Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

A metamodeling approach for pattern specification and management*

Liang DOU†, Qiang LIU, Zong-yuan YANG

(*Department of Computer Science and Technology, East China Normal University, Shanghai 200241, China*)

†E-mail: ldou@cs.ecnu.edu.cn

Received Feb. 5, 2013; Revision accepted July 9, 2013; Crosschecked Sept. 16, 2013

Abstract: The formal specification of design patterns is central to pattern research and is the foundation of solving various pattern-related problems. In this paper, we propose a metamodeling approach for pattern specification, in which a pattern is modeled as a meta-level class and its participants are meta-level references. Instead of defining a new metamodel, we reuse the Unified Modeling Language (UML) metamodel and incorporate the concepts of Variable and Set into our approach, which are unavailable in the UML but essential for pattern specification. Our approach provides straightforward solutions for pattern-related problems, such as pattern instantiation, evolution, and implementation. By integrating the solutions into a single framework, we can construct a pattern management system, in which patterns can be instantiated, evolved, and implemented in a correct and manageable way.

Key words: Design patterns, Metamodeling, Pattern management system, Kermeta, Java modeling languages
doi:10.1631/jzus.C1300040 **Document code:** A **CLC number:** TP311

1 Introduction

The Gang of Four (GoF) design patterns (Gamma *et al*., 2004) document elegant design solutions to facilitate the reuse of designs. The design patterns are widely adopted as a mechanism in disseminating the best practices within a given context in software design. However, patterns are originally specified in the Unified Modeling Language (UML) diagrams and English annotations. They are intended to be read by humans, and not to be processed by machines. To make pattern descriptions machine-processable and to provide tool support for pattern-related problems, the community proposes the dedicated Pattern Specification Language (PSL). However, Eden and Hirshfeld (2001) observed that PSL is lacking some essential concepts that are frequently used but are not explicitly available in UML, such as Variable and Set. Moreover, the main practice field of design patterns is the object-oriented (OO) modeling paradigm. The inherent incompatibility between the formal methods and the modeling paradigm makes it extremely difficult to build a practical tool that supports pattern practitioners. Therefore, interest has been growing in metamodeling design patterns, which holds potential for higher-level abstraction.

Metamodeling is a four-layer architectural modeling proposed by the Object Management Group (OMG). As shown in Fig. 1 (Atkinson and Kuhne, 2003), meta-metamodel level (M3) defines a modeling framework in which the abstract syntax of modeling languages in the metamodel level (M2) can be defined. The model level (M1) is populated with models constructed from the modeling languages in M2. The M0 includes entities in the world, which are modeled by M1 models. For example, the Eclipse Modeling Framework (http://www.eclipse.org/modeling/emf/) is a widely used modeling framework whose M3 level is called ECore, which includes a set

* Project (Nos. 61070226 and 61003181) supported by the National Natural Science Foundation of China

of primitive modeling concepts (EClass, EReference, among others) that can be used to describe metamodels, such as the metamodel of UML.

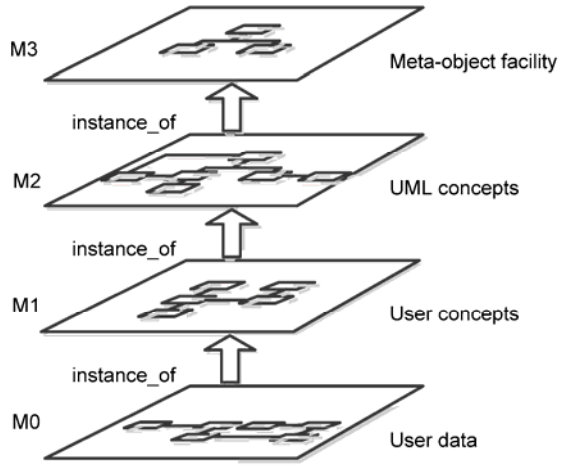

Fig. 1 Object Management Group (OMG)'s metamodeling infrastructure

In this paper, we propose a metamodeling approach for pattern specification. Each pattern is modeled as an EClass. Each EClass owns a set of EReferences for modeling the pattern participants. By using EReference, the concepts of Variable and Set are incorporated into our approach, which allows for specifying patterns in a more abstract manner. An EReference can define a variable with different multiplicities, which are capable of modeling a set of elements. Furthermore, all EReferences refer to elements in the UML metamodel. In this way, we reuse the UML metamodel rather than define a completely new metamodel. Given that UML is widely adopted in the modeling community, we believe it is capable of precisely specifying design patterns as long as some pattern-level abstract concepts, such as Variable and Set, are introduced into it in an appropriate way. In specifying the behavior of the pattern, we introduce another concept of dynamic relation, which is unavailable in UML. By explicitly defining the dynamic relations, the precise specification and preservation of the behavior of the pattern are feasible.

Specifically, we address three pattern-related problems: pattern instantiation, pattern evolution, and pattern implementation.

Pattern instantiation: Pattern instantiation considers how to produce an instance of pattern that meets the given pattern specification. In the context of metamodeling, a metamodel of a pattern is capable of generating infinite conforming pattern instances. However, some instances are illegal. A legal pattern instance is a model whose instantiated participants conform to the invariant of pattern. Given the specification of a pattern that includes a set of EReferences and an invariant, assigning values properly to all EReferences is important in conforming to the invariant. Our approach provides an easy way to define an EOperation in the EClass of a pattern, which creates UML elements, assigns EReferences, and ensures that the invariant is satisfied after execution.

Pattern evolution: The existing instances of patterns in a system make it easy to understand and maintain the design. However, if the system design evolves in an arbitrary manner, then these instances could be destroyed, and their desirable properties would disappear during evolution. Therefore, a set of rules should guide the evolution of patterns, such that the pattern instances always conform to the invariant. Our solution defines a set of evolving operations (using EOperations from ECore) in the EClass of a pattern. Each operation implements an evolving rule of the pattern and manipulates the legal pattern instances properly to make sure they are still legal after execution.

Pattern implementation: Pattern implementation refers to how design patterns should be implemented in a particular programming language. Traditional solutions (Mapelsden *et al*., 2002; Eden *et al*., 2006) focus on generating code skeletons of pattern structure for specific implementation platforms. However, the behaviors of patterns are ignored in the generated code skeletons. Our approach improves the situation by programmatically generating Java Modeling Language (JML) (Leavens *et al*., 1998) specifications for pattern-level operations. For example, UML operation elements are referenced by the EReferences of a pattern. The generated JML specifications can provide guidelines for system implementation and verify pattern behavior by leveraging the JML runtime checking mechanism.

