

Chapter V

Formal Specification and Verification of Design Patterns

Jing Dong, University of Texas at Richardson, ISA

Paulo Alencar, University of Waterloo, Canada

Donald Cowan, University of Waterloo, Canada

Abstract

This chapter introduces our approaches to formal specification of the structural and behavioral aspects of Design patterns. We investigate the logic-based formalisms in our specification methods and demonstrate the applications of these methods in examples. Our formal specifications methods not only help for rigorous, precise, and unambiguous descriptions of Design patterns, but also allow us to verify the consistencies of Design pattern applications and compositions.

Introduction

A Design pattern (Gamma, Helm, Johnson, & Vlissides, 1995) documents expert solutions to a recurring problem in a particular context. Design patterns are then applied in many applications to improve the quality of the resulting software system and the productivity of the associated personnel. Thus, the specification and description of Design patterns are critical to their successful application.

Design patterns are typically described informally to facilitate understanding by software developers. However, there are several drawbacks to the informal representation of Design patterns. First, informal specifications may be ambiguous, and Design patterns may not be able to be expressed precisely in an informal language. Second, informal representation may not be amendable to rigorous analysis. Each Design pattern may have some particular properties that characterize it, and the application of a Design pattern should maintain these properties. Formal specification allows the use of automated tools to check such properties. Third, formal specifications of Design patterns also form the basis for the discovery of Design patterns in large software systems. Rigorous descriptions of Design patterns allow reverse engineering tools and techniques to be built to discover the Design patterns used in a design.

We have investigated several approaches to the formal specification of Design patterns and their composition in the past decade. In particular, we separated the structural and behavioral aspects of Design patterns and proposed specification methods based on first-order logic, temporal logic, temporal logic of action, process calculus, and Prolog. We also explored verification techniques based on theorem proving. The main objective of this chapter is to describe our investigations on formal specification techniques for Design patterns, and then demonstrate using these specifications as the methods of reasoning about Design pattern properties when they are used in software systems.

Background

The application and composition of Design patterns is still an ad-hoc process as designers struggle to discover the inconsistencies and errors in Design pattern application and composition. This discovery process is mostly based on personal experience. With the development of software tools, such as Rational Rose, which can transform an object-oriented design into implementations of different object-oriented programming languages, software developers have been partially released from tedious and error-prone work in mapping a design into an implementation. However, design errors are changed into implementation errors, which become even harder to detect because they are transformed and blended into complex implementation structures. Failure to discover the errors in software design and development may result in huge losses of money and even human lives, as demonstrated in the failed launch of the \$500 million Ariane 5 (Jezequel & Meyer, 1997) in 1996 and the unsuccessful automation of the London Ambulance Service (Dalcher, 1999).

Design analysis can assist in the discovery of errors in a design early in the development phase and reduce the cost of finding and correcting them downstream. Formal specification

and verification techniques are useful for design analysis in that formal specifications are more precise, clear, expressive, and unambiguous than informal representations, such as graphical and textual notations. Formal specification can be the basis for formal verifications, which can help detect errors. Thus, a number of papers on formalizing Design patterns have appeared in the literature.

A formal specification approach based on logics is presented in Eden and Hirshfeld (2001). Some graphical notations are also introduced to improve the readability of the specifications. While their approach concentrates on the visual formalism of Design patterns, our specifications of Design patterns tend to reduce the ambiguity and allow reasoning about the properties.

Similar to our ideas presented in Dong, Alencar, and Cowan (2000), Taibi and Ngo David (2003) propose specifying the structural aspect of Design patterns in the first order logic (FOL) and the behavioral aspect in the temporal logic of action (TLA). Their approach using FOL and TLA differs from ours, thus resulting in different specifications of Design patterns.

The structural and behavioral aspects of Design patterns in terms of responsibilities and rewards are formally specified in Soundarajan and Hallstrom (2004). Following the ideas of the design by the contract approach in Meyer (1992), the structural and behavioral specifications are captured as responsibilities, whereas the rewards capture the benefits of applying the pattern with the expected behavior in a system. Similar to our goal of avoiding over-specification, their formal specifications retain flexibility.

The composition of two Design patterns based on a specification language (DisCo) has been discussed in Mikkonen (1998). The behavior of each pattern is formalized as a layer in DisCo. The composition of Design patterns is defined as a refinement on the layers of specifications. Similar to our approach, the refinement is property-preserving, such that the refinements of one pattern by another preserves all properties of both patterns.

