A UML-Based Pattern Specification Technique

Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, Eunjee Song
Colorado State University, Fort Collins, CO 80523, USA

Abstract

Informally described design patterns are useful for communicating proven solutions for recurring design problems to developers, but they are inadequate for more rigorous use of design patterns. For example, they cannot be used as compliance points against which solutions that claim to conform to the patterns are checked. Pattern specification languages that utilize mathematical notation provide the needed formality but often at the expense of usability. In this paper we present a rigorous and practical technique for specifying pattern solutions expressed in the Unified Modeling Language (UML). The specification technique paves the way for the development of tools that support rigorous application of design patterns to UML design models. The technique has been used to create specifications of solutions for several popular design patterns. We illustrate the use of the technique by specifying Observer and Visitor pattern solutions.

Keywords: Design patterns, object-oriented models, pattern specification, UML

1 Introduction

A design pattern describes a family of solutions for a class of recurring design problems. Popular forms of design patterns consist of structured, informal descriptions of solutions for problems targeted by the patterns (e.g., see [1, 5, 17, 18]). The informal descriptions have proven to be effective at communicating design experience to developers, but they lack the formality needed to support rigorous use of design patterns. Precise specification of pattern

solutions enables the development of pattern-based development techniques and supporting tools that can be used to (1) systematically build solutions from pattern specifications (e.g., see [16]), (2) verify the presence of pattern solutions in designs (e.g., see [19]), and (3) systematically incorporate a pattern solution into a design (e.g., see [4]). Formal pattern specification languages that utilize mathematical notation (e.g., see [2, 13]) provide the concepts needed to precisely describe pattern solutions, but using them requires sophisticated mathematical skills. Pattern specification languages that are based on familiar software modeling concepts are more likely to be usable by software developers.

The pattern specification technique described in this paper supports rigorous specification of pattern solutions expressed in the UML. The UML is used for the following reasons:

- The UML is considered to be the *de facto* standard for object-oriented modeling, and there is a rapidly growing UML user base in industry. In this context, work that supports rigorous application of design patterns to UML models is relevant.

- The Object Management Group (OMG) is promoting an initiative called *Model Driven Architecture* (MDA) that supports the use of models as primary artifacts of development (see http://www.omg.org/mda). MDA is intended to raise the level of abstraction at which complex systems are developed. Technology that supports transformation of models is considered to be a key enabler of MDA. This has generated interest in developing tools to support the transformation of models using design patterns. Such tools require precise specification of pattern solutions expressed in a widely used modeling notation such as the UML [4].

The pattern specifications created by the technique are metamodels that characterize UML design models of pattern solutions. A pattern's metamodel is obtained by specializing the UML metamodel.

The remainder of this paper is organized in the following manner. In Section 2 we give an overview of the UML and its metamodel, and briefly discuss how the metamodel can be specialized. In Section 3 we describe the approach to specifying pattern solutions using a simple *Observer* pattern, and in Section 4 we illustrate the approach by using it to specify *Visitor* pattern solutions. In Section 5 we discuss the experience gained as a result of using the technique to specify popular design patterns, including patterns described by Gamma *et al.* [5]. In Section 6 we give an overview of related work on specifying design patterns. We conclude in Section 7 with an overview of our plans to further evolve this work.

2 Background

A UML design model consists of a number of diagrams, each describing a design view. In this paper, a pattern solution is described from two perspectives: the structural view is described by a class diagram and the interaction view is described by sequence diagrams. In this section we give an overview of UML class and sequence diagrams and outline how the UML metamodel can be specialized.

2.1 UML Diagrams

Fig. 1 shows examples of the types of UML diagrams used in this paper. The diagrams used in this paper conform to the UML 2.0 standard (see http://www.omg.org/uml). A UML class diagram describes classifiers (e.g., classes and interfaces) and relationships between classifiers. A class is a classifier that characterizes a family of objects in terms of attributes and operations that are common to the objects. Associations between classes specify links between class objects. The ends of associations, referred to as *association-ends* in this paper, have properties such as multiplicity.

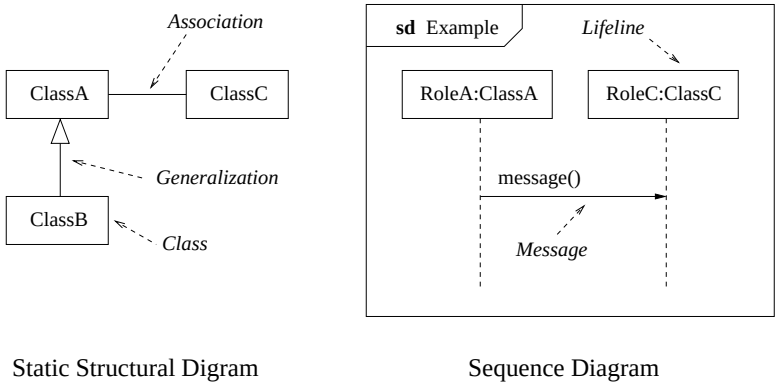


Figure 1: Overview of UML Class and Sequence Diagrams

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.

2.2 Specializing the UML Metamodel

The UML metamodel characterizes valid UML models. It consists of a class diagram and a set of well-formedness rules that define the abstract syntax of the UML. Informal descriptions of semantics are also included in the metamodel. The metamodel class diagram consists of classes whose instances are UML model elements. For example, instances of the metamodel class *Association* are UML associations. Well-formedness rules that are not expressible in the metamodel class diagram are expressed using the *Object Constraint Language* (OCL) [20] where possible, and in natural language otherwise.

Fig. 2 shows a part of the UML metamodel class diagram (class attributes and multiplicities are not shown). The diagram states that UML classifiers (instances of $Classifier$) can have attributes (instances of $Property$) and operations (instances of $Operation$), and that

an association (instance of *Association*) has association-ends (instances of *Property*) that are connected to classifiers.

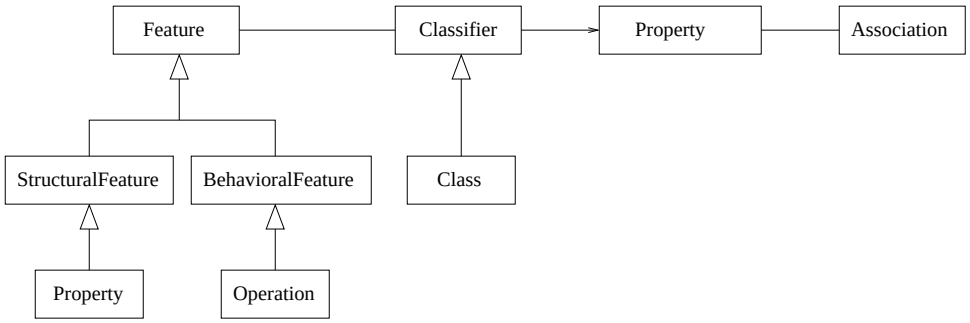


Figure 2: A Part of the UML Metamodel

The UML metamodel can be specialized to produce a restricted form of the UML metamodel that defines a proper subset of valid UML models. Specializing the UML metamodel to obtain a pattern specification involves the following:

- Specializing the abstract syntax by subtyping UML metamodel classes and by making the well-formedness rules more restrictive. The result is an abstract syntax for models describing pattern solutions.

- Defining parameterized OCL constraints, called constraint templates, representing constraints that must be expressed in models characterized by the specialized metamodel. The parameterized constraints capture semantic properties of patterns.

The result is a pattern metamodel that characterizes models describing structural and behavioral aspects of pattern solutions.

3 Specifying Pattern Solutions

A pattern specification consists of a *Structural Pattern Specification* (SPS) that specifies the class diagram view of pattern solutions, and a set of *Interaction Pattern Specifications* (IPSs) that specifies interactions in pattern solutions. The SPS is the core of a pattern specification. The IPSs are defined in terms of interaction participants specified by elements in the SPS. A UML model conforms to a pattern specification if its class diagram conforms to the SPS and the interactions described by sequence diagrams conform to the IPSs.

