A UML-Based Metamodeling Language to Specify Design Patterns

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

Abstract

A design pattern describes a generic solution for problems that occur repeatedly. Current descriptions of design patterns describe solutions with graphical notation and complementing text. To encourage the use of design patterns, the development of pattern supporting tools is imperative. This requires design patterns to be specified precisely. There has been considerable work done on pattern specifications. They suffer from either complication or lack of formality and features. In this paper, we describe a metamodeling technique to specify design patterns that is formal and easy to understand and use. The technique uses the Role-Based Metamodeling Language *(RBML) based on the UML. The RBML is able to capture various design perspectives of patterns such as static structure, interactions, and state-based behavior. We give an overview of the RBML and demonstrate the technique using the* Iterator *design pattern. We show how pattern specifications can be used for checking pattern conformance using the model of a television remote control application.*

1 Introduction

Design patterns [7, 15] describe a generic solution for problems that occur repeatedly. Solutions are mostly described using a combination of diagrams and textual descriptions. The informal nature of these descriptions leads to complications in applying design patterns effectively into a design model. To encourage the use of design patterns we are investigating rigorous pattern specification techniques that pave the way for the development of tools that support practical and rigorous applications of design patterns. Tools in which codified forms of patterns are embedded can be used to verify the presence of patterns in a model, to incorporate a pattern into a model, or to generate models [3].

Formal approaches for specifying design patterns have been proposed [2, 11, 14]. A major drawback of the approaches is that they require mathematics and formal logic background which has discouraged pattern authors from adopting them. Other works [3, 13, 1, 8] suffer from lack of formality or feature; they focus mostly on structural aspects of patterns, and pay little attention to behavioral aspects.

We describe a metamodeling approach that uses a pattern specification language called *Role-Based Metamodeling Language* (RBML). The RBML allows pattern authors to specify various perspectives of design patterns such as static structure, interactions, and state-based behavior. The RBML uses visual notations based on the *Unified Modeling Language* (UML) [**?**] and textual constraints expressed in *Object Constraint Language* (OCL) [16] to specify pattern properties.

The UML infrastructure is defined as a four-layer metamodel architecture: Level M3 defines a language for specifying metamodels, level M2 defines the UML metamodel, level M1 consists of UML models specified by the M2 metamodel, and level M0 consists of object configurations specified by the models at level M1. The pattern specifications in this paper specify families of models at the M1 level by constraining the metamodel defined at the M2 level. Adding constraints to the UML metamodel results in a *specialized* metamodel that specifies a subset of valid UML models. A specification of a pattern's solution space can be obtained by constraining the UML metamodel to create a specialized UML metamodel that specifies the solution space of the pattern.

We give an overview of the RBML in Section 2 and describe how it can be used to specify design patterns using the *Iterator* design pattern with an SPS and an SMPS in Section 3. IPSs are not used in the specification of the Iterator pattern since there is no interactions involved among the pattern participants that need to be captured in IPSs. We illustrate the use of the SPS and the SMPS to check pattern conformance of a model for a television remote application in Section 4. We describe a related work in Section 5, and conclude in Section 6.

2 Overview of the RBML

The RBML is a pattern specification language for characterizing families of UML models. In this paper, a Pattern Specification (PS) defines a family of UML models in terms of roles [10] , where a role is associated with a UML metaclass as its base. A role specifies properties that model elements that play (conform to) the role must possess, that is, the properties determine a subset of the instances of its base. For example, a role with the base Association metaclass defines properties that determine a subset of UML associations (instances of Association) satisfying the properties. A conforming model of a PS is a model (e.g., a class diagram, a sequence diagram) that consists of model elements that play the roles defined in the PS. A conforming model may also have application-specific model elements if they do not violate the constraints defined in the PS.

