specific activities such as library service, medical attendance, video rental, and real estate rental. Their approach supports only specification of syntactic structural properties.

Other work on precisely defining pattern properties include those of Lauder and Kent [19], and Guennec et al. [20]. Lauder and Kent [19] use graphical constraint diagrams for precise visual presentation of patterns. Guennec et al. [20] 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 [21]. 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.

Object-based notions of roles have been developed by Dirk Riehle and the creators of the OORAM approach (e.g., see [22,23]). An object-based role specifies properties that objects in the run-time environment must have if they are to play the role. The RBML requires that roles be played by model elements (e.g., classes and associations), not by run-time application-specific objects.

that generates models from pattern specifications and user provided bindings has been developed. A user inputs bindings and selects properties for elements that satisfy metamodel-level constraints (e.g., association multiplicities). The tool then uses the inputs to generate a model.

Another prototype tool that we are developing allows users to directly use the sub-language defined by a domain pattern. The tool would look like a typical modeling tool, that is, it would have a symbol area and a drawing area. The symbol area includes symbols representing the domain concepts characterized by the domain pattern. In the case of the CICO pattern, there will be symbols for *Item* and *User* classes (or hierarchies). Modelers would select a symbol from the symbol area and drag it onto the drawing area. For example, dragging an *Item* class to the drawing area would result in an item class being displayed with slots for features that play the feature roles defined in the *Item* role. The metamodel (i.e., the specialized UML metamodel defined by the pattern) constrains how the symbols can be connected together in the drawing area.

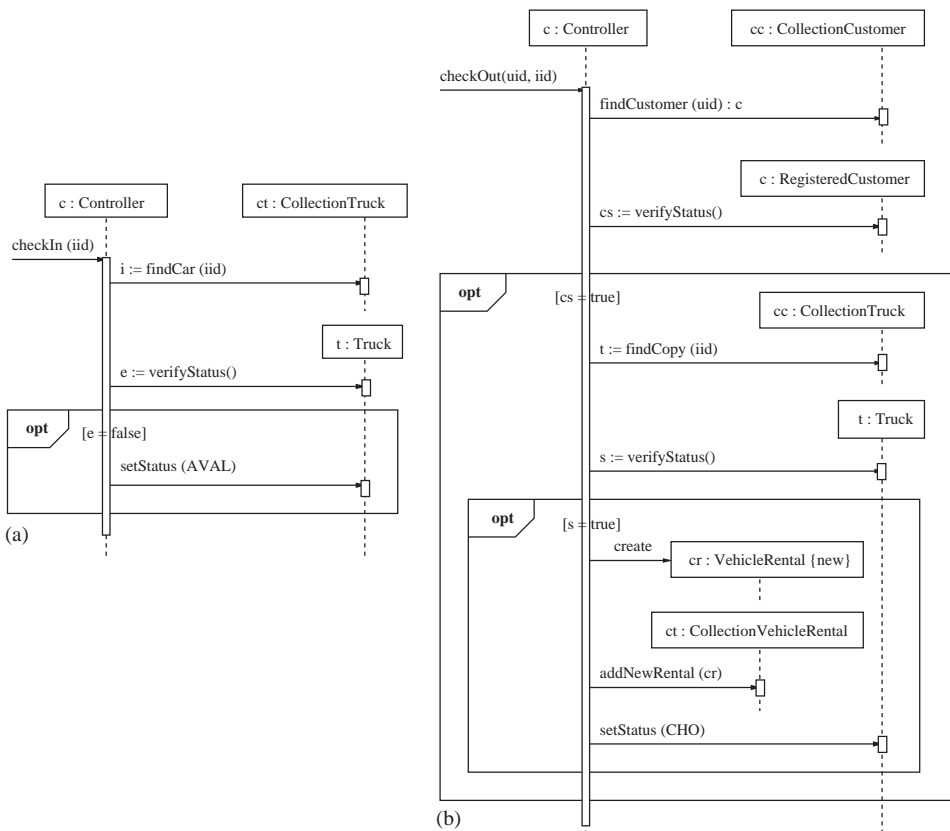


Fig. 16. CICO conformant car rental scenarios: (a) sequence diagram for the car rental checkin scenario; (b) sequence diagram for the car rental checkout scenario.