3.1 Structural Pattern Specifications (SPSs): An Overview

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

An SPS consists of a structure of *pattern roles* [9] (henceforth referred to as roles), where a role specifies properties that a UML model element must have if it is to be part of a pattern solution model. Formally, a role defines a subtype of a UML metamodel class. The metamodel class is called the *base* of the role. A role with a base B specifies a subset of instances of the UML metamodel class B. For example, a role that has the metamodel class *Association* as its base specifies a subset of UML associations. A UML model element *conforms to* (or *plays*) a role if it satisfies the properties defined in the role, that is, the element is an instance of the subtype defined by the role.

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

is a subtype of *Relationship* (e.g., *Association*, *Generalization*).

3.1.1 The SPS Notation

A classifier role is represented by a syntactic variant of the UML class symbol. The structure of a classifier role is shown in Fig. 3. The top compartment of a classifier role consists of

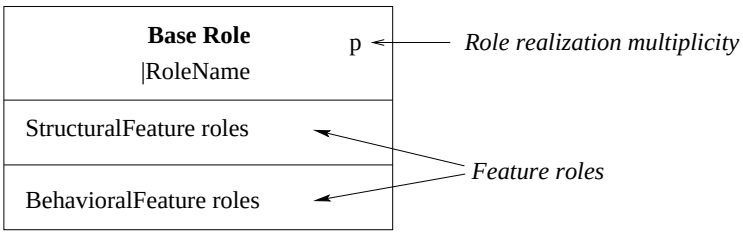


Figure 3: Structure of a Classifier Role

three parts:

- A label of the form **Base Role**, where **Base** is the name of the role's base (i.e., the name of a metamodel class).
- A declaration of the form $|RoleName$, where $RoleName$ is the name of the role. We use the symbol "|" to indicate that the following string is a role name.
- A *realization multiplicity*, p, that can restrict the number of classifiers playing the role in a conforming class diagram. The multiplicity can be omitted if the number of conforming classifiers is not constrained (i.e., the multiplicity is $*$).

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

- *StructuralFeature roles* specify properties represented by structural features of conforming classifiers. A StructuralFeature role can be played by an attribute or a query (i.e., a value-returning function with no side-effects).

- *BehavioralFeature roles* specify behavioral properties associated with conforming classifiers. A BehavioralFeature role can be played by an operation.

Each feature role is associated with a realization multiplicity that can constrain the number of features (e.g., attributes or operations) in a conforming classifier playing the feature role. A realization multiplicity with a lower bound of 0 (e.g., $*$) indicates that the feature may or may not be present in a conforming classifier (i.e., it is an optional feature). Examples of class roles are shown in Fig. 4.

A relationship role is represented by a syntactic variant of the UML association symbol. Like classifier roles, each relationship role is associated with a label that indicates the base of the role. Association roles also have association-end roles that define subtypes of the UML metamodel *Property* class (see Section 2.2). Association-end roles specify multiplicity, navigability, and other properties associated with conforming association-ends. An association-end role is also associated with a realization multiplicity that can constrain the number of association-ends playing the role in a conforming model. The realization multiplicity for an association role is inferred from the realization multiplicities of its association-end roles and thus they are not shown in the SPSs presented in this paper. An example of an association role is shown in Fig. 4.

Roles with realization multiplicities that have lower limits greater than 0 (e.g., 1..*) are referred to as *mandatory* roles. A conforming model must have models elements that conform to these roles. Both *Subject* and *Observer* in Fig. 4 are mandatory classifier roles. Roles that have realization multiplicities with lower limits that are 0 are referred to as *optional* roles. An SPS must have at least one mandatory role. If all SPS roles are optional then the SPS metamodel characterizes all valid UML class diagrams and thus is not a good discriminator.

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

straints. Examples of metamodel-level constraints are given in Section 3.1.2.

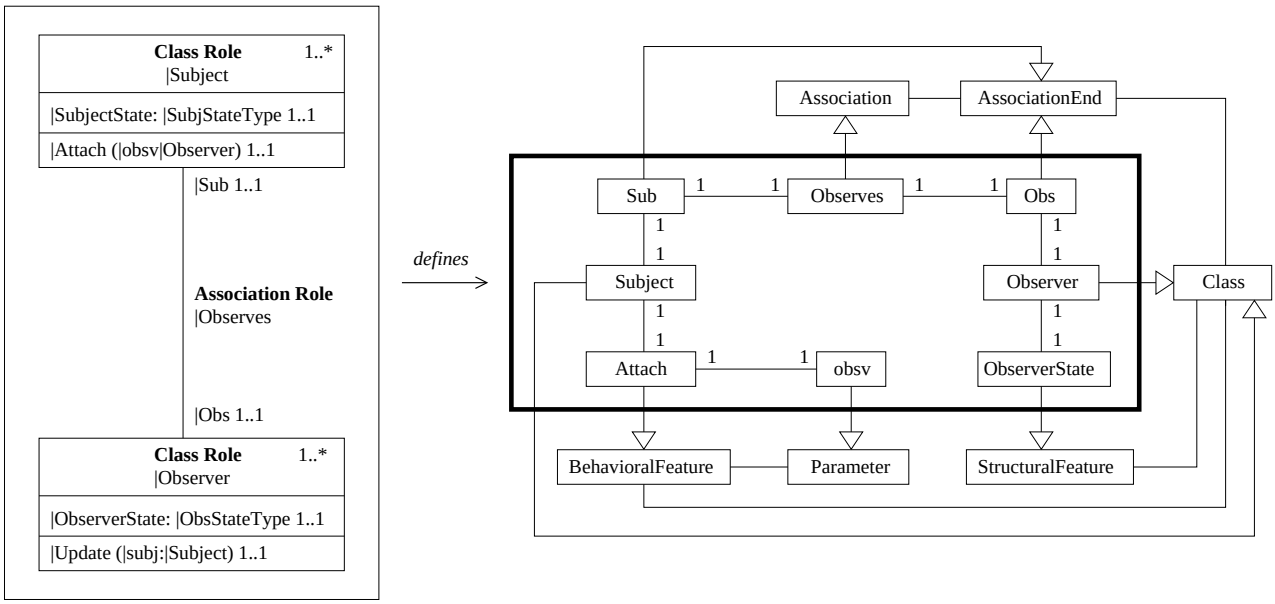
Semantic pattern properties are expressed as constraint templates in an SPS. For example, constraint templates are used to constrain the form of specifications for operations that conform to BehavioralFeature roles. Constraint templates are described in more detail in Section 3.3. Metamodel-level constraints and constraint templates are defined separately from the SPS role structure to avoid cluttering the diagram.

3.1.2 An SPS Example

Fig. 4(a) shows a partial SPS that specifies solutions of a restrictive variant of the *Observer* pattern [5] (metamodel-level constraints, constraint templates, and some feature roles are not shown). In this variant of the pattern, there can be one or more observer classes and one or more subject classes. An observer class must have only one *Observes* association with a subject class and a subject class must have only one *Observes* association with an observer class.

The SPS in Fig. 4(a) consists of two class roles, *Subject* and *Observer*, and an association role, *Observes*. The roles define subtypes (specializations) of classes in the UML metamodel, as shown in Fig. 4(b) (not all specializations are shown). For example, the *Observer* role defines a subtype of *Class* called *Observer* in the metamodel (see Fig. 4).