Formal specification of Design patterns and their composition based on the language of temporal ordering specification (LOTOS) is proposed in Saeki (2000). In particular, the behavioral aspect of the Command and Composite patterns and their combination is specified. In addition to behavior, we are interested in the structure. Our work also presents methods for reasoning about the specifications of Design patterns.

Law-governed support for realizing Design patterns has been investigated in Pal (1995). Some rules and constraints of Design patterns have been defined. However, the property checking is performed at implementation level, whereas our form of checking work is during design.

Main Thrust of the Chapter

In this section, we present the main issues related to the description of Design patterns. We then introduce our specification approaches based on first-order logic, temporal logic of action, and Prolog.

Issues, Controversies, Problems

Design patterns are a means to capture successful software development design practice within a particular context. Patterns should not be limited in what they can describe and can be used to encapsulate good design practices at both the specification and implementation levels. Thus, Design patterns can be applied at many different levels of abstraction in the software development lifecycle, and can focus on reuse within architectural design as well as detailed design and implementation. In fact, a system of patterns for software development should include patterns covering various ranges of scale, beginning with patterns for defining the basic architectural structure of an application and ending with patterns describing how to implement a particular design mechanism in a programming language.

The descriptions of Design patterns are generally informal diagrams and text, which usually make them relatively easy to understand. This informality also facilitates the flexibility of Design patterns. However, the problems with informal specification of Design patterns (or informal specification in general) are ambiguity and difficulty to reason about. When there is ambiguity in the descriptions, the designers may make mistakes while applying a Design pattern. Such mistakes are difficult to find without proper reasoning techniques and tools.

Design patterns are typically described in terms of several aspects, such as intent, motivation, structure, behavior, sample code, and related patterns. Although not all aspects of a Design pattern can be formalized, some functional aspects (structure and behavior) are amenable to formal specification and rigorous reasoning. In this chapter, we present our formal techniques and methods for the specification and verification of the structural and behavioral aspects of Design patterns. In the rest of this chapter, for simplicity, we use the phrase “formalizing Design patterns” to mean formalizing both the structural and behavioral aspects of Design patterns.

Solutions and Recommendations

In this section, we present our formal specification techniques for Design patterns based on first-order logic, temporal logic of action and Prolog. In addition, we investigate the verification of Design patterns.

First-Order Logic

In specifying structural aspects of Design patterns, we investigated a formal specification method using general first-order logic to represent each Design pattern structure as a logic theory (Dong et al., 2000).

To illustrate the problem, let us consider the Composite pattern and the Iterator pattern from (Gamma et al., 1995) as examples. The structural aspect of the Composite and Iterator patterns is depicted in Figure 1. The Component class is an abstract class which defines the interfaces of the pattern. The Composite and the Leaf classes are concrete classes defining the attributes and operations of the concrete components. The Composite class can contain a group of children, whereas the Leaf class cannot. The Composite pattern is often used to

represent part-whole hierarchies of objects. The goal of this pattern is to treat composition of objects and individual objects in the composite structure uniformly. In the Iterator pattern, the Iterator class is an abstract class which provides the interfaces of the operations, such as *First*, *Next*, *IsDone*, *CurrentItem*, to access the elements of an aggregate object sequentially without exposing its underlying representation. The ConcreteIterator class inherits the operation interfaces from the Iterator class and defines concrete operations which access the corresponding concrete aggregate. The Aggregate class defines a common interface for all aggregates that the Iterator accesses. The ConcreteAggregate class defines an operation to create the corresponding concrete Iterator.

The representations of the Composite pattern and the Iterator pattern contain predicates for describing classes, state variables, methods, and their relations. More precisely, the following sorts denote the first-class objects in a pattern: *class* and *object*. We also make use of sorts *bool* and *int*. The signature for the Composite pattern is:

- **Add:** class \rightarrow bool
- **Remove:** class \rightarrow bool
- **GetChild:** class \times int \rightarrow bool
- **Operation:** class \rightarrow bool
- **Variable:** class \times object \rightarrow bool
- **Inherit:** class \times class \rightarrow bool

The Signature for the Iterator pattern is:

- **CreateIterator:** \rightarrow bool
- **New:** class \rightarrow bool
- **First:** \rightarrow bool
- **Next:** \rightarrow bool
- **IsDone:** \rightarrow bool
- **CurrentItem:** \rightarrow bool
- **Variable:** class \times object \rightarrow bool
- **Inherit:** class \times class \rightarrow bool

