On the Composition of Design Patterns

Ian Bayley and Hong Zhu
School of Technology, Oxford Brookes University, Oxford OX33 1HX, UK
(ibayley || hzhu)@brookes.ac.uk

Abstract

Design patterns are usually applied in a composed form with each other. It is crucial to be able to formally reason about how patterns can be composed and to prove the properties of composed patterns. Based on our previous work on formal specification of design patterns and formal reasoning about their properties, this paper focuses on the composition of design patterns. A notion of composition of patterns with respect to overlaps is formally defined based on two operations on design patterns, which are the specialization of a pattern with constraints and the lifting of a pattern with a subset of components as the key. The composition of design patterns is illustrated by the composition of Composite, Strategy and Observer patterns. A case study of the formalisation of the relationship between patterns as suggested by GoF is also reported.

1 Introduction

Design patterns are codified reusable knowledge of solutions to recurring design problems [9, 1]. In the past few years, a large number of software design patterns have been described, cataloged [9, 1] and included in software tools [11, 16, 14]. While each design pattern is usually specified separately, they are rarely used alone. They are almost always used together as a composition. A trivial form of composition is when they are used in different parts of a system without overlaps. However, in most cases, there are overlaps on the parts that design patterns are applied. Thus, this paper is devoted to the composition where there are overlaps.

In our previous work, we developed a technique for the formal specification of design patterns in first order logic. We have successfully specified all 23 design patterns in the classic GoF book [9] on both structural features [2] and behavioral features [5]. A formal theory has been developed for the uses of formal specifications in reasoning about design patterns [3]. In particular, we have investigated how to prove that a concrete design represented in UML diagrams conforms to a design pattern, how to deduce the properties of a design pattern from the specification, and how to reason about the relationship between design patterns. The notion of pattern composition has also been defined and preliminarily studied. This paper further develops the theory and reports a case study.

The remainder of the paper is organized as follows. Section 2 gives the background to the paper with a brief review of our approach to the formalization of design patterns and the reasoning about design patterns. Section 3 is the main body of the paper. It defines the notions of overlaps and composition with overlaps. Properties of such compositions are formally proved. Section 4 illustrates the theory with the example of the MVC pattern, which is the composition of Composite, Strategy and Observer. A case study of the relationships between pattern as suggested in GoF [9] is also reported. Section 5 concludes the paper with a discussion of related works and future work.

2 Background

This section briefly reviews our approach to the formal specification of and to reasoning about design patterns in order to give the background to the remainder of the paper.

Generally speaking, our approach is a meta-modeling approach. Each pattern is a subset of the design models with certain structural and behavioral features. Therefore, the formal specification of patterns is a meta-modeling problem. As in [2, 5], our approach to meta-modeling is first to define the domain of all models by an abstract syntax in the meta-notation GEBNF [21], which stands for Graphic Extension of BNF. It extends the traditional BNF notation with a ‘reference’ facility to define the graphic structures of diagrams. In addition, each syntactic element in a definition of a language construct is assigned with an identifier (called field name) so that a first-order language (FOL) can be induced from the abstract syntax definition. Given a formal definition of the domain of models, for each design pattern, we then define a predicate in the induced first-order language to constrain the models such that each model that satisfies the predicate is an instance of the pattern.
In summary, a meta-model in our approach comprises an abstract syntax in GEBNF plus a first-order predicate. The following expands the specification of an example pattern according to our approach.

2.1 The Domain of Models

In this subsection, we first review the meta-notation GEBNF \[21, 2\] and then use it to define the domain of models for class diagrams and sequence diagrams.

2.1.1 GEBNF Notation

In GEBNF, the abstract syntax of a modeling language is defined as a tuple \( \langle R, N, T, S \rangle \), where \( N \) is a finite set of non-terminal symbols such as \( \text{ClassDiag} \), and \( T \) is a finite set of terminal symbols, such as \( \text{String} \). Non-terminal symbols represent the constructs of the modeling language. Each terminal symbol represents a specific value that can occur in a model, such as a string of characters that can occur in a class node as the name of a class. Furthermore, \( R \subset N \) is the root symbol and \( S \) is a finite set of production rules of the form \( Y ::= \text{Exp} \), where \( Y \in N \) and \( \text{Exp} \) can be in one of the following two forms.

\[
L_1 : X_1, L_2 : X_2, \ldots, L_n : X_n \\
X_1|X_2|\ldots|X_n
\]

where \( L_1, L_2, \ldots, L_n \) are field names, and \( X_1, X_2, \ldots, X_n \) are the fields. Each field can be in one of the following forms: \( Y, Y^*, Y^+, \lfloor Y \rfloor, \lceil Y \rceil \), where \( Y \in N \cup T \).