The class roles shown in Fig. 4 indicate that conforming class diagrams must have at least one class that conforms to the *Subject* role (as indicated by the $1..*$ realization multiplicity in the role), and at least one class that conforms to the *Observer* role. A class that conforms to the *Subject* role (referred to as a *Subject* class) must have exactly one structural feature (e.g., an attribute or query) that conforms to the *SubjectState* role and exactly one operation that conforms to the *Attach* role. A class that conforms to the *Observer* role must have exactly one structural feature that conforms to the *ObserverState* structural feature role,



(a) Example of an SPS

(b) A Partial View of the Specialized UML Metamodel

Figure 4: Partial Views of an Observer Pattern SPS and its Metamodel

and one operation that plays the *Update* BehavioralFeature role.

The association role *Observes* specifies associations between *Subject* and *Observer* classes. Each conforming association must have one association-end connected to a *Subject* class and the other association-end connected to an *Observer* class. In a conforming class diagram, the association-end connected to a *Subject* class must conform to the *Sub* role and the association-end connected to an *Observer* class must conform to the *Obs* role. The realization multiplicity on the *Sub* role specifies that a *Subject* class must be part of only one *Observes* association. Similarly, an *Observer* class must be part of only one *Observes* association.

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

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

In the above, *Subject* is the *Class* subtype (subclass) defined by the role, *isAbstract* is an attribute inherited from the metamodel class *Class* and *self* refers to an instance of the *Subject* subtype (i.e., a *Subject* class). A similar constraint is associated with the *Observer* role.

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

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

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

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

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

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

3.2 Establishing Structural Conformance to an SPS

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

- Bind models elements to roles: Model elements are bound to the roles they are intended to play.

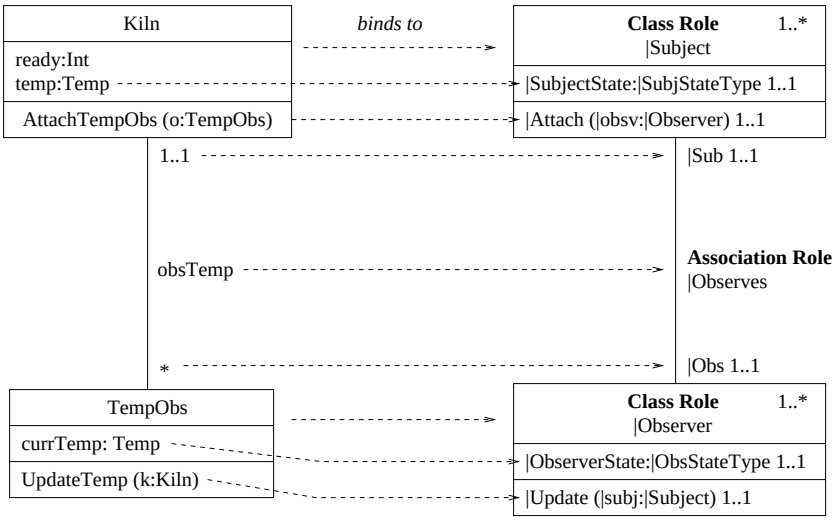
- Check compliance with classifier role realization multiplicities: This involves checking that the number of classifiers bound to a classifier role satisfy the realization multiplicities associated with the role, and checking that mandatory roles have classifiers bound to them.

- Establish structural conformance of classifiers to their bound roles: For each classifier bound to a classifier role this requires establishing that (1) the classifier satisfies the metamodel-level constraints associated with the classifier role, (2) the features bound to feature roles in the classifier role satisfy the realization multiplicities of the feature roles, and that (3) the mandatory feature roles have features bound to them.

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

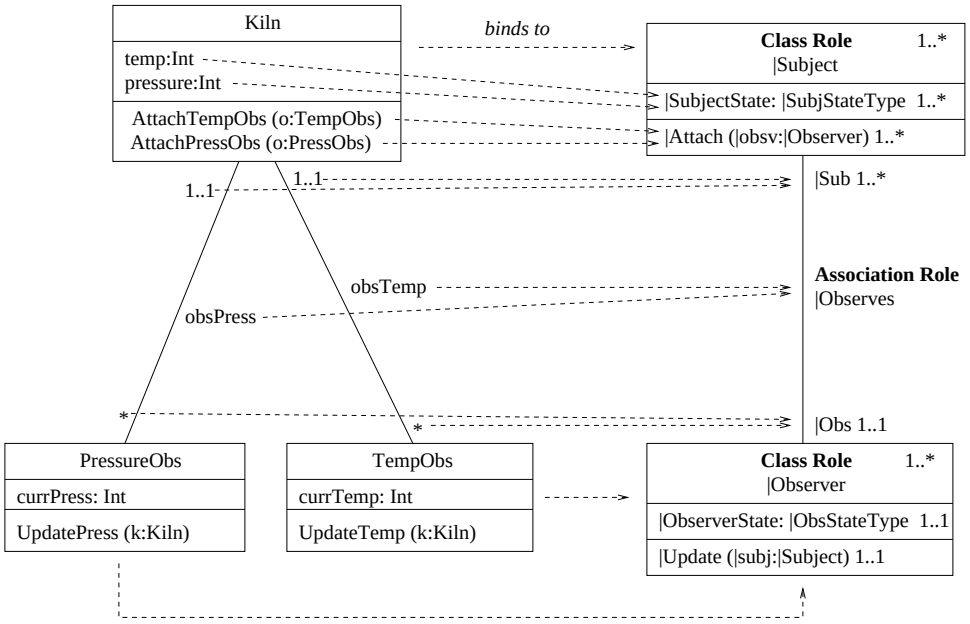
A class diagram that structurally conforms to the *Observer* pattern SPS, with respect to a binding, is shown in Fig. 5(a). The bindings are indicated by the dashed lines between the class diagram and the SPS in Fig. 5 (e.g., *Kiln* is bound to the *Subject* role). The class *Kiln* describes kiln objects whose temperatures are monitored by *TempObs* objects.

A partial view of a less restrictive variant of the *Observer* pattern and a conforming class diagram are shown in Fig. 6. The SPS shown in Fig. 6(b) specifies class diagrams in which *Subject* classes can have one or more structural features that can be monitored and can be part of one or more associations connected to *Observer* classes. The *Observer* pattern variant shown in Fig. 4 is a specialization of this less restrictive pattern variant, that is, the



(a) A Conforming Class Diagram (b) Observer Pattern Specification

Figure 5: A Structurally Conforming Observer Class Diagram



(a) A Conforming Class Diagram (b) Pattern Specification for a Variant Observer Pattern

Figure 6: A Partial SPS for a Variant of the Observer Pattern and a Conforming Class Diagram

set of class diagrams characterized by the SPS in Fig. 4 is a proper subset of the set of class diagrams characterized by the SPS shown in Fig. 6(b).

3.3 Specifying Semantic Pattern Properties in an SPS

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

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

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

pre: true

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

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

The *Subject* role is also associated with the following BehavioralFeature roles (these roles are not shown in Fig. 4(a)):

- **Detach** specifies behaviors that remove observers from subjects;

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

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

context |Subject::|Detach(|obsv:|Observer)

pre: self.|Obs → includes (|obsv)

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

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

pre: true

post: result = |SubjectState

context |Subject::|SetState(|newState:|SubjStateType)

pre: true

post: |SubjectState = |newState

The operation template associated with the *Update* feature role in *Observer* is given below:

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

pre: true

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

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

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

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

context |Subject

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

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

3.4 Establishing Full Conformance to an SPS

A class diagram fully conforms to an SPS, with respect to a binding of model elements to roles, if (1) it structurally conforms to the SPS (see Section 3.2), and (2) the semantic properties expressed by constraints in the class diagrams (e.g., operation specifications and class invariants) conform to the constraint templates in the SPS. Establishing that the semantic properties expressed in a class diagram conform to constraint templates in an SPS involves (1) instantiating the constraint templates using the role bindings, and (2) establishing that the constraints given in the class diagram refine the instantiations of the constraint templates.