Our approach provides a way to "program at the design level". For instance, a designer has made a design-level decision to use the Observer pattern. Instead of implementing the pattern from scratch, the designer can use our pattern instantiation library to produce the system design. What the designer needs

to do is to identify some important roles in the pattern such as Observer and Subject. Then our approach will automatically take care of the other roles and add concrete relations among them. Furthermore, our approach is useful for pattern evolution in the whole process of system development. The designer can use our tool to evolve the pattern by providing only a few key parameters and our tool will produce the rest of the code. In this sense, our approach can significantly raise the level of abstraction and lower the barrier of using design patterns.

2 Specifying design patterns

Two essential concepts of pattern specification are Variable and Set. Design patterns should not be considered specifying properties of particular elements (e.g., class and operation), but specifying properties of variables. For example, in Fig. 2, the Observer should be interpreted as a variable of the UML class rather than a constant of the class. Although only one concrete observer is shown in Fig. 2, the pattern actually specifies a set of infinite concrete observers with common properties and behaviors.

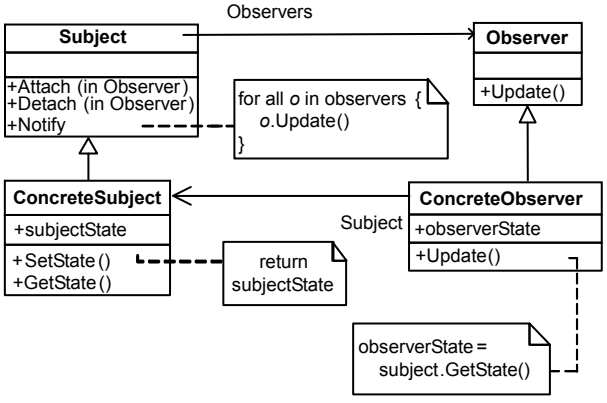

Fig. 2 Observer pattern in Unified Modeling Language (UML)

However, the concepts of Variable and Set are not available in the UML metamodel. To overcome this shortcoming, Albin-Amiot and Guehénéuc (2001) and Mapelsden *et al*. (2002) built new metamodels that explicitly contain the concepts of Variable and Set along with some frequently used elements in the UML metamodel, from which a design pattern description can be obtained. The main disadvantage of these approaches is that they degrade pattern specification to the M1 level, which means each pattern specification is actually a model, and not a metamodel. Hence, the instances of patterns are no longer UML models, but M0 level entity models. As a result, designers cannot work directly with UML models, and they need to learn a new language to understand pattern specification.

To avoid such problems, we propose a metamodeling approach for pattern specification. By using metamodeling design patterns, we construct a metamodel for each pattern such that the pattern instances are M1 level models. The main idea is to reuse the UML metamodel as a foundation for all patterns. To create a metamodel for a new pattern, we simply define an EClass modeling the pattern and define a set of EReferences referring to all its participants. In this way, pattern participants are no longer restricted to UML classes and can use any type of UML element, e.g., operation, property, and parameter. Pattern specification can document every element involved in its structure and behavior. More importantly, the concepts of Variable and Set are incorporated into the concept of EReference because the name of an EReference is essentially a variable name, and its multiplicity is capable of modeling the concept of Set. For example, a part of the metamodel of the Observer pattern in ECore is as described in Fig. 3.

The EClass DesignPattern is an abstract class for all patterns that defines a set of attributes and operations shared by all concrete design patterns. ObserverPattern, as a concrete pattern, inherits from the DesignPattern and has a set of EReferences corresponding to elements in the UML metamodel. For example, EReference Subject defines a class variable with multiplicity [1..1], which can be instantiated by one class constant. EReference ConSubs defines a class variable with multiplicity [0..*], which can refer to a set of class constants. EReference Attach is a participant of type Operation with multiplicity [1..1], and EReference ConSubStates is of type Property with multiplicity [0...*]. They refer to an operation owned by Subject and a set of states owned by each concrete subject. According to their multiplicities, participants can be divided into singular and set participants, with values [1..1] and [0..*], respectively.

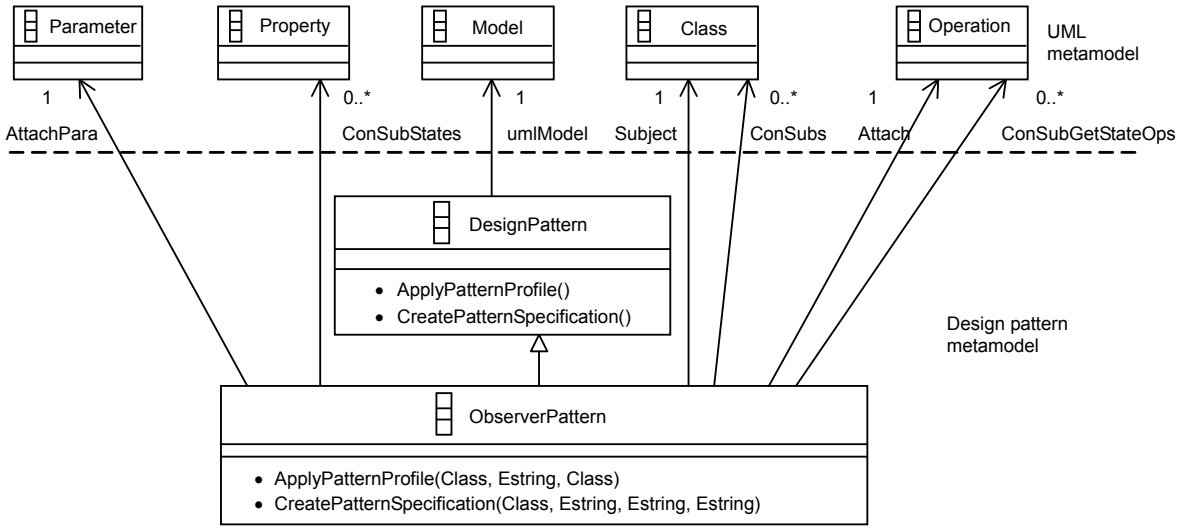

Fig. 3 A part of the metamodel of the Observer pattern in ECore

Kermeta (http://www.kermeta.org/), which is a modeling and aspect-oriented programming language, is conceived as a core for (meta)modeling. Kermeta is compatible with ECore, and ECore metamodels can be automatically translated into Kermeta codes. Using the Kermeta language, the metamodel of a pattern can be easily described. The structural part of the metamodel of the Observer pattern written in Kermeta is presented in Table 1.