Table 1 contains (partial) theories associated with the two patterns. θ_C denotes the theory of the Composite pattern and θ_I denotes the theory of the Iterator pattern. The theory θ_C is divided into three class groups and one relation group. The first group defines the abstract class Component and four method interfaces. The second group corresponds to the Leaf class. The third group contains theories about the Composite class, which include the definition of a state variable and the operations applied to it. The last group defines two inheritance relations. The first class in each inheritance relation is the parent class and the second class is the child class. The theory θ_I is divided into five groups. The first four groups contain theories about four classes in the pattern. The last group contains two inheritance relations.

Prolog

In another approach (Alencar, Cowan, Dong, & Lucena, 1999), Design patterns are represented in Prolog (Clocksin & Mellish, 1987) and stored in a Prolog database. There are several advantages of using Prolog as a repository of design knowledge. First, the representations of these Design patterns can be reused by instantiating the corresponding Prolog rules of each pattern when they are applied to produce a concrete domain-specific pattern representation. Second, the properties and constraints of each Design pattern can be described as Prolog rules, and these rules can be used to check the consistencies of the Design patterns. Third, the addition and removal of structural facts about Design patterns can be accomplished by using the Prolog *assert* and *retract* clauses. Fourth, the transformation of the Design pattern representations in Prolog to code templates can be performed by a transformation tool such as Draco-PUC (Leite, Sant' Anna, & Freitas, 1994).

Figure 1. The composite and iterator patterns

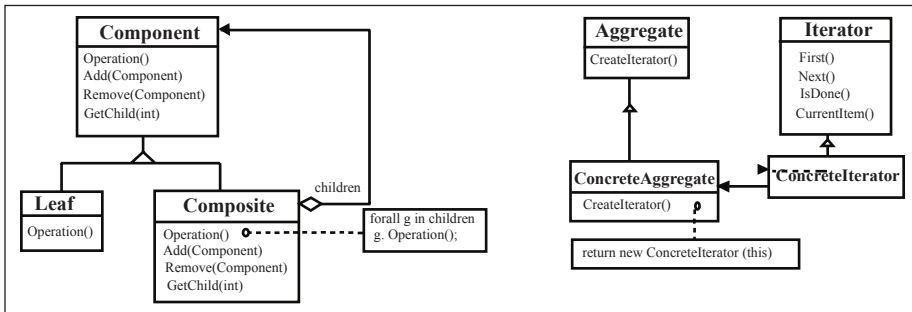


Table 1. Partial composite pattern and iterator pattern theories

θ_c	θ_i
AbstractClass(Component) Operation(Component) Add(Component) Remove(Component) GetChild(Component, int)	AbstractClass(Aggregate) CreateIterator
Class(Leaf) Operation(Leaf)	Class(ConcreteAggregate) CreateIterator \rightarrow New(ConcreteIterator)
Class(Composite) Variable(Component, Children) Operation(Composite) \rightarrow [$\forall g$ [Children(g) \rightarrow Operation(g)]] $\forall v$ [Add(v) \rightarrow Children(v)] $\forall v$ [Children(v) \rightarrow Remove(v)] $\exists v$ [Children(v) \wedge GetChild(v, int)]	AbstractClass(Iterator) First Next IsDone CurrentItem
Inherit(Component, Leaf) Inherit(Component, Composite)	Class(ConcreteIterator) Variable(Aggregate, aggregates)
	Inherit(Aggregate, ConcreteAggregate) Inherit(Iterator, ConcreteIterator)

Design patterns are represented in terms of object-oriented design primitives in a predicate-like format. Each design primitive consists of two parts: *name* and *argument*. The *name* part contains the name of a feature or a relationship in object-oriented design, such as class or inheritance. The *argument* part contains general information about a feature or a relation such as the information on the participants of an inheritance relationship. In the following, we present the syntax and the meaning of the design primitives used in this chapter:

- **Class(C):** C is a class.
- **Abstractclass(C):** C is an abstract class.
- **Inherit(A, B):** B is a subclass of A.
- **Attribute(C, A, V, T):** V is the name of an attribute in class C with type T. T is optional. A describes the access right of this attribute, that is, public, private, or protected.
- **method(C, A, F, R, P₁, T₁, P₂, T₂, ...):** F is a method of a class C. A describes the access right of this method, that is, it can be public, private, or protected. R describes the return type.