The result of instantiating a constraint template is an application-specific OCL expression of the properties described by the constraint template. For example, instantiating the

property template given in Section 3.3 using the binding shown in Fig. 5 results in the following constraint:

context Kiln

obsTemp → forAll(currTemp = temp)

The identity function plays the role of *Function* in the property template. The class diagram shown in Fig. 5 must have a constraint that implies the above instantiation if it is to fully conform to the *Observer* SPS. In general, a class diagram that fully conforms to an SPS containing property templates must have constraints that imply instantiations of the property templates.

Instantiating the *Attach* operation template using the binding shown in Fig. 5 produces the following:

context Kiln::AttachTempObs(tobs: TempObs)

pre: self.TempObs → excludes(tobs)

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

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

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

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

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

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

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

As an example, consider the following pre- and postcondition for the *AttachTempObs*

operation shown in Fig. 5:

context Kiln::AttachTempObs(tobs: TempObs)

pre: self.TempObs → excludes(tobs)

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

ready = ready@pre + 1

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

self.TempObs → excludes(tobs) and

self.TempObs = self.TempObs@pre → including(tobs) and

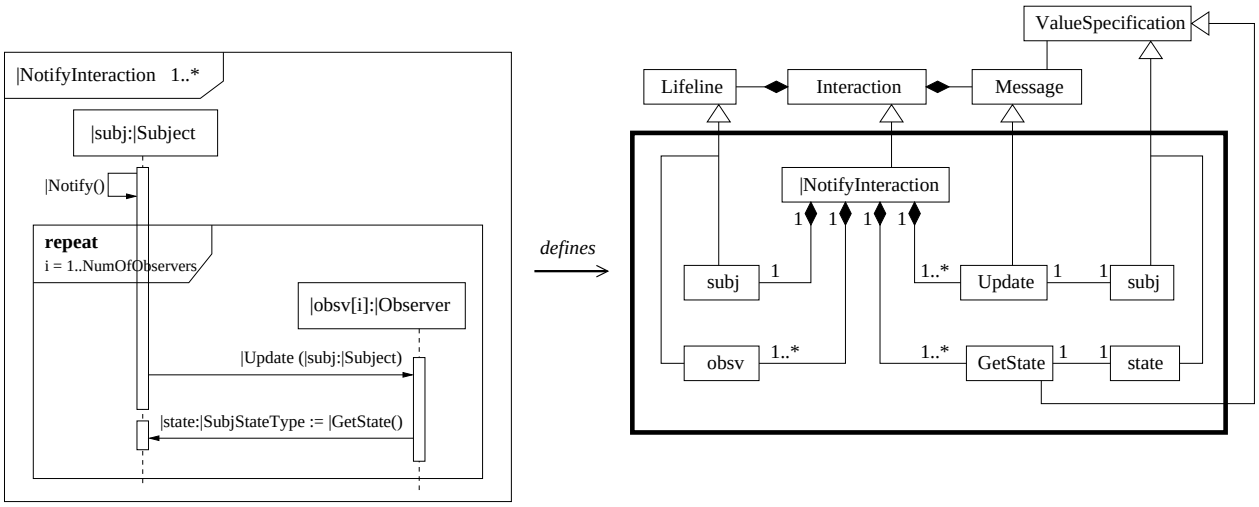
ready = ready@pre + 1 ⇒ self.TempObs = self.TempObs@pre → including(tobs)

Automated support for structural conformance checking is possible: mechanisms that check conformance of UML models to the abstract syntax defined by the UML metamodel can be extended to support well-formedness checks for patterns as defined by SPSs. Tools that can mechanically discharge most proof obligations are not likely to appear in the near future, but it is possible to build a tool that generates proof obligations that can then be discharged by humans.

3.5 Interaction Pattern Specifications (IPSs)

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

Fig. 7(a) shows an IPS that describes the pattern of interactions between a subject and its observers initiated by the invocation of the subject's $Notify$ operation. The expression $|subj : |Subject$ indicates that the lifeline role *subj* is played by an instance of a *Subject* class



(a) Example of an IPS

(b) A Partial View of the Specialized UML Metamodel

Figure 7: An IPS for the Observer Pattern and a Partial View of its Specialized UML Metamodel

(i.e., a class that conforms to the *Subject* role defined in the *Observer* SPS). The lifeline role $obsv[i]$ is played by the i^{th} observer in the set of observers attached to the subject playing the *subj* role. The **repeat** fragment in the IPS indicates that the enclosed interaction is repeated for each observer attached to the subject playing the *subj* role. $NumOfObservers$ is the number of observers attached to the subject. The **repeat** fragment is used to concisely represent parts of conforming interaction diagrams that have a common structure.

The IPS describes the following interaction pattern:

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

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

An IPS consists of an *interaction role* that defines a specialization of the UML metamodel

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

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

Part of the metamodel defined by the $NotifyInteraction$ IPS is given in Fig. 7(b). Lifeline roles define specializations of the $Lifeline$ class and message roles define specializations of $Message$.

A sequence diagram conforms to an IPS if the conforming interactions respect the relative order specified in the IPS. A sequence diagram that conforms to the $NotifyInteraction$ IPS is shown in Fig. 8. The subject participant in the interaction, s, has two observers attached

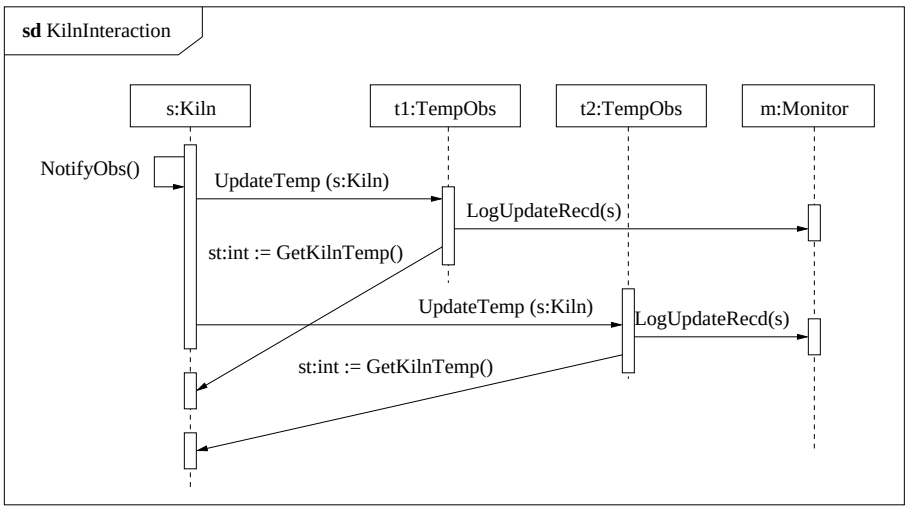


Figure 8: A Sequence Diagram that conforms to the Observer IPS

to it: $t1$ and $t2$. The sequence diagram has the interaction pattern specified in the IPS: the

relative order of the conforming messages, $NotifyObs$, $UpdateTemp$, and $GetKilnTemp$ is the same as the relative order specified in the IPS.