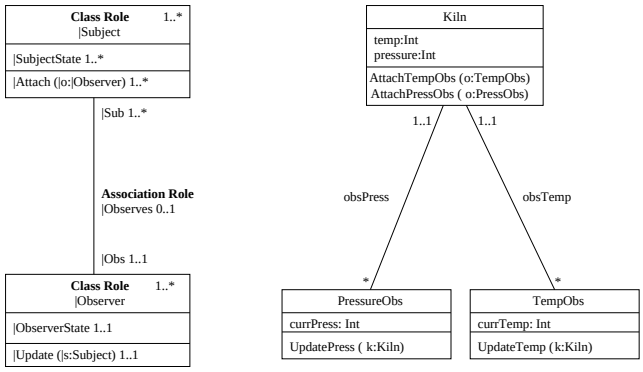
2.1 Static Pattern Specifications (SPSs)

An SPS defines constraints on the UML metamodel to restrict down to the solution space of a pattern regarding structural aspects. Conforming models of an SPS are UML class diagrams. An SPS consists of *classifier* and *relationship* roles whose bases are Classifier and Relationship metaclasses in the UML metamodel. A classifier role is connected to other classifier roles by relationship roles. An SPS role is associated with *metamodel-level constraints* and *constraint templates* expressed in OCL. Metamodel-level constraints are well-formedness rules that further restrict the UML metamodel (see examples in Section 3.1). Constraint templates specify model-level constraints (e.g., pre- and post-conditions) that are produced by substituting role names in templates to conforming model elements in the conforming model (see examples in Section 3.1).

Fig. 1 shows an SPS and a conforming class diagram [1]. The SPS in Fig. 1(a) has two classifier roles *Subject* and *Observer* and an association role *Observes* in between. We use the symbol "|" to indicate our roles. Text labels (e.g., *Class Role*) may be used with roles to denote their bases in the UML metamodel when the base is not obvious. The *Subject* role has *SubjectState* role whose a base is the StructuralFeature metaclass and *Attach* role whose a base is the BehavioralFeature metaclass. The *Attach* role has *o* parameter role that is played by parameters whose a type is a class that plays the *Observer* role. Properties of *Observer* role are interpreted similarly.

Dashed arrows show a mapping between roles in the SPS and model elements in the conforming model. For example, both the *AttachTempObs* and the *AttachPressObs* operations play the *Attach* role. Stereotype symbols may be used in conforming models to denote the roles that the model elements play (see examples in Section 4). The multiplicity constraint (1..*) following the role name indicates that one or more operations can play the role. The two association ends on *Kiln* class conform to *Sub* role whose base is AssociationEnd metaclass. The multiplicities (1..1) on the association ends are defined in metamodel-level constraints (not shown) of the *Sub* role.

[1] Metamodel-level constraints and constraint templates are not shown



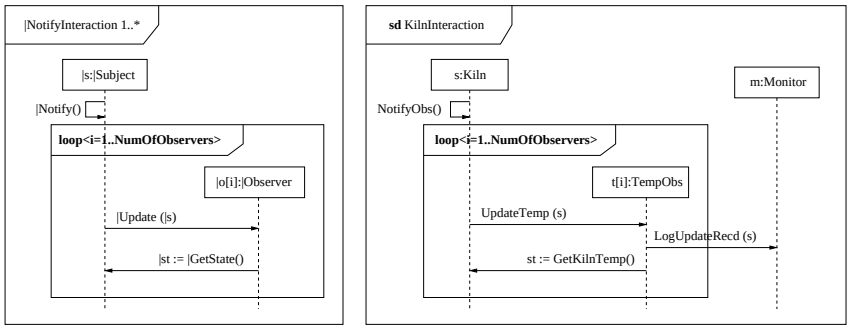
(a) An SPS for a Variant Observer Pattern (b) A Conforming Class Diagram

Figure 1. An SPS and a Conforming Class Diagram

2.2 Interaction Pattern Specifications (IPSs)

IPSs are used to constrain interactions between pattern participants. An IPS consists of an *interaction* role that defines a specialization of the UML metamodel class $Interaction$. An interaction role is a structure of *lifeline* and *message* roles whose bases are Lifeline and Message metaclasses in the UML metamodel. Each lifeline role is associated with a classifier role in SPSs: 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 SPSs: a conforming message specifies a call to an operation that plays the behavioral feature role.