The method's parameters and their types are P₁, T₁, P₂, T₂, ..., respectively, and this part is optional. The return type R is also optional if the method has no parameters.

- **Return(C, F, V):** V is the return value of the method F in the class C.

This predicate represents a pointer to the dynamic instantiation of class C₂ in the method F of class C₁. P is the initial value of the class C₂. P can contain zero or more parameters depending on the number of parameters for the constructor of class C₂.

- **Assign(C, F, L, R):** Right value R is assigned to left variable L in the method F of the class C.
- **Invoke(C, C_f, O, O_f, P):** A method O_f which belongs to the object O is invoked in the method C_f of the class C, where P is the parameter of the method O_f. P can contain zero or more parameters depending on the number of parameters the method O_f.
- **Member(E₁, S₁, E₂, S₂, ...):** E₁ is an element of set S₁. E₂ is an element of set S₂, and so on. When universal quantification *forall* and *member* are used together, it enumerates set S₁, S₂, ..., S_n simultaneously, that is, the first elements of all sets are enumerated first, then the second elements.

A higher level of abstraction is provided by introducing pattern primitive operators. Pattern primitive operators are represented in terms of design primitive operators and they allow general object-oriented schemas such as delegation, aggregation, and polymorphism to be defined. Pattern primitive operators can capture the subpatterns, which occur frequently in the declarative representation of Design patterns. They can also be used to change, transform,

or make the declarative representation evolve. This operator can assist with the evolution of the pattern schema and also with the application of this pattern.

As an example, a pattern primitive operator called abstract coupling or *polymorph* can be represented in Prolog as follows:

```
polymorph(Interface, Imp, Binding, ConcretelmpSet, ImpOperation, Operation) :-
  assert(abstractclass(Imp)),
  assert(method(Imp, public, ImpOperation)),
  forall(member(Concretelmp, ConcretelmpSet),
    assert(inherit(Imp, Concretelmp)) ),
  forall(member(Concretelmp, ConcretelmpSet),
    assert(class(Concretelmp)) ),
  forall(member(Concretelmp, ConcretelmpSet),
    assert(method(Concretelmp, public, ImpOperation))),
  assert(abstractclass(Interface)),
  assert(attribute(Interface, private, Binding, Imp)),
  assert(method(Interface, public, Operation)),
  assert(invoke(Interface, Operation, Binding, ImpOperation)).
```

```
extend_polymorph(Imp, NewConcretelmpSet, ImpOperation) :-
  forall(member(Concretelmp, NewConcretelmpSet),
    assert(inherit(Imp, Concretelmp)) ),
  forall(member(Concretelmp, NewConcretelmpSet),
    assert(class(Concretelmp)) ),
  forall(member(Concretelmp, NewConcretelmpSet),
    assert(method(Concretelmp, public, ImpOperation))).
```

```
retract_polymorph(Imp, OldConcretelmpSet, ImpOperation) :-
  forall(member(Concretelmp, OldConcretelmpSet),
    retract(inherit(Imp, Concretelmp)) ),
  forall(member(Concretelmp, OldConcretelmpSet),
    retract(class(Concretelmp)) ),
  forall(member(Concretelmp, OldConcretelmpSet),
    retract(method(Concretelmp, public, ImpOperation))).
```

The Prolog rules of *polymorph* represent the structure of polymorphism. The arguments of the *polymorph* predicate denote the generic elements, for example, class, attribute, or method. *Interface* and *Imp* are abstract classes. *Binding* represents an object reference, which is a state variable of the *Interface* class. *ImpOperation* and *Operation* are two important methods. *ConcretelmpSet* defines a set of concrete classes which may include, for example, *ConcretelmpA* and *ConcretelmpB*. This Prolog representation contains more information

than the UML representation, which cannot, for example, represent an undetermined number of classes. All the arguments will be instantiated by class and operation names, and these names result in specific domain knowledge.