4 Specifying Visitor Pattern Solutions

A class diagram and a sequence diagram describing a typical Visitor pattern solution are respectively shown in Fig. 9 and Fig. 10. This solution is used by Gamma *et al.* [5] to describe the structure and behavior of Visitor pattern solutions. The model describes a

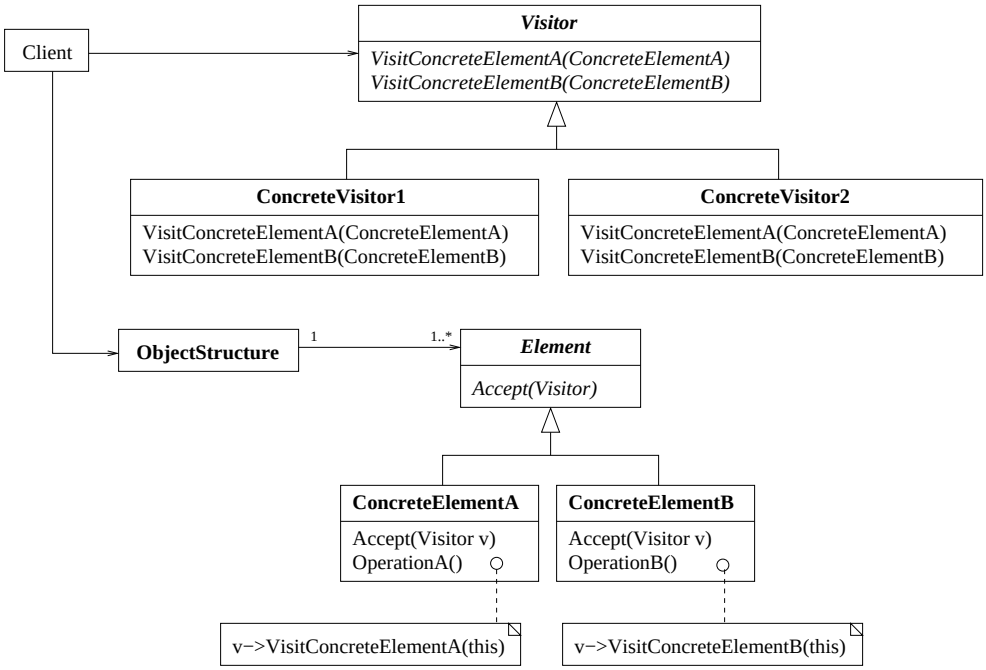


Figure 9: A Visitor Pattern Solution: Class Diagram

solution consisting of two types of visitors, $ConcreteVisitor1$ and $ConcreteVisitor2$, whose instances visit elements in an element collection (instances of $ObjectStructure$) consisting of two types of elements, $ConcreteElementA$ and $ConcreteElementB$.

The sequence diagram shown in Fig. 10 describes a typical interaction in which the $anObjectStructure$ object (an instance of $ObjectStructure$) calls the $Accept$ operation for

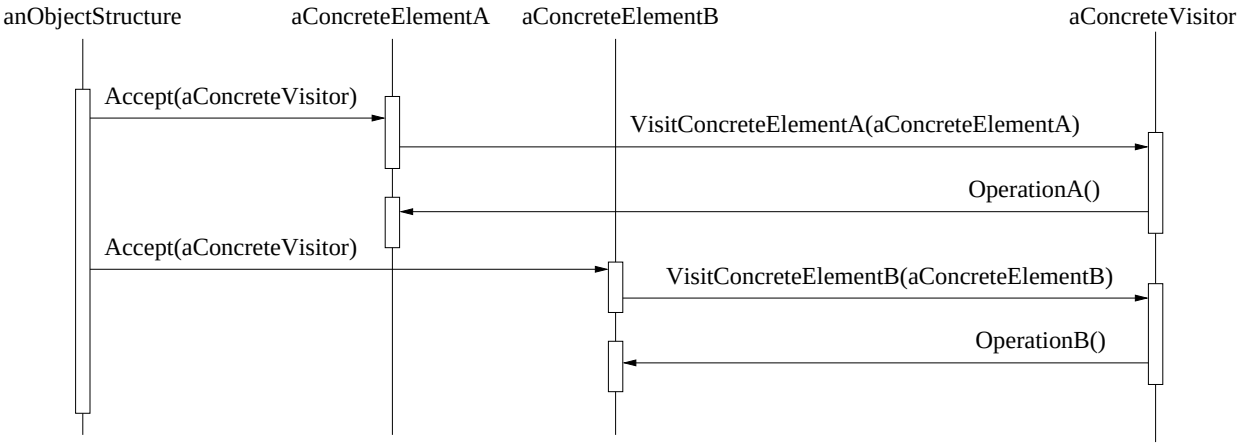


Figure 10: A Visitor Pattern Solution: A Sequence Diagram

each of its elements. The element collection consists of two elements - $aConcreteElementA$ is an instance of $ConcreteElementA$ and $aConcreteElementB$ is an instance of $ConcreteElementB$. Execution of the *Accept* operation in an element results in an operation call to the visitor passed in as an argument of the *Accept* operation. The visitor then performs an operation on the element.

A pattern specification for a variant of the Visitor pattern described by Gamma *et al.* [5] is presented in this section. It characterizes simple solution models involving flat sets of elements such as the one described above, and more complex solutions that involve composite element structures.

4.1 A Visitor SPS

Fig. 11(a) shows an SPS for the Visitor pattern. The SPS consists of two role hierarchies: The $Visitor$ and the *Element* role hierarchies. A role hierarchy is used to classify roles. For example, there are two types of $Visitor$ roles in the SPS: the $AbstractVisitor$ role must be played by classifiers that are either interfaces or abstract classes (referred to as abstract classifiers), and the $ConcreteVisitor$ role must be played by concrete classes.

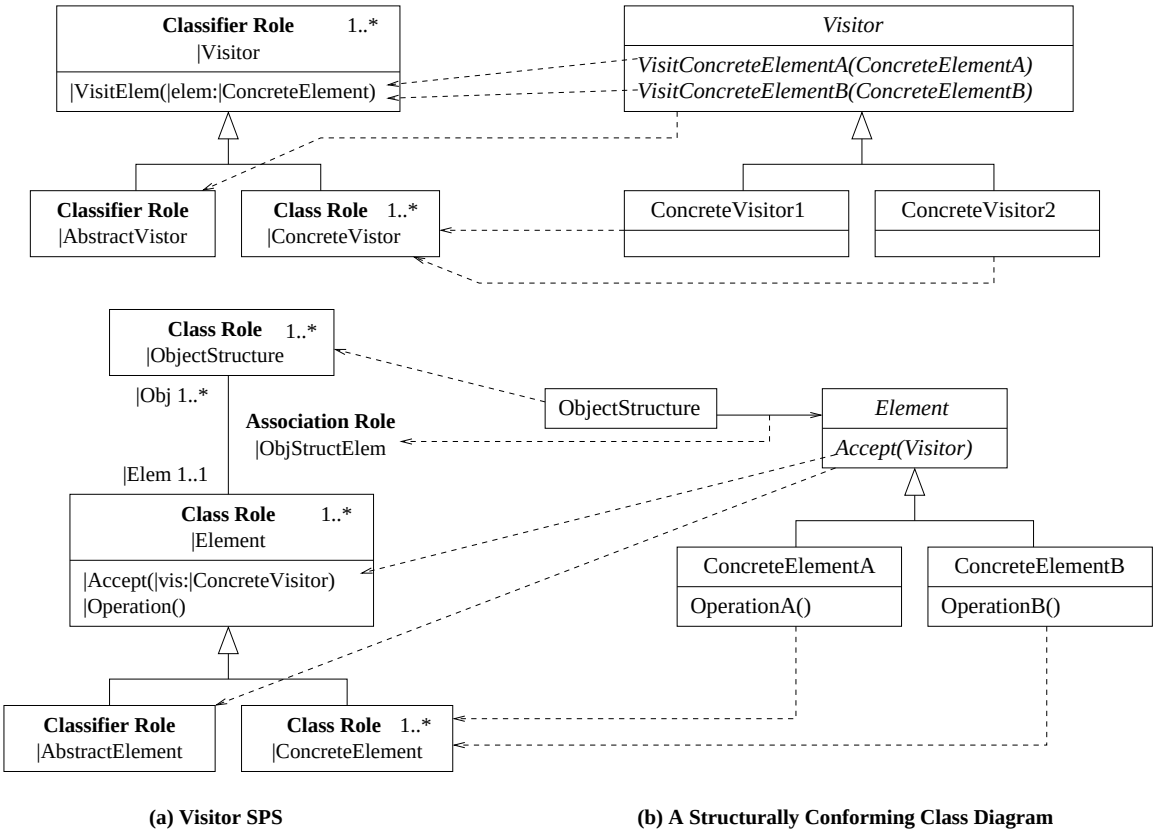


Figure 11: Visitor SPS

The metamodel-level constraints that express these constraints will be given later. A sub role in a role hierarchy inherits all the roles associated with its super role. For example, $ConcreteVisitor$ inherits the $VisitElem$ feature role defined in its super role, $Visitor$.