Fig. 2 shows an IPS and a conforming sequence diagram. The IPS in Fig. 2(a) describes that invocation of a subject's $Notify$ operation (i.e., an operation that plays 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. The lifeline role $|s : |Subject$ is associated with the classifier role $Subject$, and the message role $Update$ is associated with the feature role $Update$ in the SPS in Fig. 1(a). The metamodel-level constraint on the *Update* role that messages playing the



(a) An IPS of Variant Observer Pattern

(b) A Conforming Sequence Diagram

Figure 2. An IPS and a Conforming Sequence Diagram

Update role must be asynchronous calls is defined as follows (a similiar constraint is defined on the *GetState*):

context |Update **inv**: self.messageSort = asynchcall

A sequence diagram conforms to an IPS if the conforming interactions respect the relative order specified in the IPS. A conforming sequence diagram of the $NotifyInteraction$ IPS is shown in Fig. 1(b). The sequence diagram has the interaction pattern specified in the IPS: the relative order of the conforming message, $NotifyObs$, $UpdateTemp$, and $GetKilnTemp$ is the same as the relative order specified in the IPS. The application specific lifeline *m* receiving *LogUpdateRecd* message is added during realization.

2.3 StateMachine Pattern Specifications (SMPSs)

SMPSs are used to specify state-based behavior of patterns. An SMPS consists of *state*, *transition*, and *trigger* roles whose bases are State, Transition, and Trigger metaclasses, respectively. An SMPS role may be associated with metamodel-level constraints (see examples in Section 3.2).

Fig. 7 shows an SMPS for turnstyle systems and a conforming model of subway turnstyle system [2]. The SMPS in Fig. 7(a) describes the following:

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

[2]Metamodel-level constraints are not shown

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

A conforming model in Fig. 7(b) shows that *Locked* plays the *Closed* role, *Unlocked* plays the *Open* role, and both *Coin* and *Card* triggers play the *Pay* trigger role indicated. *Alarm*, *PassBar*, and *Reset* are application-specific properties.

3 Specification of Iterator Design Pattern

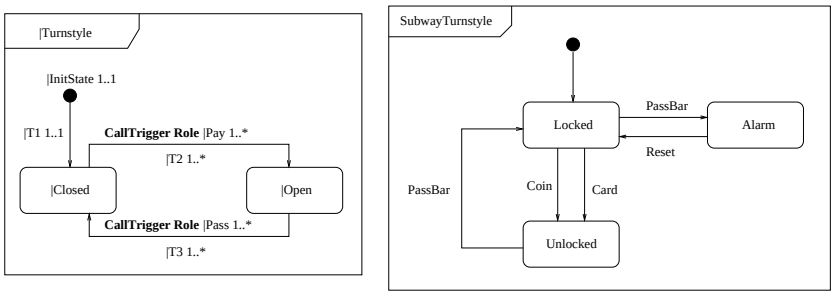
The *Iterator* design pattern [7] provides sequential access to the elements of an aggregate object independent of its underlying implementation details. In this section, we present an SPS and an SMPS of this pattern.

3.1 Iterator SPS

Fig. 5 shows an SPS of the *Iterator* pattern based on our interpretation of the descriptions in [7]. It is not intended to reflect a complete characterization for the pattern. The SPS specifies the following properties:

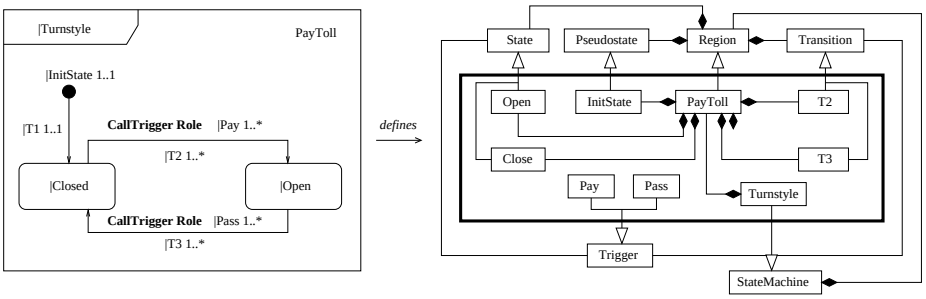
- ***Aggregate hierarchy*** consists of *Aggregate*, *AbstractAggregate*, and *ConcreteAggregate* classifier roles and *AggregateRealization* and *AggregateGeneralization* relationship roles. The relationship roles designate that there may be a generalization or a realization relationship between the conforming elements of *AbstractAggregate* and *ConcreteAggregate*.