The meaning of the meta-notation is given in Table 1. Note that where an element is underlined, it is a reference to an existing element on the diagram as opposed to the introduction of a new element.

2.1.2 Meta-Model of UML Diagrams

For the sake of simplicity and to save space, a simplified meta-model of UML class and sequence diagrams \[17\] is defined by removing some less commonly used attributes and flattening the inheritance hierarchy between meta-classes. The following is the definition in GEBNF. More details can be found in \[5\].

\[
\text{ClassDiag} ::= \text{classes} : \text{Class}^+, \text{assocs} : \text{Rel}^*, \text{inherits} : \text{Rel}^*, \text{compag} : \text{Rel}^* \\
\text{Class} ::= \text{name} : \text{String} \\
\text{Operation} ::= \text{name} : \text{String}, \text{params} : \text{Parameter}^* \\
\text{Parameter} ::= \text{name} : \text{String}, \text{type} : \text{Type}, \text{direction} : \text{ParaDirKind}, \text{mult} : \text{Multiplicity} \\
\text{ParaDirKind} ::= \text{“in” | “inout” | “out” | “return”} \\
\text{Multiplicity} ::= \text{[lower : \text{Natural}], [upper : \text{Natural}]}^*\text{“*”} \\
\text{Property} ::= \text{name} : \text{String}, \text{type} : \text{Type}, \text{isStatic} : \text{Bool}, \text{mult} : \text{Multiplicity} \\
\text{Rel} ::= \text{[name} : \text{String}], \text{source} : \text{End}, \text{end} : \text{End} \\
\text{End} ::= \text{node} : \text{Class}, \text{[name} : \text{String}], \text{[mult} : \text{Multiplicity} \\
\text{SequenceDiagram} ::= \text{lifelines} : \text{Lifeline}^*, \text{msgs} : \text{Message}^*, \text{ordering} : (\text{Message}, \text{Message})^* \\
\text{Lifeline} ::= \text{actions} : \text{Activation}^*, \text{isStatic} : \text{Bool}, \text{className} : \text{String}, \text{[objectName} : \text{String}] \\
\text{Activation} ::= \text{start}, \text{finish} : \text{Event}, \text{others} : \text{Event}^* \\
\text{Message} ::= \text{send}, \text{receive} : \text{Event}, \text{sig} : \text{Operation}
\]

Here, \text{Natural} denotes the type of natural numbers, \text{Bool} the type of boolean values, and \text{String} the type of character strings.

2.2 Induced First Order Language

As shown in \[21\], an abstract syntax in GEBNF induces a first-order language for writing first-order predicates as constraints on the models.

In a GEBNF definition, every field \( f : X \) of a term \( T \) introduces a function \( f : T \rightarrow X \). Function application is written \( f(x) \) for function \( f \) and argument \( x \). For example, because \text{opers} : \text{Operation}^* is a field of \text{Class}, then \text{C.opers} denotes the set of operations in class \text{C}. When there is just one class diagram or one sequence diagram, functions on them are written without their arguments, as \text{classes}, \text{lifelines} etc.

From functions induced from a GEBNF syntax, first-order predicates can be defined as usual using relations and operators on sets and basic data types and using logic connectives and quantifiers. Further functions and relations can be defined as usual in the first-order logic. For the sake of readability, we will also use infix and prefix forms for defined functions and relations. Thus, we also write the application of function \( f \) to argument \( x \) with the more conventional prefix notation \( f(x) \).

Let \( f \) be any formula in the first order language induced from a given GEBNF definition of models, and \( m \) be any valid model according to the GEBNF definition. The evaluation of the formula \( f \) on \( m \) in the context of an assignment \( \alpha \) of free variables occurred in \( f \) to elements in \( m \) is defined as usual in the first order language, and we write
Eva_{\alpha}(f, m) to denote the value of the evaluation in the context of \( \alpha \). A formula is called a predicate, if its value is a truth value. In particular, if the evaluation of a ground predicate (i.e. without free variables) on a model \( m \) results in the value true, we say that the model \( m \) satisfies the predicate \( f \), and write \( m \models f \). We will also write \( p \vdash q \) if we can deduce formula \( q \) from \( p \) in the first order logic. By the semantic consistency of first order languages, we have the following proposition.