The Visitor SPS specifies class diagrams consisting of *Visitor* classifiers that can be abstract classifiers (e.g., interfaces, abstract classes) or concrete classes, and *ObjectStructure* classes associated with *Element* classifiers. There must be at least one class that plays the $ConcreteVisitor$ role, at least one class that plays the *concreteElement* role and at least one class that plays the *ObjectStructure* role in a conforming class diagram. Abstract visitor and element classifiers are optional.

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

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

context |AbstractVisitor **inv**: self.oclIsTypeOf(Interface)

or (self.oclIsTypeOf(class) and self.isAbstract = true)

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

context |AbstractVisitor **inv**: self.oclIsTypeOf(Class)

and self.isAbstract = false

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

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

An association-end that conforms to *Elem* must have a multiplicity of one or more:

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

The class diagram shown in Fig. 9 structurally conforms to the Visitor SPS with respect to the bindings shown in Fig. 11. A more complex class diagram that conforms to the Visitor SPS is shown in Fig. 12. This diagram includes an element class structure that describes composite elements. Stereotypes are used to indicate the roles played by model elements (this is an informal use of UML stereotypes - the stereotype syntax is used simply to visually mark elements). An instance of *CompositeEquipment* is a composite element structure that can also be an element in a larger element structure (i.e., it can be visited by an instance of the visitor class *PricingVisitor*). The *CompositeEquipment* thus plays two roles: *Element* and *ObjectStructure*.

The semantic properties expressed in the visitor pattern concern the interactions that take place in the context of the *VisitElem*, *Accept* and *Operation* behaviors. These properties are described by the pattern's IPS (see Section 4.2). There are no constraint templates associated with the Visitor SPS.

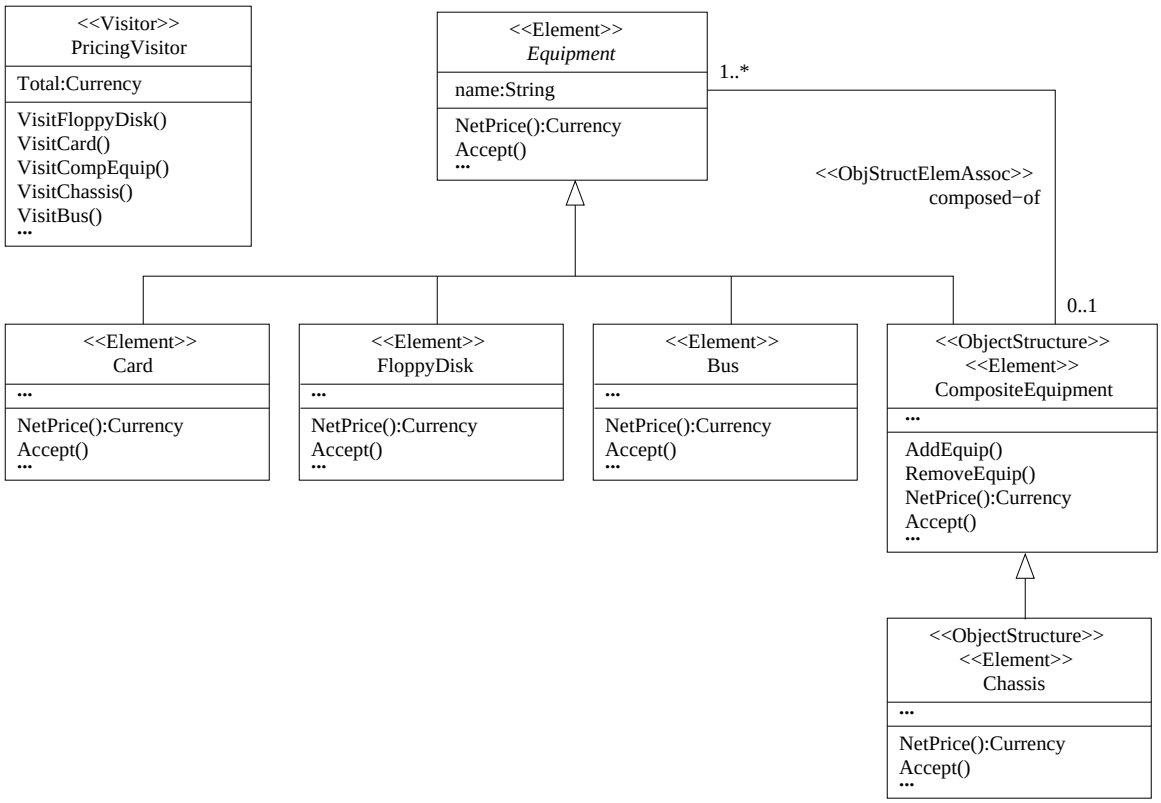


Figure 12: A More Complex Visitor Conformant Class Diagram

4.2 An IPS for the Visitor Pattern

Fig. 13 shows an IPS named *CompositeInteraction* that describes the interactions that take place when accessing a composite element structure with a visitor.

An instance of an *ObjectStructure* class plays the lifeline role *obj*. The i^{th} element of the object structure plays the lifeline role $elem[i]$. The interaction structure enclosed in the **repeat** fragment is repeated for each element in the object structure that plays the lifeline role *obj*. $NumOfElements$ is the number or elements associated with the object structure. An *Accept* message is sent to each element, $elem[i]$, in the object structure. If the element, $elem[i]$, is a composite element then the interaction structure defined in *CompositeInteraction* is recursively applied with $elem[i]$ becoming the *ObjectStructure*