In Table 1, EReferences model only the basic information of participants, including their names, types, and multiplicities. Obviously, their relationships are not presented. For example, class *Subject* should own operation *Attach*, and concrete subjects in *ConSubs* should all inherit from class *Subject*. Therefore, based on existing relations in the UML metamodel, we formally define the relationships among pattern participants. Table 2 presents a set of primitive relationships, while Table 3 shows part of their implementation in Kermeta, which is a powerful model oriented language and can describe structure and behavior of a design pattern. For example, *HasOperation* is a Boolean type operation in Kermeta, which means class *c* has operation *op*. The implementation of this relation is based on the UML relationship *ownedOperation* and it simply checks whether *op* is in the *ownedOperation* of class *c*. *HasAttribute* is based on the relationship *ownedAttribute*.

Table 1 Metamodel of the Observer pattern in Kermeta

```
class ObserverPattern inherits DesignPattern {
  reference Observer: uml::Class[1..1]
  reference ConObss: uml::Class[0..*]
  reference Subject: uml::Class[1..1]
  reference ConSubs: uml::Class[0..*]
  reference ConObsSts: uml::Property[0..*]
  reference ConSubSts: uml::Property[0..*]
  reference Attach: uml::Operation[1..1]
  reference AttachPara: uml::Parameter[1..1]
  reference Detach: uml::Operation[1..1]
  reference DetachPara: uml::Parameter[1..1]
  reference Notify: uml::Operation[1..1]
  reference Update: uml::Operation[1..1]
  reference ConSubGetStOps: uml::Operation[0..*]
  reference ConSubSetStOps: uml::Operation[0..*]
  reference ConObsUpdOps: uml::Operation[0..*]
  reference SubAsObs: uml::Association[1..1]
  reference ConobsAsConsub: uml::Association[0..*]
}
```

Table 2 Primitive relationships

Name	Parameter(s)	Description
IsAbstract	c: Class	c is an abstract class
Inherit	c1: Class; c2: Class	c1 inherits from c2
HasOperation	c: Class; op: Operation	c has an operation op
HasAttribute	c: Class; p: Property	c has a property p
HasParameter	op: Operation; para: Parameter	op has a parameter p
HasAssociation	c1: Class; a: Association; c2: Class	c1 is associated with c2 via a

Table 3 Part of the implementation of primitive relationships in Kermeta

```
operation HasOperation(c: Class, op: Operation): Boolean is
do
   result := c.ownedOperation.contains(op)
end
operation HasAttribute(c: Class, p: Property): Boolean is
do
   result := c.ownedAttribute.contains(p)
end
operation HasAssociation(c1: Class, a: Association, c2:
 Class): Boolean is
do
   result := a.memberEnd.exists {p1 | a.memberEnd.exists
     {p2 | (p1.type == c1 and p2.type == c2) or
           (p1.type == c2 and p2.type == c1)}}
end
```

Primitive relationships describe the relationships between singular participants. Complex relationships between set participants can be defined based on primitive relationships. Table 4 presents a set of complex relationships. *Hierarchy*(*cset*, *super*) specifies the relationship between a class set *cset* and a class element *super*. It is defined on top of the primitive relationship *Inherit* and uses the universal quantifier *forAll* to check whether all the elements in *cset* inherit from *super*. For example, all classes in *ConSubs* should inherit from *Subject* in the Observer pattern.

Things can get complicated when specifying relationships between sets. There are three relationships between two sets: one-to-one, one-to-many, and many-to-many. For instance, a one-to-one relationship exists between *ConSubs* and *ConSubStates*, which means each class in *ConSubs* should own one and only one property in *ConSubStates*. Each property in *ConSubStates* should be owned by one and only one class in *ConSubs*. To specify this kind of abstract relationship, we use two general properties of relationships, namely Existence and Uniqueness. For sets A and B, and a binary primitive relationship R, Existence means for all a in A, there is one element b in B such that $R(a, b)$ holds. Uniqueness means if $R(a1, b)$ and $R(a2, b)$ are satisfied, then $a1=a2$ holds. In Table 4, we present the one-to-one relationship between a class set and an operation set, which is defined by combining four Existence and Uniqueness constraints. Many-to-one and one-to-many relationships can also be defined by different combinations of constraints.

Table 4 Binary complex relationship

```
operation Hierarchy(cset: Class[0..*], c: Class): Boolean is
do
  result := cset.forAll {cls | Inherit(cls, c)}
end
// existence and uniqueness of operation sets
// HasOpExOp and HasOpOneOp
operation HasOpExOp(cset: Class[0..*],
  opset: Operation[0..*]): Boolean is
do
  result := cset.forAll {c | opset.exists {o |
    HasOperation(c, o)}}
end
operation HasOpOneOp(cset: Class[0..*],
  opset: Operation[0..*]): Boolean is
do
  result := cset.forAll {c | opset.forAll {o1 | opset.forAll
   {o2 | not (HasOperation(c, o1) and HasOperation(c, o2))
   or o1 == o2}}}
end
// existence and uniqueness of class sets HasOpExCls
// and HasOpOneCls
...
// one-to-one relationship between the class set and
// the operation set
operation HasOpOtO(cset: Class[0..*], opset:
  Operation[0..*]): Boolean is
do
  result:=HasOpExOp(cset, opset) and
  HasOpOneOp(cset, opset) and HasOpExCls(cset, opset)
  and HasOpOneCls(cset, opset)
end
```

Similarly, ternary complex relationships can be defined based on ternary primitive relationships. As examples, Table 5 defines complex relationships based on ternary primitive relationship *HasAssociation*. In this case, six constraints are needed to impose a one-one-one ternary relationship among three sets. Selectively combining the restrictions could result in weaker relationships, which are also useful in pattern specification. For instance, in the Observer pattern, a concrete observer may refer to only one concrete subject through an association; however, many concrete observers through several associations can refer to a concrete subject. We could specify this relationship by a many-many-one complex relationship *HasAssoMtoMtoO*.