**Proposition 1.**

Let \( p \) and \( q \) be any given ground predicates on models in a given domain \( D \) defined by a GEBNF definition. If \( p \vdash q \), we have that for all models \( m \in D \), \( m \models p \) implies \( m \models q \).

Here follows some functions and relations used in many patterns. For the sake of space, their formal definitions are omitted, but they can be found in [5].

We write \( C_1 \leftarrow \rightarrow C_2, C_1 \rightarrow \rightarrow C_2, \) and \( C_1 \rightarrow \rightarrow C_2 \) for the aggregate, inheritance and association relations between classes \( C_1 \) and \( C_2 \), respectively.

Let \( C \) be a class. Then \( \text{subs}(C) \) denotes the set of concrete subclasses of \( C \). \( C, \text{op} \) denote the redefinition of \( \text{op} \) for class \( C \). \( \text{isAbstract}(C) \) means class \( C \) is abstract.

Let \( m \) and \( m' \) be messages. We write \( m < m' \) for \( (m, m') \in \text{ordering} \). We define \( \text{fromAct}(m) \) to be the unique activation that \( m \) sends to, \( \text{fromLL}(m) \) to be the unique lifeline that \( m \) is from, and \( \text{fromClass}(m) \) to abbreviate \( \text{fromLL}(m).\text{class} \). Similarly, we define \( \text{toAct}(m), \text{toLL}(m) \) and \( \text{toClass}(m) \). \( \text{trigs}(m, m') \) means that message \( m \) starts (or “triggers”) an activation that sends message \( m' \). For all messages \( m \) and objects \( o \), \( \text{hasReturnParam}(m, o) \) is true if \( o \) is the return parameter for \( m \). If there is only one such \( o \) for a message \( m \), we write \( \text{returns}(m) = o \).

For operations \( \text{op} \) and \( \text{op}' \), \( \text{calls}(\text{op}, \text{op}') \) means \( \text{op} \) calls \( \text{op}' \), and we promote this relation to classes. A much-used predicate is \( \text{callsHook} \), defined when an operation calls another at the root of an inheritance hierarchy.

In the sequel, we define patterns only for design models that are well-formed and consistent with respect to a set of constraints [21]. Such constraints can also be written in the first order logic. Readers are referred to [5] for some examples.

### 2.3 Formal specification of patterns

Both the structural and behavioural features of patterns can be formally specified as predicates on diagrams in the same way as consistency constraints. We have successfully specified all 23 patterns in the GoF book [9]. Here, we only give one example to illustrate the style and to give the background to the discussions on reasoning about design patterns. More examples will be given later in the paper, but without detailed explanations. The readers are referred to [5] for more details and [4] for a complete list of all specifications of GoF patterns.

For each pattern, the formal specification consists of three parts. The first part, entitled Components, declares a set of variables, which are existentially quantified over the scope of all predicates in the specification of the pattern. In this way, it sets the background for the formulae by asserting the existence of certain components in the system design. The second part, entitled Static Conditions, consists of a number of predicates for the structural relations between the components. Such predicates can be evaluated using the information contained in the class diagram of a design. The third part, entitled Dynamic Conditions, consists of a number of predicates for the dynamic behaviour of the system, using information in the sequence diagram of a design, and sometimes in the class diagram too. In the latter case, consistency between the diagrams is ensured by the consistency constraints. Formally, we have the following definition.

**Definition 1 (Formal Specification of DP)**

A formal specification of a design pattern is a triple \( P = \langle V, Pr_s, Pr_d \rangle \), where \( Pr_s \) and \( Pr_d \) are predicates on the domain of UML class diagrams and sequence diagrams, respectively, and \( V \) is a set of declarations of the variables that are free in the predicates \( Pr_s \) and \( Pr_d \). Let \( V = \{ v_1 : T_1, v_2 : T_2, \ldots, v_n : T_n \} \). The semantics of the
Class Adapter pattern has the static condition Adapter the Adaptee and (1) above, root Formally, pattern then the operation must be declared in the root class. is sent from a class that is not explicitly mentioned in the desirable to prove that a concrete design really conforms to a design pattern. The following defines the notion of conformance of a concrete design to a pattern.