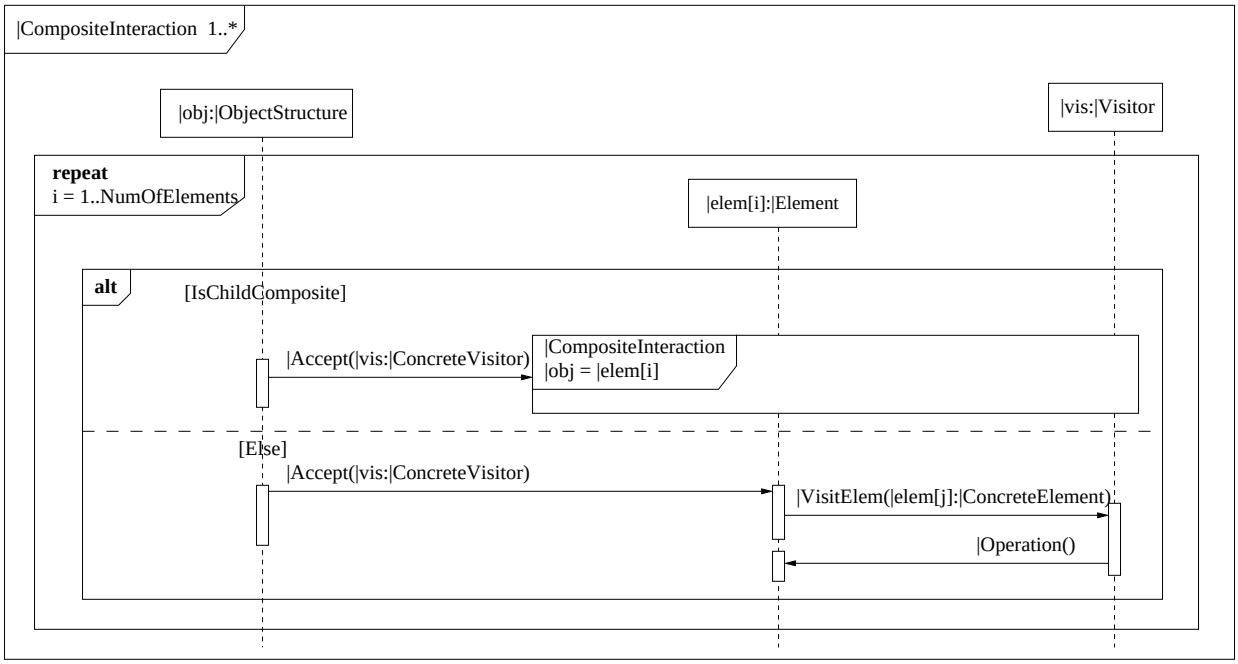


Figure 13: A Visitor IPS

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

The simple interaction diagram shown in Fig. 10 conforms to the Visitor IPS:

- The *anObjectStructure* lifeline conforms to the lifeline role *obj*,

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

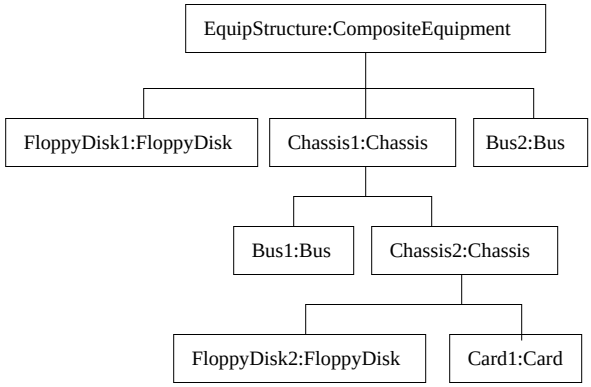


Figure 14: A Composite Part Structure

An example of a composite element structure described by the class diagram given in Fig. 12 is shown in Fig. 14. The composite element $EquipStructure$ consists of three elements: a primitive element $FloppyDisk1$, a primitive element $Bus2$, and a composite element $Chassis1$. The composite element $Chassis1$ consists of a primitive element $Bus1$ and a composite element $Chassis2$. Fig. 15 shows a Visitor sequence diagram that is based on the composite structure shown in Fig. 14.

The interaction sequence involving $FloppyDisk1$ and the sequence involving $Bus2$ have the structure specified by the *Else* part of the **alt** fragment. The interaction sequences involving $Chassis1$ has the structure specified by the $IsChildComposite$ region of the **alt**

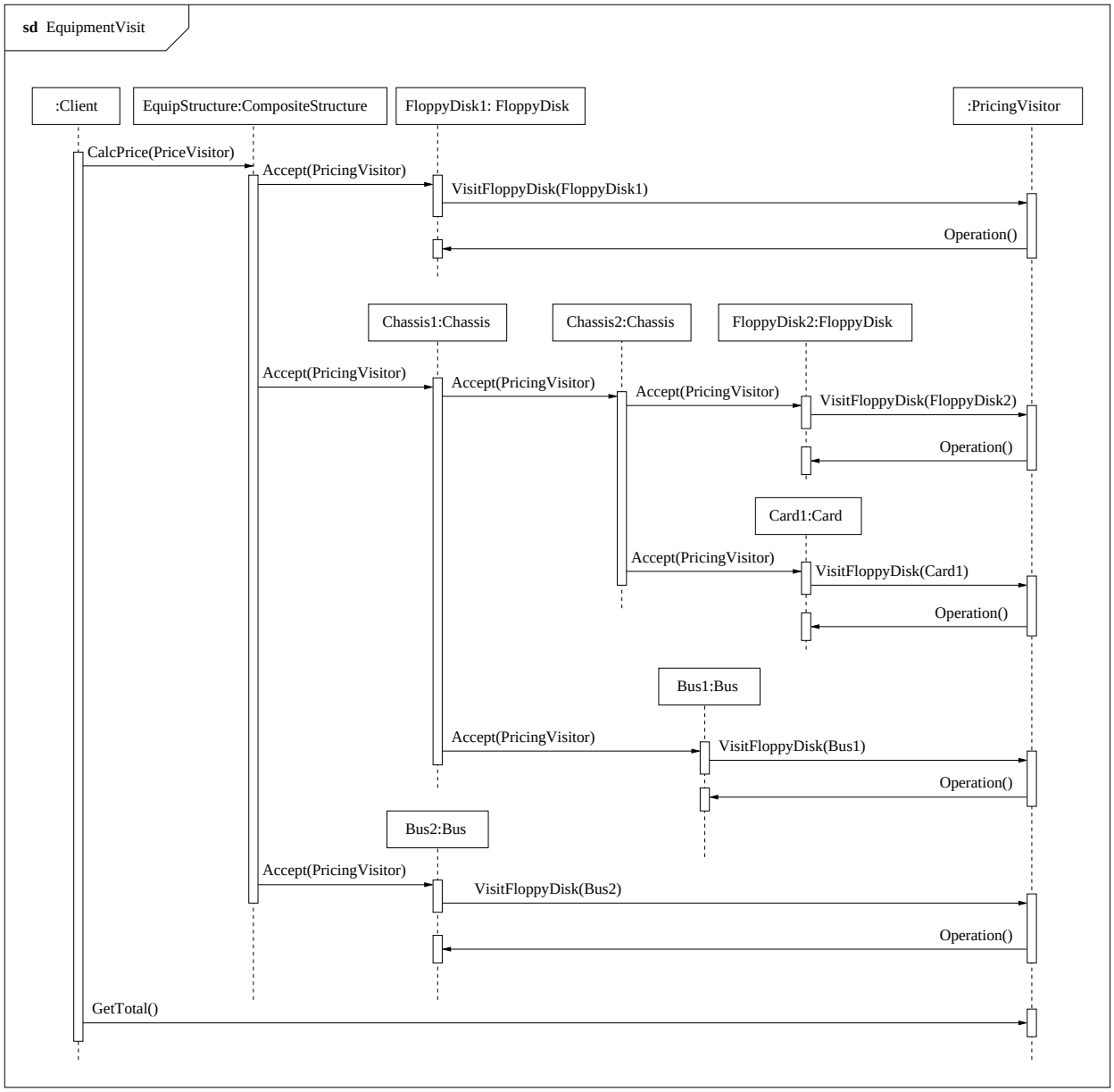


Figure 15: A Conforming Visitor Sequence Diagram

fragment. Establishing this involves recursively applying the $CompositeInteraction$ structure: $Chassis1$ becomes the $ObjectStructure$ lifeline, $Card1$ becomes the primitive element involved in the interactions described by the $Else$ region, and $Chassis2$ becomes the composite element involved in the interactions described by the $IsChildComposite$ region.

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

5 An Analysis of Early Experience

The goal of our work is to create a practical pattern specification technique that supports the use of patterns during design modeling. To achieve this goal we developed a pattern specification language that (1) uses the UML syntax to the extent possible, and (2) specifies patterns as specializations of the UML metamodel to support the use of patterns in UML system modeling. The UML was used as the syntactic base for the pattern specification language to make it easier for UML modelers to create, understand, and evolve pattern specifications, and to enable the use of UML modeling tools for creating and evolving pattern specifications.