  Fig. 6 shows conforming class diagrams of the *Aggregate* hierarchy. In Fig. 6(a) the realization relationship between *InterfaceA* and its realizing classes plays *AggregateRealization* role, and the generalization between *ClassB* and *ClassC* plays *AggregateGeneralization* role. Similarly, in Fig. 6(b) the generalization be-



(a) An SMPS (b) A Conforming State Machine Diagram

Figure 3. An SMPS and a Conforming Statechart Diagram



(a) Example of an SMPS (b) The Specialized UML Metamodel

Figure 4. An SMPS and a Conforming Statechart Diagram

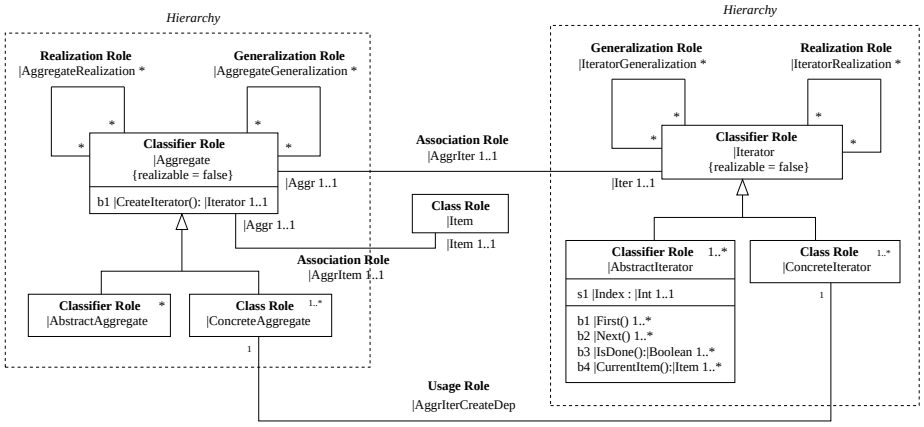


Figure 5. Iterator SPS

tween *ClassD* and its subtypes conforms to *Aggregate-Generalization* role.

Aggregate is an abstract role (i.e., {realizable = false}) where its properties (e.g., *CreateIterator*) are inherited by *AbstractAggregate* and *ConcreteAggregate*. The *CreateIterator* behavior creates an object of type *Iterator* and returns the object. The multiplicity following the *CreateIterator* role constrains that there should exactly one attribute that plays the role. The following are the metamodel-level constraints defined for the Aggregate hierarchy (similar constraints are defined for the $Iterator$ hierarchy role):

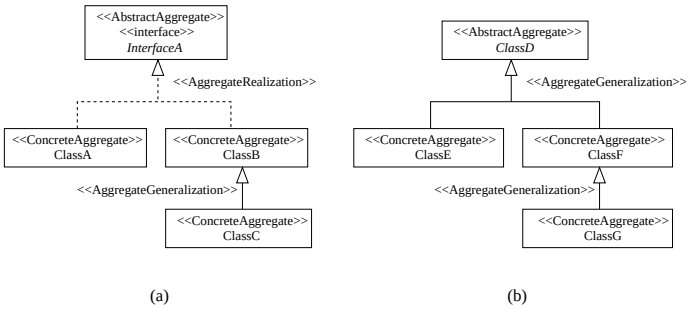


Figure 6. Conforming Class Diagrams of *Aggregate* Hierarchy

1. Conforming *AbstractAggregate* classifiers must either be interfaces or abstract classes:

   context |AbstractAggregate **inv**:
   (self.oclIsTypeOf(Interface) or
   (self.oclIsTypeOf(Class) and
   self.isAbstract = true)) and
   self.Property.AggregationKind = shared