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

Appendix A. vehicle rental system

Fig. 15 shows a car rental class diagram created using the CICO SPS. The diagram describes a design in which customers rent vehicles. Two types of vehicles can be rented: trucks and leisure vehicles. Unlike the library example, the *Item* hierarchy is bound to a multi-level generalization hierarchy: the *Vehicle* class is specialized by *Truck* and *Leisure* and *Leisure* is further specialized by *Van* and *Sedan*.

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

Vehicle rental scenarios obtained from the CICO IPSs are shown in Fig. 16.

References

- [1] V.R. Basili, 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.
- [2] J.-M. Morel, J. Faget, The REBOOT Environment, in: Proceedings of the Second International Workshop on Software Reusability: Advances in Software Reuse, IEEE Computer Society Press, Lucca, Italy, 1993, pp. 80–88.
- [3] R. Prieto-Diaz, Status report: software reusability, IEEE Software 10 (3) (1993) 61–66.
- [4] The Object Management Group (OMG), Unified Modeling Language: Superstructure, Version 2.0, Final Adopted Specification, OMG, <http://www.omg.org>, August 2003.
- [5] G. Arango, R. Prieto-Diaz, Introduction and overview: domain analysis concepts and research directions, in: R. Prieto-Diaz, G. Arango (Eds.), Domain Analysis and Software Systems Modeling, IEEE Press, New York, 1991, pp. 9–32.
- [6] The Object Management Group (OMG), Unified Modeling Language, Version 1.4, OMG, <http://www.omg.org>, September 2001.
- [7] D. Kim, R. France, S. Ghosh, E. Song, A role-based metamodeling approach to specifying design patterns, in: Proceedings of 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC), Dallas, Texas, November 2003.
- [8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, MA, 1995.
- [9] D.S. Wile, J.C. Ramming, Special section: domain specific languages, IEEE Transactions on Software Engineering 25 (3) (1999) 289–290.

- [10] J. Gray, T. Bapty, S. Neema, J. Tuck, Handling crosscutting constraints in domain-specific modeling, *Communications of the ACM* 44 (10) (2002) 87–93.
- [11] G.F. Rogers, *Framework-Based Software Development in C++*, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- [12] S.T. for Adaptable Reliable Systems (STARS), STARS conceptual framework for reuse processes, Vol. 1: Definition, Version 3.0, Technical Report STARS-VC-A018/001/00, Unisys STARS Technology Center, October 1993.
- [13] M.L. Griss, Software reuse: from library to factory, *IBM Systems Journal* 32 (4) (1993) 1–23.
- [14] T. Lewis, L. Rosenstein, W. Pree, A. Weinand, E. Gamma, P. Calder, G. Andert, J. Vlissides, K. Schmucker, *Object Oriented Application Frameworks*, Manning Publication Co., Greenwich, CT, USA, 1995.
- [15] W. Tracz, L. Coglianese, P. Young, Domain-specific software architecture engineering process outline, *ACM SIGSOFT Software Engineering Notes* 18 (2) (1993) 40–49.
- [16] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-oriented domain analysis FODA: feasibility study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, CMU, 1990.
- [17] R.T.V. Braga, F.S.R. Germano, P.C. Masiero, A family of patterns for business resource management, in: *Proceedings of the Fifth Annual Conference on Pattern Languages of Programs (PLoP'98)*, Monticello, IL, USA, 1998, URL jerry.cs.uiuc.edu/plop/plopd4-submissions/plopd4-submissions.html.
- [18] R.T.V. Braga, F.S.R. Germano, P.C. Masiero, A pattern language for business resource management, in: *Proceedings of the Sixth Pattern Languages of Programs Conference (PLoP'99)*, Vol. 7, Monticello, IL, USA, 1999, pp. 1–34.
- [19] A. Lauder, S. Kent, Precise visual specification of design patterns, in: *Proceedings of ECOOP'98*, 1998, pp. 114–136.
- [20] A. Guennec, G. Sunye, J. Jezequel, Precise modeling of design patterns, in: *Proceedings of UML'00*, 2000, pp. 482–496.
- [21] A. Eden, *Precise specification of design patterns and tool support in their application*, Ph.D. Thesis, University of Tel Aviv, Israel, 1999.
- [22] T. Reenskaug, P. Wold, O.A. Lehne, *Working with Objects: The OORAM Software Engineering Method*, Manning/Prentice-Hall, Englewood Cliffs, NJ, 1996.
- [23] D. Riehle, T. Gross, Role model based framework design and integration, in: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, ACM Press, Vancouver, Canada, 1998, pp. 117–133.