The Prolog operators *assert* and *retract* are used to insert or remove certain facts into or from a Prolog database, respectively. The *forall* predicate represents the universal quantification operator. It can quantify over a set of class names and add the corresponding facts about each class name into the Prolog database. For instance, the Prolog rule `forall(member(ConcreteImp, ConcreteImpSet), assert(inherit(Imp, ConcreteImp)))`, corresponds to the following first-order logic formula:

$$\forall \text{ConcreteImp} \in \text{ConcreteImpSet} : \text{inherit}(\text{Imp}, \text{ConcreteImp}).$$

The nondeterminism in *polymorph* leaves space for evolution, that is, for adding or removing concrete classes which inherit from the abstract class *Imp*. The addition or removal of one such class can be performed by the *extend_polymorph* or the *retract_polymorph* rules, respectively, which in turn *assert* and *retract* the corresponding facts related to the insertion or removal of this concrete class.

Notice that the primitive operators represent basic constituents of an object-oriented design and that the structural information related to Design patterns can be represented by pattern primitive operators and design primitive operators. For example, *polymorph* is used to represent the Bridge, State, and Strategy patterns. In this way, the design of an object-oriented application can be assembled by combining the design components stored in the Prolog database. In addition, the evolution (addition or removal of design components) of a software system design can be achieved by applying specific Prolog rules.

The Prolog representation of the structural aspect of the Bridge pattern is shown as follows. It uses the *polymorph* pattern primitive as a design subcomponent.

```
bridge(Abstraction,Implementor,Imp,RefinedAbstractionSet,
ConcreteImplementorSet,ImpOperation,Operation) :-
  polymorph(Abstraction, Implementor, Imp, ConcreteImplementorSet, ImpOperation,Operation),
  forall(member(RefinedAbstraction,RefinedAbstractionSet),
    assert(inherit(Abstraction,RefinedAbstraction))),
  forall(member(RefinedAbstraction,RefinedAbstractionSet),
    assert(class(RefinedAbstraction)) ).
```

```
extend_bridge_abstract(Abstraction,NewRefinedAbstraction):-
  assert(inherit(Abstraction, NewRefinedAbstraction)),
  assert(class(NewRefinedAbstraction)).
```

```
extend_bridge_imp(Implementor, NewConcreteImplementor, ImpOperation) :-
  extend_polymorph(Implementor, NewConcreteImplementor, ImpOperation).
```

```
retract_bridge_abstract(Abstraction, OldRefinedAbstraction):-
  retract(inherit(Abstraction, OldRefinedAbstraction)),
  retract(class(OldRefinedAbstraction)).
```

```
retract_bridge_imp(Implementor, OldConcreteImplementor, ImpOperation) :-
  retract_polymorph(Implementor, OldConcreteImplementor, ImpOperation).
```

The main purpose of the Bridge pattern is to separate the abstraction from its implementation so that they can vary independently. The *bridge* Prolog rule is used to specify the structure facts of the Bridge pattern. It uses the *polymorph* Prolog rule to specify part of the structure facts of the Bridge pattern, including the Abstraction, Implementor, ConcreteImplementor, and their relationships. The two *forall* rules are used to specify the refined abstraction classes and their inheritance relationships to the Abstract class. Since the Bridge pattern allows extensions of both the refined abstractions and the concrete implementations, there are two rules related to *extend_bridge* and two rules related to *retract_bridge*.

When a designer chooses a Design pattern to solve a particular application problem during the actual design, he or she can save this design decision as facts in the Prolog database by applying the rules that correspond to the Design pattern and using as parameters the domain specific names required by the application. In this way, the Prolog facts which represent the structural constituents of that Design pattern will be saved in Prolog database.

There are four major tasks during instantiation. First, to improve readability and understandability, the generic element names (e.g., classes, attributes, or methods) are replaced by application domain names or domain-specific vocabulary. Each Design pattern encapsulates general design practice that is independent on the application domain. The instantiation of a Design pattern applications leads to domain-dependent designs. This replacement or renaming is achieved by instantiating the arguments of the corresponding Prolog rule of the Design pattern.

Second, according to the application requirements, a number of concrete components are created. The structural solution provided by Design patterns often involves an undefined number of concrete classes, which depends on the application. This undefined character is due to the fact that Design patterns are domain-independent abstract design solutions. We capture this Design pattern characteristic by using sets of elements in Prolog. The arguments in the Prolog rules, which represent Design patterns, can be single elements or sets of elements, that is, one argument may represent a set of elements. Arguments related to sets of elements, when instantiated, assume the value of a fixed number of elements.