2. Conforming *ConcreteAggregate* classifiers must be concrete classes:

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

- ***Iterator hierarchy*** consists of *Iterator*, *AbstractIterator*, and *ConcreteIterator* classifier roles and *IteratorRealization* and *IteratorGeneralization* relationship roles. *AbstractIterator* has *Index* structural feature role and behavior feature roles of *First*, *Next*, *IsDone*, and *CurrentItem*. A constraint template of the *Next* role is given below:

  context |AbstractIterator :: |Next()
  pre: |Index $\geq$ 1 and
  |Index $<$ self.|Aggregate.|Item
  $\rightarrow$ asSequence() $\rightarrow$ size()
  post: |Index = |Index@pre + 1

- ***Relationship roles***: *AggrItem* association role between *Aggregate* and *Item* represents that objects of a classifier playing *Aggregate* role are aggregations of objects of a class playing *Item* role. *AggrIterCreateDep* dependency role describes that *ConcreteAggregate* is responsible for creating objects of a class playing *ConcreteIterator*. The association role *AggrIter* is used for iterators to keep track of the current item in aggregates.

3.2 Iterator SMPS

Fig. 7 presents an SMPS for the *Iterator* role in the *Iterator* pattern. The SMPS depicts the following behavior:

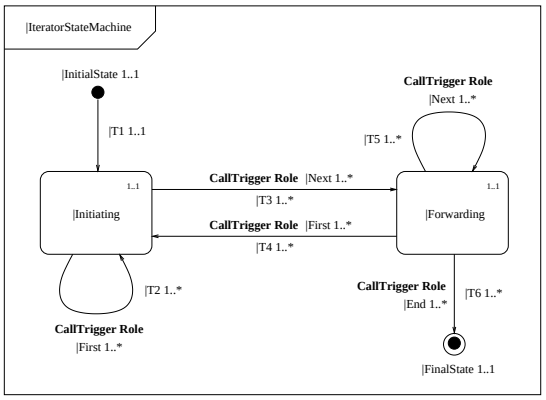


Figure 7. An SMPS of *Iterator* Role in Fig. 5

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

- ***Forwarding***: An object in this state has three possible transitions: 1) the object may move back to a state of *Initiating* when a call trigger of *First* is received; 2) the object may stay at the current state if a call trigger of *Next* is received; or 3) the object terminates its lifetime at a state of *FinalState* when a call trigger of *End* occurs.

Corresponding metamodel-level constraints are specified in Fig. 8. They address the following:

- Conforming state machines of *IteratorStateMachine* and transitions playing *T1* may be extended (isFinal = false). Constraints for the other transition roles are the same as for *T1*.

- States playing *Initiating* and *Forwarding* must be simple states (isSimple = true) and may be extended.

- Trigger playing *First*, *Next*, and *End* roles must be call triggers that are caused by the receipts of *First*, *Next*,

and *isDone* operation calls in Fig. 5. The relationship how SMPSs are linked to SPSs can only be shown at meta level.

- States playing *InitialState* must be *Initial* states.

StateMachine Role |IteratorStateMachine	**Transition Role** |T1
{self.isFinal = false}	{self.isFinal = false}
State Role |Initiating	**State Role** |Forwarding
{self.isSimple = true and self.isFinal = false}	{self.isSimple = true and self.isFinal = false}
CallTrigger Role |First	**CallTrigger Role** |Next
{self.operation.oclIsKindOf(|First)}	{self.operation.oclIsKindOf(|Next)}
CallTrigger Role |End	**PseudoState Role** |InitialState
{self.operation.oclIsKindOf(|IsDone)}	{self.kind = #initial}

Figure 8. Metamodel-level Constraints

SMPSs can be viewed as determining a constrained form of the UML metamodel. Fig. 9 describes the relationship between SMPSs and the UML metamodel. The figure shows that the *MyStateMachine* role whose conforming state machine is *MyGeneralStateMachine*, defines a subset of instances of UML *StateMachine* metamodel class whose instances are *GeneralStateMachine*, *MyGeneralStateMachine*, and *MySpecialStateMachine*, while keeping the specialization relationship between *MyGeneralStateMachine* and *MySpecialStateMachine*.