Based on the above primitive and complex relationships, we finally show the complete structural specification of the Observer pattern in Table 6. With Variable and Set serving as abstraction mechanisms,

Table 5 Ternary complex relationships

```
// existence and uniqueness of ass and cs2
operation HasAssoExAssCls2(cs1: Class[0..*],
  ass: Association[0..*], cs2: Class[0..*]): Boolean is
do
  result := cs1.forAll {c1 | ass.exists {a | cs2.exists {c2 |
    HasAssociation(c1, a, c2)}}}
end
operation HasAssoOneAssCls2(cs1: Class[0..*],
  ass: Association[0..*], cs2: Class[0..*]): Boolean is
do
  result := cs1.forAll {c | ass.forAll {a1 | ass.forAll {a2 |
    cs2.forAll {c1 | cs2.forAll {c2 | Implies((HasAssocia-
    tion(c, a1, c1) and HasAssociation(c, a2, c2)),
    (a1 == a2 and c1 == c2))}}}}}
end
// four other constraints
...
// many-many-one relation
operation HasAssoMtoMtoO(cs1: Class[0..*],
  ass: Association[0..*], cs2: Class[0..*]): Boolean is
do
  result := HasAssoExAssCls2(cs1, ass, cs2) and
    HasAssoExCls1Cls2(cs1, ass, cs2)
end
// one-one-one relation
operation HasAssoOtoOtoO(cs1: Class[0..*],
  ass: Association[0..*], cs2: Class[0..*]): Boolean is
do
  result := HasAssoExAssCls2(cs1, ass, cs2) and
  HasAssoOneAssCls2(cs1, ass, cs2) and
  HasAssoExCls1Cls2(cs1, ass, cs2) and
  HasAssoOneCls1Cls2(cs1, ass, cs2) and
  HasAssoExCls1Ass(cs1, ass, cs2) and
  HasAssoOneCls1Ass(cs1, ass, cs2)
end
```

the specification details the essential properties of each participant, and leaves other properties customizable. For example, the specification requires that each concrete observer in set *ConObss* should own a property in the set *ConObsSts* as its state. However, the invariant does not specify any other thing about the observer's state, such as name, type, and multiplicity. As a result, the value of all the unspecified properties can be customized without violating a pattern invariant. In this sense, our specification of the Observer pattern has a larger set of legal pattern instances than the original pattern specification proposed by Gamma *et al*. (2004).

Table 6 Specification of the Observer pattern

```
class ObserverPattern inherits DesignPattern {
...
inv struct_spec is do
  IsAbstract(Observer) and
  IsAbstract(Subject) and
  Hierarchy(ConObss, Observer) and
  Hierarchy(ConSubs, Subject) and
  HasAttOtO(ConObss, ConObsSts) and
  HasAttOtO(ConSubs, ConSubSts) and
  HasOperation(Subject, Attach) and
  HasParameter(Attach, AttachPara) and
  HasOperation(Subject, Detach) and
  HasParameter(Detach, DetachPara) and
  HasOperation(Subject, Notify) and
  HasOperation(Observer, Update) and
  IsAbstractOp(Update) and
  HasOpOtO(ConObss, ConObsUpdOps) and
  HasOpOtO(ConSubs, ConSubGetStOps) and
  HasOpOtO(ConSubs, ConSubSetStOps) and
  HasAssociation(Observer, SubAsObs, Subject) and
  HasAssoMtoMtoO(ConObss, ConobsAsConsub, ConSubs)
end
...
}
```

3 Managing design patterns

Based on the specification in the previous section, pattern-related problems can be solved by programmatically manipulating pattern participants at the meta-level. Specifically, the solutions are encapsulated in a set of EOperations. In this section, we focus on three pattern-related problems: pattern instantiation, evolution, and implementation.

3.1 Pattern instantiation

In general, a pattern modeled as an EClass may contain a set of EReferences and EOperations. EReferences refer to participants and OCL-like constraints that specify relationships between them. For a legal pattern instance, its instantiated participants should conform to the pattern invariant. To instantiate a pattern, the pattern instantiation operation *CreateInitStruct*() is defined to assign values (UML constants) to all the pattern's EReferences. The main concern during the implementation of *CreateInitStruct*() is to make sure that the pattern invariant is satisfied after instantiation. With the design-by-contract characteristic of Kermeta (operations support

pre- and post-conditions, and classes use invariants), the correctness of its implementation can be verified by checking the invariant (*checkInvariants*) during runtime after the execution of *CreateInitStruct*().

When initializing a pattern instance, we treat singular participants (with multiplicity [1..1]) and set participants (with multiplicity [0..*]) differently. Singular participants are assigned to newly created UML elements with corresponding types and default names. For example, the singular participant Subject in the Observer pattern can be assigned to the UML class constant with a default name *MySubject*. We initialize all the set participants as empty sets because at the beginning, we do not know what elements should populate these sets. When all the singular participants are properly assigned, and all the set participants are empty, we call the instance the initial structure (IS) of the pattern.

Table 7 shows our implementation for the instantiation of the Observer pattern. We develop a library in Kermeta to facilitate the creation of UML elements, such as class, operation, and association, among others. The codes create UML elements with corresponding types and default names, and properly create relationships between them to conform to the invariant. For example, when assigning EReference *Attach*, *createOperation* is used to generate a UML operation, and its first parameter indicates that the operation is owned by the class referenced by EReference *Subject*. This ownership is exactly required in the invariant.

By using this lightweight design, a pattern instance can be saved as a standard UML model, which can be directly consumed by other UML tools. For example, by executing the operation main() from Table 8, the generated IS of the Observer pattern is saved as the UML model and can be opened with Topcased (http://www.topcased.org) (Fig. 4).

Table 7 Instantiation of the Observer pattern

```
method CreateInitStruct(): Void is
do
  // call super() to initialize UML primitive types
  super()
  Observer := createClass(umlModel, "Observer", true)
  Subject := createClass(umlModel, "Subject", true)
  Attach := createOperation(Subject, "Attach", false)
  AttachPara := createParameter(Attach, "obs", Observer)
  Detach := createOperation(Subject, "Detach", false)
  DetachPara := createParameter(Detach, "obs", Observer)
  Notify := createOperation(Subject, "Notify", false)
  Update := createOperation(Observer, "Update", true)
  SubAsObs := createAssociation(umlModel, Subject, "",
    1, 1, false, AggregationKind.none, Observer, "observers",
    0, -1, true, AggregationKind.none)
end
```

Table 8 Example codes for instantiation of the Observer pattern

```
operation main(): Void is
Do
  // create an instance of the Observer pattern
  var AnOP: ObserverPattern init ObserverPattern.new
  // create Initial Structure
  AnOP.CreateInitStruct()
  // check the pattern invariant
  AnOP.checkInvariants
  // save as a UML model
  AnOP.save("ObserverPatternd1")
end
```