Third, Design patterns are added as Prolog facts into the design knowledge base. In the representation of the Design pattern, there is a collection of *assert* statements associated with a Prolog rule. The application of the Prolog rule automatically leads to the insertion of the selected facts into the database through these *assert* statements. Through argument instantiation discussed in the previous two tasks, the free variables of the inserted facts are unified with domain-related names.

Fourth, a Design pattern can be extended or retracted if it does not violate the constraints of the pattern. It is important to have, in the documentation of the Design pattern, information about the evolution of the patterns. We also provide rules in Prolog about the evolution of

Design patterns. These rules and constraints restrict the addition and removal of elements to avoid undesired interactions among components of a single Design pattern. However, these rules do not preclude interactions among different Design patterns.

Temporal Logic of Action

In contrast to the structural aspect of Design patterns, we specify the behavioral aspect using the temporal logic of action (TLA) (Lamport, 1994). We choose TLA because it is an axiomatic style of semantic definition suitable for describing both safety and fairness properties.

In TLA, semantics is provided by assigning a meaning $[F]$ to each syntactic object F . The semantics are defined in terms of a mapping from the set of variable names to the collection of values.

We use the following TLA notations: f denotes a list of variables in the old states, whereas f' denotes a list of variables in the new state. $\Box F$ denotes always F , and $\Diamond F$ denotes that eventually F will be true. A describes an action which relates the old state to the new state. $[A]_f$ is equivalent to $A \vee (f' = f)$, which represents that either action A is taken or there is no change on state function f . $\langle A \rangle_f$ is equivalent to $A \wedge (f' \neq f)$ which represents that action A is taken and the state function f has been changed. $Enabled A$ means that it is possible to take the action A . Weak fairness, $WF_f(A)$, is defined as $\Box \Diamond \langle A \rangle_f \vee \Box \Diamond \neg Enabled \langle A \rangle_f$. It asserts that eventually action A must either be taken or become impossible to be taken.

We define a formula Φ that represents the semantics of each Design pattern, meaning that $\sigma[\Phi]$ equals *true* if the behavior σ represents a possible application of the pattern.

For example, the semantics of the *Composite* pattern describes the behavior related to inserting or removing objects of an aggregate. All objects in the aggregate have a common type. The fairness condition is that eventually an insertion or deletion occurs unless both are impossible. Removing an object from an empty aggregate is one example of impossi-

Table 2. Semantics of the composite pattern

$Init_\Phi$	\equiv	$op = \text{"ready"} \wedge children = \text{"emptybag"}$
S_{add}	\equiv	$op = \text{"ready"} \wedge op' = \text{"add"} \wedge child' \in \text{Component} \wedge children' = children$
E_{add}	\equiv	$op = \text{"add"} \wedge op' = \text{"ready"} \wedge children' = children + child$
S_{remove}	\equiv	$op = \text{"ready"} \wedge op' = \text{"remove"} \wedge child' \in \text{Component} \wedge children' = children$
E_{remove}	\equiv	$op = \text{"remove"} \wedge op' = \text{"ready"} \wedge child \in children \wedge children' = children - child$
$S_{getchild}$	\equiv	$op = \text{"ready"} \wedge op' = \text{"get"} \wedge child' \in \text{Index} \wedge children' = children$
$E_{getchild}$	\equiv	$op = \text{"get"} \wedge op' = \text{"ready"} \wedge child' = children(index) \wedge children' = children$
$N_{children}$	\equiv	$E_{add} \vee E_{remove}$
N	\equiv	$N_{children} \vee S_{add} \vee S_{remove} \vee S_{getchild} \vee E_{getchild}$
U	\equiv	$\langle op, child, children, index \rangle$
Φ	\equiv	$Init_\Phi \wedge \Box [N]_u \wedge WF_u(N_{children})$

Table 3. Semantics of the iterator pattern

$Init_{\Psi}$	\equiv	$position = \text{"start"}$
M_{first}	\equiv	$position' = \text{"start"}$
M_{next}	\equiv	$position' = Next(position)$
M_{isdone}	\equiv	$position = \text{"last"} \wedge position' = position$
$M_{current}$	\equiv	$val' = aggregate(position) \wedge position' = position$
M_I	\equiv	$M_{first} \vee M_{next}$
M	\equiv	$M_I \vee M_{isdone} \vee M_{current}$
v	\equiv	$\langle position, val \rangle$
Ψ	\equiv	$Init_{\Psi} \wedge \square [M]_v \wedge WF_v(M_I)$

bility. The semantics of the *Iterator* pattern (Gamma et al., 1995) describes the behavior of traversing an aggregate of objects. It defines the traversing methods which are independent of the structure of the aggregate which they traverse. These traversing methods require primitive behaviors, such as *first* and *next*. An internal state memorizes the current position of traversing. Multiple-traversing can be performed provided that enough state variables are available for recording the current position of each traverse. The fairness condition is that traversing steps will eventually proceed which will update the corresponding internal state variables.