A state machine can be specialized by 1) adding regions, states, and transitions, 2) extending regions and states, or 3) redefining (replacing) transitions [**?**]. An extension of a region and a state occurs when new states and/or transitions are added (e.g., a simple state becomes a composite state). A redefinition of a transition occurs when the target state of the transition is changed.

Note that for the specialization cases, the *MySpecialStateMachine may* or *may not* conform to *MyStateMachine* role. For example, if *MySpecialStateMachine* has a transition that is redefined to have a different target than in *MyGeneralStateMachine*, then *MySpecialStateMachine* does not conform to *MyStateMachine* SMPS. Therefore, given a statemachine that conforms to an SMPS, it is possible to have non-conforming specializations of the state machine to the SMPS.

In order for *MySpecialStateMachine* to be a conforming state machine of *MyStateMachine* SMPS, the *isFinal* constraint must be set to true for all roles in the SMPS whose conforming model elements are generalizable (e.g., states, transitions, events). This constraint disallows the cases of specialization described above. In spite of this restriction,

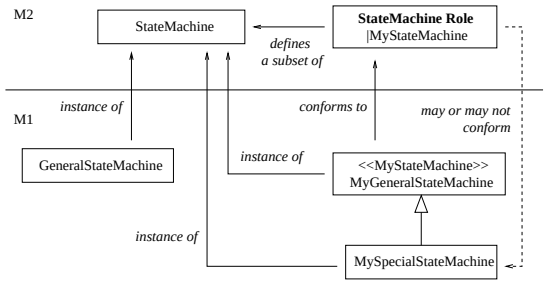


Figure 9. SMPS and UML metamodel

state machines can still be extended in a limited, but conforming way (e.g., by adding new transitions) if *isFinal* is set to false for the StateMachine role itself.

4 Conforming Models of Iterator Pattern Specification

In this section, we show conforming models of the SPS and the SMPS described in Section 3 using a television remote control application that allows viewers to easily surf channels through features like *channel next* and *previous*. Establishing that an application model conforms to Pattern Specifications requires identifying model constructs that play the roles in the Pattern Specifications.

Fig. 10 shows a class diagram that conforms to the Iterator SPS in Fig. 5. Stereotypes with model elements represent the roles that the model elements play. For example, *CreateChanIter* operation in *ChannelList* and *ConcChannelList* plays the *CreateIterator* role in Fig. 5. *lastChannel*, *previousChannel*, and *hasPrevious* are application-specific operations. *lastChannel* operation returns the last channel in the list, *previousChannel* returns the previous channel of the current channel, and *hasPrevious* checks if the current channel is the first channel in the list.

Built by substituting role names in the constraint templates in Section 3.1 to conforming model elements, pre- and post-conditions for *nextChannel* operation is given below:

context ChannelIterator :: nextChannel()
  pre: idx $\geq$ 1 and
    idx $<$ self.ChannelList.Channel $\rightarrow$
    asSequence() $\rightarrow$ size()
  post: idx = idx@pre + 1

Fig. 11 shows a statechart diagram that conforms to the Iterator SMPS in Fig. 7. Model elements that play the roles in Fig. 7 are stereotyped.

Fig. 12 shows a case in which a specialization of the television remote state machine *does not* conform to the Iterator