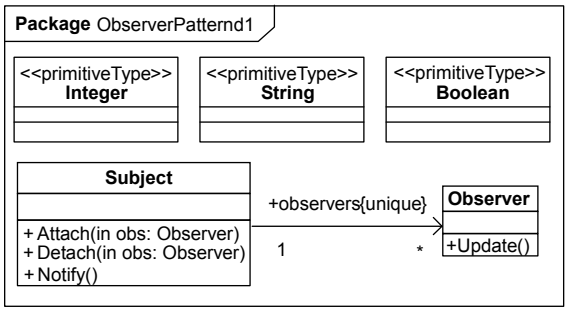

Fig. 4 A generated initial structure (IS) of the Observer pattern

3.2 Pattern evolution

During pattern instantiation, a legal pattern instance is produced. However, if the instance evolves arbitrarily, its invariant may be violated. Pattern participants are interconnected with each other by a set of complex relationships. Moreover, the evolution of an instance has to follow a set of rules. For example, if a designer wants to add a new concrete subject to the Observer pattern, simply creating the UML class would violate the pattern invariant. The violation occurs because the specification of the Observer pattern requires that each concrete subject should own a property in the set *ConSubStates* and two operations in sets *ConSubGetStateOps* and *ConSubSetStateOps*.

For each pattern, we define a set of EOperations called evolving operations to implement the evolving rules of the pattern. Singular participants are already

properly instantiated in pattern instantiation. Hence, set participants need to be handled by evolving operations. In the Observer pattern, an example of an evolving operation is *AddConcreteSubject*, which adds a new concrete subject to the pattern. Table 9 gives our implementation of *AddConcreteSubject* in which all the parameters are properly handled in five steps. Similarly, *AddConcreteObserver*, which adds a new concrete observer, is defined in Table 10. The operation performs six steps of changes to ensure that the invariant is conformed to after adding a new concrete observer. Evolving operations can also be defined to take names (of type String) as parameters instead of references of the UML element. Table 11 shows the signature of a new version for *AddConcreteObserver*. This operation is straightforward to use, since UML elements are created inside the operation by given names.

In this way, the designer can write programs that use our evolving operations to evolve the patterns. Fig. 5 shows the produced instance of the pattern by executing the example codes from Table 12. With a few lines of Kermeta codes, domain-specific instances of patterns can be produced by setting corresponding parameters.

Table 9 Evolution of the Observer pattern: add a concrete subject

```
operation AddConcreteSubject(ConSub: Class, ConSubSt:
  Property, ConSubStTy: Type, ConSubGetStOp:
  Operation, ConSubSetStOp: Operation): Void is
do
   // 1. add the new concrete subject to the UML model
   // and ConSubs
   umlModel.packagedElement.add(ConSub)
   ConSubs.add(ConSub)
   // 2. add property for ConSub and add ConSubSt to
   // set ConSubSts
   ConSubSt.type := ConSubStTy
   ConSub.ownedAttribute.add(ConSubSt)
   ConSubSts.add(ConSubSt)
   // 3. add Generalization from ConSub to Subject
   createGeneralization(ConSub, Subject)
   // 4. add the operation ConSubGetStOp for ConSub
   setRetType(ConSubGetStOp, ConSubStTy)
   ConSub.ownedOperation.add(ConSubGetStOp)
   ConSubGetStOps.add(ConSubGetStOp)
   // 5. add the operation ConSubSetStOp for ConSub
   createParameter(ConSubSetStOp, "value", ConSubStTy)
   ConSub.ownedOperation.add(ConSubSetStOp)
   ConSubSetStOps.add(ConSubSetStOp)
end
```

Table 10 Observer pattern evolution: add a concrete observer

```
operation AddConcreteObserver(ConObs: Class, ConObsSt:
 Property, ConObsStType: Type, ConObsUpdOp: Operation,
 LisToSub: Class): Void is
do
 // 1. add ConObs to the UML model and set ConObss
 umlModel.packagedElement.add(ConObs)
 ConObss.add(ConObs)
 // 2. add ConObsSts to ConObsSt
 ConObsSt.type := ConObsStType
 ConObsSts.add(ConObsSt)
 // 3. make ConObs inherit from Observer
 createGeneralization(ConObs, Observer)
 // 4. add observer state for ConObs
 ConObs.ownedAttribute.add(ConObsSt)
 // 5. create an association between ConObs and LisToSub
 var newasso: uml::Association init Association.new
 newasso := createAssociation(umlModel, ConObs, "",
 1, 1, false, AggregationKind.none, LisToSub, "subject",
 1, 1, true, AggregationKind.none)
 ConobsAsConsub.add(newasso)
 // 6. add the Update operation for ConObs
 ConObs.ownedOperation.add(ConObsUpdOp)
 ConObsUpdOps
```

Table 11 Signature of a new version for AddConcreteObserver

```
operation AddConcreteObserverE(ConObsName: String,
 ConObsStName: String, ConObsStType: Type,
 ConObsUpdOpName: String, LisToSub: uml::Class):
 Class[1..1]
```

Table 12 Example usage of pattern evolution

```
operation main(): Void is
do
 var AnOP: ObserverPattern init ObserverPattern.new
 AnOP.CreateInitStruct()
 var newsub: Class init Class.new
// add a new concrete subject
 newsub := AnOP.AddConcreteSubjectE("MyConSub1",
   void, AnOP.IntPrimType, void, void)
// create a new concrete observer and make it listen
// to the newly created concrete subject newsub
 AnOP.AddConcreteObserverE("MyConObs1", void,
 AnOP.IntPrimType, void, newsub)
// create another concrete observer and also make it listen
// to the newsub
 AnOP.AddConcreteObserverE("MyConObs2", void,
   AnOP.IntPrimType, void, newsub)
 AnOP.CreatePatternSpecification()
// check the invariant to verify if it is a legal instance
 AnOP.checkInvariants
// save as a UML model
 AnOP.save("ObserverPatternd3")
end
```