Verification

The main goals of formal specification of Design patterns include precise and unambiguous descriptions of Design patterns and formal verification. In the previous section, we present several approaches to formal specification to achieve the former goal. In this section, we discuss our formal verification techniques to check consistency and prove the correctness of Design patterns and their applications, compositions, and evolutions. In particular, we describe our approaches based on theorem proving techniques.

As presented in the previous section, we specify Design patterns in logic theorems (in first-order logic, TLA, Prolog). Therefore, we can check the consistencies of Design pattern specifications by proving the theorems related to the corresponding requirements.

Consistency checking is not an easy task when graphical notations are used. It requires intuition and experience. Consistency checking is also difficult because of the informal notations. On the other hand, representing Design patterns in formal logic notation allows us to describe the properties and constraints of each Design pattern in a precise way and, thus, to check automatically whether a pattern loses some properties after it is combined with other patterns.

The formal specification of Design patterns allows clear understanding of Design patterns. In addition, it helps the application, composition, and evolutions of Design patterns since

understanding of Design patterns is not the final goal of a designer. Formal specifications can also help on the applications of Design patterns. For example, our formal specification using Prolog not only provides the facts related to a Design pattern, but also describes how a designer can apply the Design pattern using the corresponding Prolog rule of the pattern. In this way, we not only specify the structure of a Design pattern, but also describe the process that the Design pattern can be applied.

Formal specification of Design patterns may also help on the compositions of Design patterns. A large software design may consist of many Design patterns. However, there may be inconsistencies among them. There are many reasons to check the consistency of the composition of Design patterns. For example, when two patterns are combined, they may share some common parts. The part that they are sharing can play one role in one pattern, but another role in the other pattern. This situation may lead to an inconsistent combination. In addition, an existing design may be modified and gradually evolve. This may lead to a situation in which the new design no longer conforms to the properties that its Design patterns must preserve. The manual discovery of the inconsistency in design can be a difficult job without a formalism or tool support. The Design pattern representations in Prolog allow us to take advantage of the deductive facilities of Prolog to find the inconsistencies automatically.

Formal specification of Design patterns can further help on the evolutions of Design patterns. Since change is a constant theme in software development, the application of a Design pattern may evolve in some particular way. In our approach, we provide the formal specification of the evolution process of each Design pattern as a Prolog rule using *extend* or *retract*. In this way, the designers may change the applications of Design patterns in some predefined ways to reduce the mistakes in the evolution process.

In order to facilitate the automation of theorem proving, we take advantage of Prolog's inference engine. More specifically, we specify Design patterns and their applications, compositions, and evolutions as Prolog clauses and rules. The properties of the Design patterns to be checked can be described as Prolog queries, which provide true/false answers. In this way, the theorems can be proved automatically. We may verify the properties of each individual Design pattern, the compositions, and evolutions of Design patterns. For example, each *ConcreteImplementor* class in the Bridge pattern should define an *OperationImp* operation. We may specify this property as a Prolog query to check whether it is satisfied in the Prolog knowledge base. This can be done straightforwardly in our approach. After the Bridge pattern is applied, the designer may change the design by adding more *ConcreteImplementor* class(es). In this case, we may also need to check whether the new *ConcreteImplementor* class has the *OperationImp* operation. This can be checked similarly. When the Bridge pattern is applied and composed with other patterns in an application, there may be inconsistencies among these patterns. For instance, one of the main intents of the Bridge pattern is to separate the Abstraction from its Implementations so that they can vary independently. In Alencar et al. (1999), we presented a case where the Bridge pattern application has interactions with other pattern application, such that the Abstraction and its Implementations are not independent anymore. We found this subtle problem using our formal specification and verification techniques in Prolog. Due to space limitation, more details can be found in Alencar et al. (1999).