To determine the extent to which the technique can be supported by UML modeling tools we developed a prototype tool for creating pattern specifications on top of the Rational Rose tool. The tool currently allows users to create SPSs, and to use the SPSs to generate conforming class diagrams. A problem was encountered when we tried to provide support for creating and instantiating constraint templates. The version of Rational Rose used did not support manipulation of OCL constraints. We are not aware of any commercially available UML modeling tool that fully supports the OCL. It is expected that this situation will change as tools that support UML 2.0 and OCL 2.0 become available.

To date we have developed full pattern specifications for the following design patterns [3]: Abstract Factory, Bridge, Decorator, Singleton, Observer, Composite, and Visitor. The pattern specification language has been presented to and used by graduate students in a software engineering course to develop specifications of design patterns. All the students

were familiar with the UML and had used the UML and patterns in previous courses. Our collective experience revealed the following about the pattern specification technique:

- The students were able to create specifications for patterns that did not involve the use of recursion in the interaction diagrams after two lectures on the pattern specification notation. It was expected that the metamodel level at which the patterns are described would create some difficulty in presenting the concepts to students not familiar with metamodeling concepts. The use of the UML notation to express roles helped in communicating the concepts to the students. Plans for designing and carrying out controlled experiments that more fully evaluate the ease of learning and using the technique are underway.

- Specifying patterns that describe behaviors localized in methods or in objects (e.g., see the Factory Method and the Iterator patterns) is problematic when the behaviors cannot be fully captured by operation templates or interaction diagrams. We are developing extensions to the pattern specification notation that will allow developers to specify solutions modeled by UML state machines and activity diagrams [10]. It is important to note that the pattern specification technique is restricted to descriptions of structure and behavior that can be expressed in the UML.

- Defining recursive behaviors (as required by the Visitor and Decorator patterns) was problematic using the UML 1.4 interaction diagrams, and the resulting IPSs were often not easy to understand. The UML 2.0 sequence diagram notation used in this paper offers a richer set of constructs, including constructs for packaging and referencing interactions. We had to modify the interpretation of these constructs to fulfill our needs (e.g., the **repeat** construct is an adaptation of the UML 2.0 loop construct), but we were able to maintain the sequence diagram "look and feel" in IPSs. The Visitor

IPS given in this paper illustrates how these constructs can be used to represent a range of behaviors concisely.

We have also used the pattern specification language to specify a large domain pattern for checkin-checkout systems [8, 11]. The pattern specifies a family of checkin-checkout systems (e.g., car rental and library systems). We used the pattern to develop UML designs for a library system and for a car rental system [8].

6 Related Work

There has been considerable work done on specifying design patterns using formal specification techniques (e.g., see [2, 13, 15]). Mikkonen [15] uses DisCo, a specification method based on the Temporal Logic of Actions [12], Eden [2] created a formal specification language called LePus to specify pattern properties, and Lano *et al.* [13] use an extension of their object calculus to specify patterns. The mathematically-based notation can make the tasks of creating and evolving the pattern specifications difficult for pattern authors.

Lauder and Kent [14] propose an approach to presenting patterns precisely and visually using graphical constraint diagrams. In their work, patterns are described in terms of three layers of models: *role-model*, *type-model* and *class-model*. A role-model describes the essential aspects of a pattern in terms of highly abstract state and behavior elements. A type-model is a refinement of a role-model in that it refines the role-model state and behavior elements in terms of types that abstractly specify domain realizations of the role-model. A class-model is a deployment of a type-model in terms of concrete classes. In their work, pattern realization is viewed as a refinement process in which a high-level pattern description is refined to a model realization. Establishing that a model conforms to a pattern (as expressed by a role-model) involves defining refinement relationships across the model levels. The authors use a

graphical form of constraints that is appealing but is not currently integrated with the UML and it is not clear how tools can support the notation.

Guennec *et al.* [6] use a UML metamodeling approach in which pattern properties are expressed in terms of *meta-collaborations* that consist of roles that are played by instances of UML metamodel classes. They point out deficiencies in the UML notion of role models and provide an alternative representation in terms of meta-collaborations that utilize a family of recurring properties initially proposed by Eden in [2]. Their work does not address (1) the specification of semantic pattern properties (e.g., behavioral properties), and (2) the characterization of UML behavioral models.

7 Conclusion and Further Work

The pattern specification technique described in this paper can be used as a base for tools that support creation and evolution of patterns, and rigorous application of design patterns to UML models. The tool-independent UML-based notation facilitates sharing of design patterns across UML modeling tools.

Specifying pattern solutions at the UML metamodel level allows tool developers to build support for creating patterns and for checking conformance to pattern specifications. This can be accomplished through interfaces that allow developers to access and specialize a tool's internal representation of the UML metamodel. This does not have to require direct modification of the internal metamodel: the specializations can be created and managed by a layer that sits on top of the UML metamodel layer in the tool. A new generation of UML tools that allow software developers to specialize the UML metamodel in limited ways are emerging (e.g., Rational XDE). These tools are expected to mature to the point where users can define pattern by specializing the metamodel as described in this paper.

A prototype tool that supports the creation of SPSs, and that uses pattern specifications to generate conforming class diagrams has been developed. A prototype pattern mining tool that utilizes pattern specifications to search for patterns in UML models generated from code is currently under development.

The popularity of the UML and the heightened interest in model-driven approaches to software development has raised interest in model transformations. Techniques and tools that support systematic and rigorous application of design patterns through model transformations can ease access to and reuse of design experience during software development. Our current work is concerned with using the pattern specification technique to support practical and rigorous pattern-based model transformation techniques [7].

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. Wiley, 1996.

[2] A. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, Israel, 1999.

[3] R. France, D. Kim, E. Song, and S. Ghosh. Role-Based Modeling Language (RBML) Specification V1.0. Technical Report 02-106, Computer Science Department, Colorado State University, Fort Collins, CO, June 2002.

[4] Robert France, Sudipto Ghosh, Eunjee Song, and Dae-Kyoo Kim. A metamodeling approach to pattern-based model refactoring. *IEEE Software*, 20(5), September/October 2003.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[6] A.L. Guennec, G. Sunye, and J. Jezequel. Precise Modeling of Design Patterns. In *Proceedings of UML'00*, pages 482–496, 2000.

[7] Sheena R. Judson and Robert B. France. Model transformations at the metamodel level. In *Proceedings of the Workshop in Software Model Engineering, UML'03 Conference*, October 2003.

[8] D. Kim, R. France, and S. Ghosh. A UML-Based Language for Specifying Domain-Specific Patterns. *Special Issue on Domain Modeling with Visual Languages, Journal of Visual Languages and Computing*, To be published in 2004.

[9] D. Kim, R. France, S. Ghosh, and 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.

[10] D. K. Kim, R. France, S. Ghosh, and E. Song. A UML-Based Metamodeling Language to Specify Design Patterns. In *Proceedings of Workshop on Software Model Engineering (WiSME) with UML 2003*, San Francisco, California, October 2003.

[11] Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. Using Role-Based Modeling Language (RBML) as Precise Characterizations of Model Families. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, Greenbelt, MD, December 2002. ACM Press.

[12] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[13] K. Lano, J. Bicarregui, and S Goldsack. Formalising Design Patterns. In *Proceedings of 1st BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computer Science*. Springer-Verlag, 1996.

[14] A. Lauder and S. Kent. Precise Visual Specification of Design Patterns. In *Proceedings of ECOOP'98*, pages 114–136, 1998.

[15] T. Mikkonen. Formalizing Design Patterns. In *Proceedings of the 20th International Conference on Software Engineering, pp. 115-124*, Kyoto, Japan, April 1998.

[16] B.-U. Pagel and M. Winter. Towards pattern-based tools. In *Proceedings of EuropLop*, Munchen, 1996.

[17] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.

[18] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[19] M. Sefikla, A. Sane, and R. H. Campbell. Monitoring Compliance of a Software System with its High-Level Design Models. In *Proceedings of the International Conference on Software Engineering, 1996*, 1996.

[20] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.