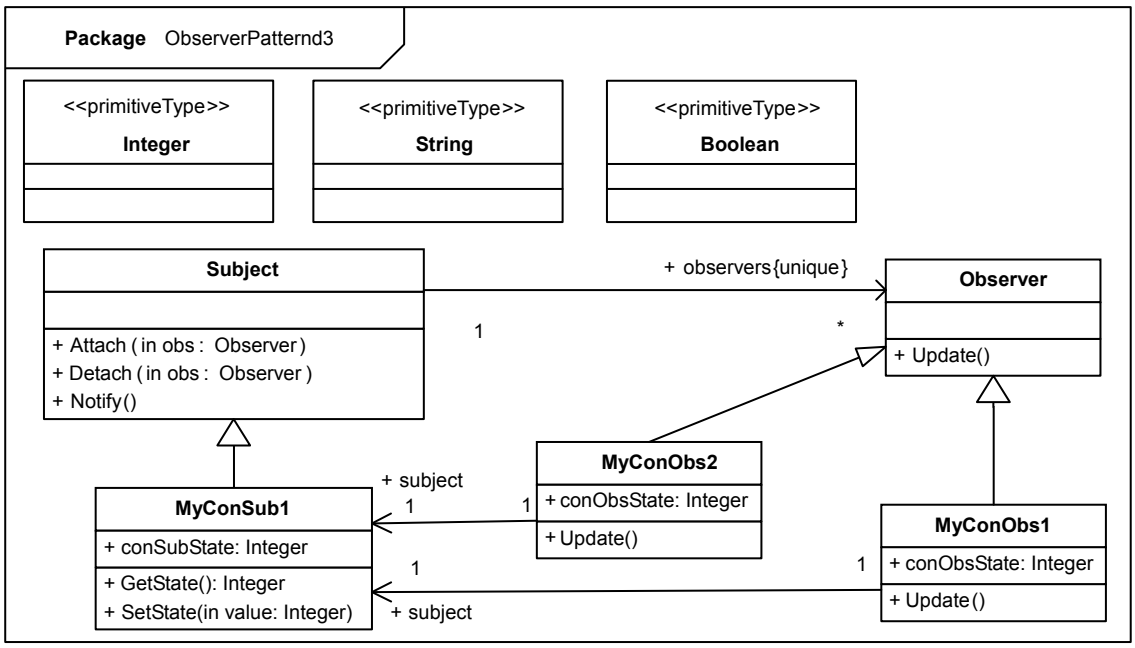

Fig. 5 A produced instance of the Observer pattern

3.3 Pattern implementation

Pattern implementation concerns transforming design level's pattern instances into a specific implementation platform. Traditional tools (e.g., Rational Rose, ArgoUML) generate only code skeletons from design models, which are mainly UML models. They focus on the structural features of a pattern but ignore its behavioral features. The behavioral features of structural patterns are not significant, and thus generating code skeletons is enough. However, for behavioral patterns, ignoring their features is a big loss because their behavioral aspects are equally important as their structural aspects.

To fully capture the behavioral features of a pattern, we introduce the concept of dynamic relation (Gamma *et al*., 2004). According to the action theory embedded in situation calculus (Reiter, 2001), a dynamic system consists of objects, dynamic relations (fluents) between objects, and a set of actions that could change the truth-value of the dynamic relations. Analogously, during runtime, a pattern instance is a dynamic system in the memory, which consists of several objects with dynamic relations among them. Operations defined on these objects are actually value-changing actions of the dynamic relations of the pattern. This view stresses the importance of defining explicitly the dynamic relations involved in the behavior of a pattern, and generating formal specifications for pattern-level operations.

Considering the behavior of the Observer pattern, two dynamic relations are defined, namely Attached and Updated. The Attached relation describes the relationship between the objects of Subject and Observer. For *sub:Subject* and *obs:Observer,* if *obs* is attached to *sub*, then the relation Attached is satisfied; otherwise, the relation is unsatisfied. To incorporate the Attached relation into the previous structural definition, we implement it as a UML operation (return type as Boolean), which is defined in the *Subject* class. Table 13 shows the definition of the EReference and the instantiation of the Attached relation. In the context of the Subject class, Attached needs only to take one parameter, *obs:Observer*, because it can always access the *Subject* instance by self-reference.

The Updated relation describes the relationship between a concrete subject and a concrete observer. For *conSub:Subject* and *conObs:Observer*, if *conSub*'s subject state is consistent with *conObs*'s subject state, then Updated is true; otherwise, it is false. Soundarajan and Hallstrom (2004) suggested the meaning of consistency be left undefined, because each observer may interpret it differently. To provide flexibility, we define an Updated operation for each concrete observer such that it can interpret

consistency with different implementations. The Updated relation is added to the Observer pattern in Table 14.

Table 13 Definition of dynamic relation: Attached

```
// EReference definition of Attached
reference Attached: Operation[1..1]
reference AttachedPara1: Parameter[1..1]
...
// instantiation of Attached
// create Attached as an operation of Subject
Attached := createOperation(Subject, "Attached", false)
// create one parameter obs of type Observer
AttachedPara1 := createParameter(Attached, "obs",
  Observer)
// set return type as Boolean
setRetType(Attached, BoolPrimType)
```

Table 14 Definition of dynamic relation: Updated

```
// EReference definition of Updated
reference ConUpdated: Operation[0..*]
...
// instantiation of Updated relation in
// AddConcreteObserver
var newupdated: Operation init Operation.new
newupdated.name := "Updated"
ConUpdated.add(newupdated)
// set newupdated as an operation of the new ConObs
ConObs.ownedOperation.add(newupdated)
setRetType(newupdated, BoolPrimType)
```

After explicitly defining dynamic relations, it is possible to specify precisely the pattern behavior by defining pre- and post-conditions for each pattern-level operation such as the value-changing action. The specification of each operation is based on its corresponding dynamic relations. In this work, we develop a model-to-text (M2T) template to generate Java codes and JML specifications for UML models. The definition of the structural transformation is straightforward because the UML and Java are both in the OO paradigm. Specifically, we define a transforming template in Acceleo (http://www.acceleo.org), which is a programmatic implementation of the M2T specification of the OMG. The template maps the UML class to Java class, UML operation to Java method, among others. Regarding pattern behavior, we define an EOperation *createPatternSpec* to generate JML specifications for each pattern-level operation. Although the meanings of the pattern behavioral specifications depend on the meanings of the dynamic relations, the forms of these specifications are independent of their meanings, and depend only on their forms. Furthermore, the templates can generate JML specifications for all pattern instances rather than a particular one. These templates are parameterized by a set of variables whose values are obtained by consulting the structure of the current pattern instance. As an example, Table 15 shows the template of the Observer pattern's behavioral specifications in which variables are italicized. The values of variables are obtained by a set of helper functions that navigate through pattern structure. The generated specifications are properly inserted into the UML model by creating pre-, post-, and body-constraints for UML operations, and are put in the right place during the implementation by using the Acceleo template. By executing the operation *main*() from Table 16, the UML model generated is as shown in Fig. 6. We can see that the dynamic relations are defined as Boolean operations in corresponding classes.