Future Trends

In large software systems, more than one Design pattern is generally applied and composed. Although each Design pattern encapsulates good design experience, their composition may not always exhibit good design composition. There may be unexpected conflicts among the Design patterns used in a system. Such conflicts and inconsistencies may result in wrong applications of Design patterns, which are then difficult to find and correct. Formal specification and verification methods and techniques may help to find such errors.

Change is a constant theme of software design and development. Because of constant changes of user requirements, platforms, technologies, and environments, software systems need to be adapted to such changes. One of the important goals of Design patterns is design for change. Thus, most Design patterns encapsulate future changes that may only affect a limited part of a Design pattern. However, the evolution information is generally not explicitly specified in the documentation accompanying each Design pattern. One of the current challenges is to specify the possible changes of a Design pattern explicitly in terms of the evolution process so that the developer may follow such a process when changes are required.

Conclusion

In this chapter, we present our approaches on formal specification and verification of Design patterns and their applications, compositions, and evolutions. We separate structural and behavioral aspects of Design patterns in our specification. We investigate different specification techniques, such as first-order logics, temporal logic of action, and Prolog, because different formalisms are suitable for specifying different aspects of Design patterns. For example, first-order logics and Prolog can be used to specify the structural aspect of Design patterns. First-order logics are a standard formalism that renders specifications which are easy to understand. However, it lacks the automated support for applying the theorems. Prolog, on the other hand, provides a knowledge base such that the specifications can be stored as facts and the corresponding rules can be applied automatically. It is also possible to reason about the properties using the knowledge base supported by Prolog.

In contrast to the structural aspect of Design patterns, the specifications of the behavioral aspect require the formalisms to be able to represent event orders and action sequences. Since first-order logics and Prolog cannot represent states and timing, they are not suitable for the behavioral specifications. We investigated the specifications of the behavioral aspect of Design patterns using temporal logic of actions. As described in the previous section, we describe the behavioral semantics of the Composite and Iterator patterns and show that temporal logic of action is a suitable formalism for specifying behavioral aspects of Design patterns.

In addition to the goal of precise and unambiguous specification of Design patterns, we explore different verification techniques. In this way, we are able to reason rigorously about the applications, compositions, and evolutions of Design patterns.

References

- Alencar, P.S.C., Cowan, D.D., Dong, J., & Lucena, C.J.P. (1999). A pattern-based approach to structural design composition. In *Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference* (pp. 160-165). IEEE CS Press.
- Clocksin, W.F., & Mellish, C.S. (1987). *Programming in Prolog*. Berlin: Springer-Verlag.
- Dalcher, D. (1999). Disaster in London: The LAS case study. In *Proceedings of the 6th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems* (pp. 41-52). IEEE CS Press.
- Dong J., Alencar, P.S.C., & Cowan, D.D. (2000). Ensuring structure and behavior correctness in design composition. In *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems* (pp. 279-287). IEEE CS Press.
- Eden, A.H., & Hirshfeld, Y. (2001). Principles in formal specification of object-oriented architectures. In *Proceedings of the 11th CASCON*. IBM Press.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.
- Jezequel, J.-M., & Meyer, B. (1997). Design by contract: The lessons of Ariane. *IEEE Computer*, 30(1), 129-130.
- Lampert, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 873-923.
- Leite, J.C.S.P., Sant'Anna, M., & Freitas, F.G. (1994). Draco-PUC: A technology assembly for domain oriented software development. In *Proceedings of the 3rd IEEE International Conference of Software Reuse* (pp. 94-100). IEEE CS Press.
- Meyer, B. (1992). Applying "design by contract". *IEEE Computer*, 40-51.
- Mikkonen, T. (1998). Formalizing Design pattern. In *Proceedings of the 20th International Conference on Software Engineering* (pp. 115-124). IEEE CS Press.
- Pal, P. (1995). Law-governed support for realizing Design patterns. *Technology of object-oriented languages and systems* (pp. 25-34).
- Saeki, M. (2000). Behavioral specification of GoF Design patterns with LOTOS. In *Proceedings of the Seventh Asia-Pacific Software Engineering Conference* (pp. 408-415). IEEE CS Press.
- Soundarajan, N., & Hallstrom, J.O. (2004). Responsibilities and rewards: Specifying Design patterns. In *Proceedings of the 26th International Conference on Software Engineering* (pp. 666-675). IEEE CS Press.
- Taibi, T., & Ngo David, C.L. (2003). Formal specification of Design pattern combination using BPSL. *International Journal of Information and Software Technology*, 45(3), 157-170.