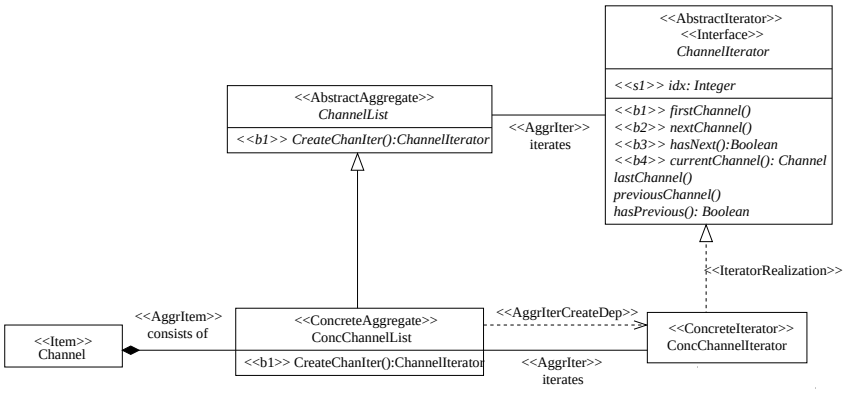


Figure 10. A Realization of *Iterator* SRM in Fig. 5

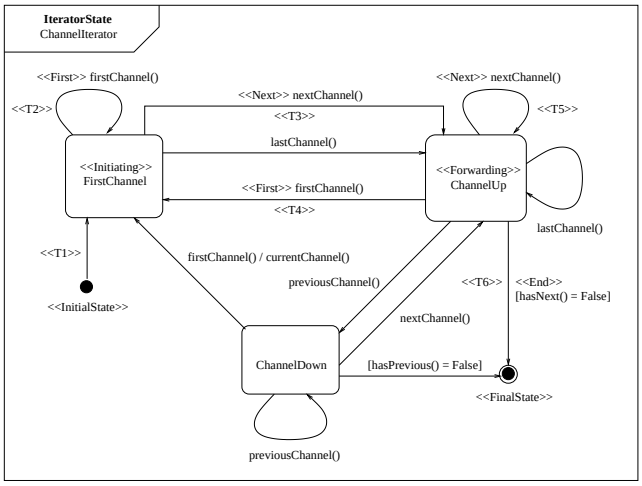


Figure 11. A Television Remote Realization of the *Iterator* SMPS in Fig. 7

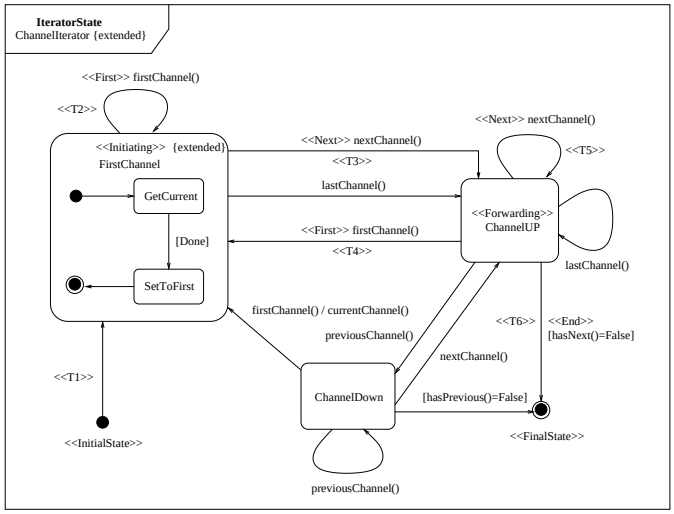


Figure 12. A Specialization of the Television Remote StateMachine in Fig. 7

SMPS. The simple state *FirstChannel* that plays the *Initiating* role in Fig. 7 is extended to a composite state to save the current index before resetting the iterator. This violates the constraint (*isSimple = true*) defined in *Initiating* role in Fig. 8, and thus the specialization does not conform to the SMPS. In order to disallow such cases, the constraint described in Section 3.2 must hold.

5 Related Work

Mapelsden *et al.* [13] propose the DPML, a visual modeling language, that provides a set of constructs (e.g., interface, method) to specify design pattern solutions. A pattern specification is instantiated to produce pattern instances that are part of UML object model. The participants in an instance are linked to objects in the object model while maintaining a binding to the UML model elements. Their approach requires both UML object model and UML class diagram to check pattern conformance of class diagrams. This can be simplified by checking conformance directly from class diagrams to the pattern specification. Linking elements of pattern instance to UML model elements is very coarse-grained (just done by type matching). No mechanisms are described as to how to specify constraints (e.g., pre- and post-conditions) on operations in pattern participants, nor are behavioral aspects of patterns addressed.