Table 15 Behavioral template of the Observer pattern

```
// default implementation of dynamic relations
// implementation of Attached relation
{ return this.obssEndName.contains(AttachedPara);
}
// implementation of Updated relation for each concrete
// observer
{ return this.lisToName.subProName == this.obsProName;
}
...
// specification of pattern-related operations
// Pre, Post and Imp for Attach
// @ requires !Attached(AtcParaN);
// @ ensures Attached(AtcParaN);
{ this.obssEndName.add(AtcParaN);
}
// Pre, Post and Imp for Detach
// @ requires Attached(AtcParaN);
// @ ensures !Attached(AtcParaN);
{ this.obssEndName.remove(AtcParaN);
}
// Pre, Post and Imp for Notify
// @ requires true
// @ ensures (forall ObserverN obs;
// Attached(obs) ==> obs.Updated());
{ for (Iterator<ObserverN>itobs =
   this.obssEndName.iterator(); itobs.hasNext(); )
  { ObserverN tempobs = itobs.next();
   tempobs.UpdateN();
  }
}
```

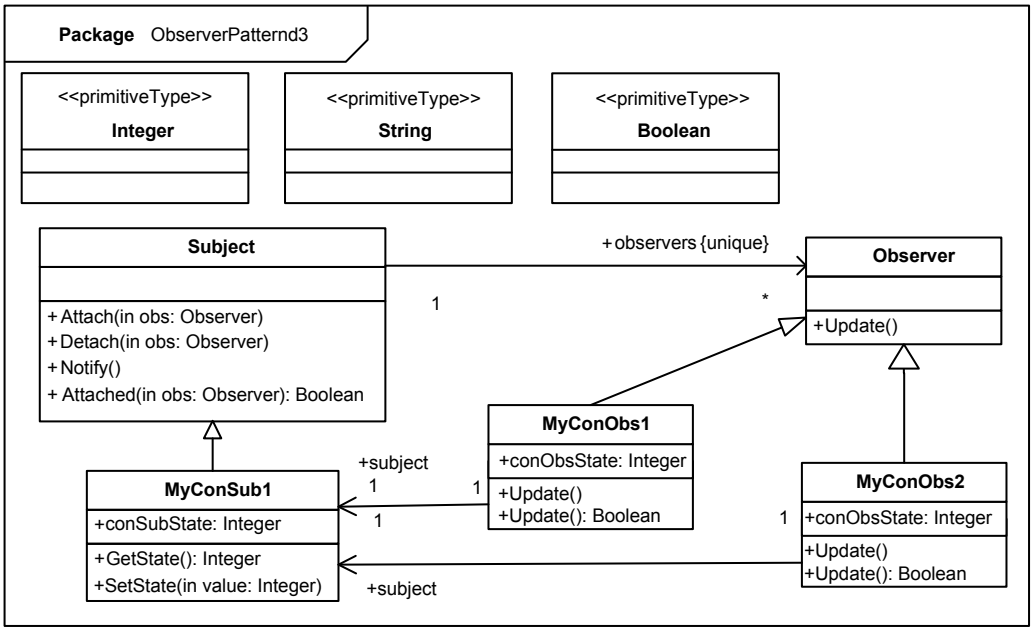

Fig. 6 A generated instance of the Observer pattern with dynamic relations

Table 16 Example codes of behavioral specification generation

```
operation main(): Void is
do
 var AnOP:ObserverPattern init ObserverPattern.new
 AnOP.CreateInitStruct()
 var newsub:uml::Class init uml::Class.new
 newsub := AnOP.AddConcreteSubjectE("MyConSub1",
  void, AnOP.IntPrimType, void, void)
 AnOP.AddConcreteObserverE("MyConObs1", void,
  AnOP.IntPrimType, void, newsub)
 AnOP.AddConcreteObserverE("MyConObs2", void,
  AnOP.IntPrimType, void, newsub)
 // generate behavioral specification
 AnOP.CreatePatternSpecification()
 AnOP.checkInvariants
 AnOP.save("ObserverPatternd3")
end
```

Table 17 JML annotated Java codes

```
public abstract class Subject {
public List<Observer> observers;
// @ ensures (forall Observer obs;
// Attached(obs) ==> obs.Updated());
public void Notify()
{ for (Iterator<Observer> itobs =
   this.observers.iterator(); itobs.hasNext(); )
  { Observer tempobs = itobs.next();
    tempobs.Update();
  }
}
public Boolean Attached(Observer obs)
{ return this.observers.contains(obs);
}
// @ requires !Attached(obs);
// @ ensures Attached(obs);
public void Attach(Observer obs)
{ this.observers.add(obs);
}
// @ requires Attached(obs);
// @ ensures !Attached(obs);
public void Detach(Observer obs)
{ this.observers.remove(obs);
}
}
```

At this time, the behavioral specifications are already embedded into the UML model as constraints in the corresponding operations. By executing the Acceleo template, JML-annotated Java codes are generated. The generated Java codes of the *Subject* class are shown in Table 17.

To conclude, we propose an approach of transforming pattern structure and behavior into Java and JML to achieve pattern implementation. We believe that this approach is the first attempt to utilize the concept of dynamic relation to preserve precisely the pattern behavioral intentions in its implementation. The JML specifications generated can provide guidelines for implementation and can be checked during runtime to verify the correctness of pattern behavior.

4 Related works

Formal specification of design patterns is center to pattern research and is the foundation for solving pattern-related problems. Eden and Hirshfeld (2001) brought Variable and Set to pattern structural specification, and modeled patterns using higher-order logic. A PSL named LePus, which is equipped with a set of new abstract graphical notations, was proposed to facilitate the understanding of system design. Mapelsden *et al*. (2002) developed Design Pattern Modeling Language (DPML) to specify patterns at an abstract level. The DPML is a completely new language with new concepts, and a tool is developed to instantiate the UML models from their pattern specifications. These imply that designers need to learn new languages (LePus or DPML) in their approaches. Our approach eliminates the effort of learning a new language because our metamodel is based on the UML metamodel itself. Furthermore, the pattern instantiation in our approach is encapsulated in a single operation and can be easily customized by changing the properties of UML elements. First-order logic (FOL) and temporal logic of action (TLA) have been used to describe pattern structure and behavior, respectively (Dong *et al*., 2000; 2007a; Dong, 2002). However, these formalisms impose their own modeling paradigms on patterns, and ignore many details necessary to pattern practice. In addition, specifying the pattern structure and behavior in two different formalisms is inconvenient. Therefore, Taibi and Ling (2003) defined a new language, Balanced Patterns Specification Language (BPSL), to combine structural and behavioral specifications in a single language. However, instead of constructing a unified conceptual model for pattern specification, BPSL focuses only on syntactic combinations of FOL and TLA. In our work, we provide a concept model of situation calculus to seamlessly combine pattern structure and behavior. Dynamic relations are defined as Boolean operations in pattern structure, and pattern-level operations are actions that can change the truth-values of these Boolean operations. Based on this approach, JML specifications can be generated to capture precisely the pattern behavior.