Fontoura *et at.* [4] describe the UML-F, a UML profile for adaptation of a framework for core and architecture reuse. The UML-F comprises four layers of tags; tags

for patterns, tags for construction principles, basic modeling tags, and presentation tags. Their work is rooted from the perspective of extensibility and implementation reuse. Therefore, patterns are described as framework with default implementation and variation points to be extended. In our approach, we focus on model reuse by defining pattern propoerties at metamodel level.

Several UML-based metamodeling approaches have been proposed [8, 1, 12]. Guennec *et al.* [8] use meta-collaborations to specify design patterns. Meta-collaborations whose collaboration roles are played by UML model elements utilize a family of recurring properties. Their work, however, does not describe how pattern properties (other than hierarchical structures of classifiers) are specified, nor is there a clear notion of what it means for a model to satisfy a role model.

Albin-Amiot and Gueheneuc [1] propose a metamodel to describe structural and behavioral aspects of design patterns for automatic code generation and design pattern detection. To instantiate a pattern, the metamodel is specialized to add needed structural and behavioral constituents. The specialized metamodel is then instantiated to produce an abstract model which is instantiated into a concrete model or parameterized to match a concrete model. We found their metamodel complicated and even gets worse when all meta classes are presented. They do not describe how to represent behavioral aspects of design patterns at model level.

Lauder and Kent [12] use graphical constraint diagrams to present patterns precisely and visually. 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 establishing refinement relationships across the model levels. Providing non-trivial automated support for such conformance checking requires support for proof generation/checking. Furthermore, the authors use a graphical form of constraints that is appealing but we found the notation difficult to understand.

6 Conclusions and Future Work

This paper has described the RBML, a pattern specification language, that can be used by tool developers who need precise descriptions of design patterns. The language has been used to specify many design patterns in the book by Gamma *et al.*. To date we have developed full pattern specifications [6, 5] for: Abstract Factory, Bridge, Decorator, Singleton, Observer, Visitor, and Composite, in addition to the Iterator pattern presented in this paper. We also used the language to specify a domain pattern for checkin-checkout systems [9]. This pattern specifies a family of checkin-checkout systems (e.g., car rental and library systems).

The RBML is developed with the intend to be compliant with existing UML drawing tools (e.g., Rational Rose, Microsoft Visio). The notational variants (e.g., realization multiplicities) in the RBML, however, may require an extension of the tools.

The popularity of the UML and the heightened interest in model-driven approaches to software development (e.g., see the OMG's Model-Driven Architecture; http://www.omg.org/mda), has raised interest in tools that support model transformations. The pattern specification technique described in this paper can also be used to support the OMG's initiatives that focus on providing tool support for pattern-based model refactoring. Our current work is concerned with developing practical and rigorous pattern-based model-refactoring techniques.

References

[1] H. Albin-Amiot and Y. G. Gueheneuc. Meta-Modelling Design Patterns: Application to Pattern Detection and Code Synthesis. In *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

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

[3] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Proceedings of the 11th European conference on Object Oriented programming, Springer LNCS 1241*, pages 472–495, 1997.

[4] M. Fontoura, W. Pree, and B. Rumpe. *The UML Profile for Framework Architectures*. Addison-Wesley, 2002.

[5] R. France, D. Kim, E. Song, and S. Ghosh. Patterns as Precise Characterizations of Designs. Technical Report 02-101, Computer Science Department, Colorado State University, Fort Collins, CO, January 2001.

[6] R. France, D. Kim, E. Song, and S. Ghosh. Using Roles to Characterize Model Families. In Haim Kilov, editor, *Practical foundations of business and system specifications*, pages 179–195. Kluwer Academic Publishers, 2003.

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

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

[9] D. Kim, R. France, S. Ghosh, , and E. Song. Using Role-Based Modeling Language (RBML) as Precise Characterizations of Model Families. In *Proceedings of The 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002)*, Greenbelt, MD, December 2002.

[10] 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.

[11] 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.

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

[13] D. Mapelsden, J. Hosking, and J. Grundy. Design Pattern Modelling and Instantiation using DPML. *ACM International Conference Proceeding Series, Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, 10:3–11, 2002.

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

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

[16] The Object Management Group (OMG). Unified Modeling Language. Version 1.4, OMG, http://www.omg.org, September 2001.