Solving pattern-related problems is a much talked about topic in related research areas. Mapelsden *et al*. (2002) studied pattern instantiation based on DPML and proposed a sophisticated tool to instantiate a pattern instance such as UML models. Pattern evolution was studied by Zhao *et al*. (2007), where evolving rules were expressed using graphical productions. One advantage of our approach is that the problem of pattern instantiation and evolution can be handled by writing a few lines of Kermeta codes in corresponding EOperations. Hannemann and Kiczales (2002) studied pattern implementation by using aspect-oriented programming to improve pattern modularity and flexibility at the code level. Our approach focuses on generating JML specifications to preserve pattern behavior in the implementation.

To complete this section, we point out some pattern-related problems that have not been studied in this work. Pattern visualization concerns how to present visually the role of each participant. One sensible solution is the use of UML profiles. For example, Dong *et al*. (2007b) extended the metamodel of the UML profile to express detailed information. To solve this problem, we plan to define the UML profile for each pattern and programmatically set a stereotype for each participant. Pattern composition concerns combining instances of patterns in a single system design. Dong *et al*. (2007a) studied this issue formally. Our approach already provides the potential for combining instances of pattern, i.e., writing Kermeta programs in which the UML element is assigned to two or more EReferences. For example, a class can be a concrete mediator in the Mediator pattern and a concrete observer in the Observer pattern. In the corresponding evolving operations, we can add the needed structural features for both patterns.

5 Conclusions and future work

In this paper, we propose a metamodeling approach based on metamodeling to formally specify design patterns. Our approach is distinguished by reusing concepts in the UML metamodel, and by providing the designers a simple way to handle several pattern-related problems. By modeling a pattern as an EClass and its participants as EReferences, the designers can programmatically control all the participants and handle pattern-related problems in a straightforward and efficient way. Specifically, we discussed three pattern-related problems and provided

solutions for each of them. Our solutions can be considered as foundations of a pattern management system in which design patterns can be properly instantiated, evolved, and implemented. We have successfully applied our approach to some classic patterns originally described by Gamma *et al*. (2004), including the Factory method pattern, Composite pattern, and Mediator pattern. All these case studies and the source codes can be found at https://github.com/lisa-dou/pattern-spec.

Our approach suggests several directions for further research. We are applying our approach to the 23 classic GoF patterns in order to enrich the library for designers. Another thing we are planning is to build the graphical user interface (GUI) for our pattern management system. Although all the functions are programmatically available, the pattern management system can be used graphically by designers without digging into the underlying library. We also plan to build the GUI by using the tool chains in Eclipse Modeling Framework (EMF), such as the Graphical Modeling Framework (GMF). Lastly, based on our approach for pattern specification, we will study more pattern-related problems, such as pattern visualization and composition, to provide a more complete solution for designers.

References

Albin-Amiot, H., Guehéneuc, Y.G., 2001. Meta-Modeling Design Patterns: Application to Pattern Detection and Code Synthesis. Proc. 1st ECOOP Workshop on Automating Object-Oriented Software Development Methods, p.1-35.

Atkinson, C., Kuhne, T., 2003. Model-driven development: a metamodeling foundation. *IEEE Softw.*, **20**(5):36-41. [doi:10.1109/MS.2003.1231149]

Dong, J., 2002. Design Component Contracts: Model and Analysis of Pattern-Based Composition. PhD Thesis, Computer Science Department, University of Waterloo, Ontario, Canada.

Dong, J., Alencar, P.S.C., Cowan, D.D., 2000. Ensuring Structure and Behavior Correctness in Design Composition. Proc. 7th IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems, p.279-287. [doi:10.1109/ECBS.2000.839887]

Dong, J., Alencar, P.S.C., Cowan, D.D., Sheng, Y., 2007a. Composing pattern-based components and verifying correctness. *J. Syst. Softw.*, **80**(11):1755-1769. [doi:10.1016/j.jss.2007.03.005]

Dong, J., Yang, S., Zhang, K., 2007b. Visualizing design patterns in their applications and compositions. *IEEE Trans. Softw. Eng.*, **33**(7):433-453. [doi:10.1109/TSE.2007.1012]

Eden, A.H., Hirshfeld, Y., 2001. Principles in Formal Specification of Object Oriented Architectures. Proc. 11th Conf. of the Centre for Advanced Studies on Collaborative Research, p.3.

Eden, A.H., Hirshfeld, Y., Kazman, R., 2006. Abstraction classes in software design. *IEE Proc.-Softw.*, **153**(4):163-182. [doi:10.1049/ip-sen:20050075]

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 2004. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, MA, USA.

Hannemann, J., Kiczales, G., 2002. Design Pattern Implementation in Java and AspectJ. Proc. 17th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, p.161-173. [doi:10.1145/582419.582436]

Leavens, G.T., Baker, A.L., Ruby, C., 1998. JML: a Java Modeling Language. Formal Underpinnings of Java Workshop.

Mapelsden, D., Hosking, J., Grundy, J., 2002. Design Pattern Modelling and Instantiation Using DPML. Proc. 40th Int. Conf. on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, p.3-11.

Reiter, R., 2001. Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, Cambridge, MA, USA.

Soundarajan, N., Hallstrom, J.O., 2004. Responsibilities and Rewards: Specifying Design Patterns. Proc. 26th Int. Conf. on Software Engineering, p.666-675. [doi:10.1109/ICSE.2004.1317488]

Taibi, T., Ling, D.N.C., 2003. Formal specification of design pattern combination using BPSL. *Inf. Softw. Technol.*, **45**(3):157-170. [doi:10.1016/S0950-5849(02)000195-7]

Zhao, C.Y., Kong, J., Zhang, K., 2007. Design Pattern Evolution and Verification Using Graph Transformation. Proc. 40th Annual Hawaii Int. Conf. on System Sciences, p.290-297. [doi:10.1109/HICSS.2007.169]