Definition 2 (Conformance of a design to a pattern)
Let m be a model and P = \(< V, PR_s, PR_d \) be a formal specification of a design pattern. The model m conforms to the design pattern as specified by P if and only if m \( \models Spec(P) \).

The conformance of a model m to a pattern can be easily proved by finding an assignment \( \alpha \) of variables in V to elements in m and evaluating Pred(P) in the context of the assignment \( \alpha \). If the evaluation results in \( true \), then the model satisfies the specification. An example of such proofs can be found in [3].

The following theorem proves that such a proof of a model’s conformance to a design pattern is valid. The proofs of the theorems given in this section are omitted, but can be found in [3].

Theorem 1 (Validity of conformance proofs)
Let m be a model and P = \(< V, PR_s, PR_d \) be a specification of a design pattern. The model m conforms to the design pattern as specified by P if and only if there is an assignment \( \alpha \) from Vars(P) to the elements in m such that Eval\( _\alpha \)(m, Pred(P)) = true.

Given a formal specification of a pattern, we can infer the properties of the systems that conform to the pattern in first order logic. The following theorem proves that properties deduced from the specification in the first order logic are satisfied by all models of the pattern.

Theorem 2 (Validity of property proofs)
Let P be a formal specification of a design pattern. Spec(P) \( \Rightarrow q \) implies that for all models m such that m \( \models P \) we have that m \( \models q \).

There are a number of different kinds of relationships between patterns. Many of the relationships can be defined as logic relations and proved in first order logic. Specialisation is such a relationship.

Definition 3 (Specialisation relation)
Let P and Q between design patterns. Pattern P is a specialisation of Q, written P \( \Rightarrow Q \), if for all models m, it is the case that m conforms to P implies that m also conforms to Q.

The specialisation relation between two patterns can be proved by inference in the first order logic that Spec(P) \( \Rightarrow Spec(Q) \). The following theorem states that such proofs are valid.

Theorem 3 (Validity of proofs of specialisation relation)
Let P and Q be two design patterns. Spec(P) \( \Rightarrow Spec(Q) \) implies that P \( \Rightarrow Q \), i.e. P is a specialisation of Q.
3 Composition of design patterns

In this section, we will first define some operations on design patterns, then use these operations to formally define the compositions of design patterns.

3.1 Motivating examples

Before we give the formal definition of composition, let’s first see a motivating example.

<table>
<thead>
<tr>
<th>Specification 2 (Composite Pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Components</strong></td>
</tr>
<tr>
<td>- Component, Composite ∈ classes,</td>
</tr>
<tr>
<td>- Leafs ⊆ classes,</td>
</tr>
<tr>
<td>- ops ⊆ Component.ops</td>
</tr>
<tr>
<td><strong>Static Conditions</strong></td>
</tr>
<tr>
<td>- CDR(Component), allAbstract(ops)</td>
</tr>
<tr>
<td>- ∀l ∈ Leafs · (l → Component ∧ ¬(l ← Component))</td>
</tr>
<tr>
<td>- isInterface(Component),</td>
</tr>
<tr>
<td>- Composite → Component</td>
</tr>
<tr>
<td>- Composite ← Component *</td>
</tr>
<tr>
<td><strong>Dynamic Conditions</strong></td>
</tr>
<tr>
<td>- ∃m ∈ messages · (toClass(m) = Composite ∧ m.sig ∈ ops ⇒ ∃m' ∈ messages · calls(m, m') ∧ m'.sig = m.sig)</td>
</tr>
<tr>
<td>- ∃m ∈ messages · (toClass(m) ∈ Leafs ∧ m.sig ∈ ops ⇒ ¬∃m' ∈ messages · calls(m, m') ∧ m'.sig = m.sig)</td>
</tr>
</tbody>
</table>

Consider the Composite and Adapter patterns formally specified in Spec. 2 and 1. Suppose that in the design of a system, we have one leaf class in the Composite pattern and this class is to be adapted using an Adapter pattern. This specific use of two patterns is a composition of patterns Composite and Adapter. To enable this composition to be specified formally, first we need to specialize Composite pattern so that it only contains one leaf component called Leaf rather than in the general case where there is a non-empty set Leaf of leaves. Let this specialised Composite pattern be called Composite1. Moreover, we also need to specify that the component leaf in Composite1 is the adapted component in the Adapter pattern, i.e. the component Leaf in Composite pattern is also the Target component in Adapter. Finally, to make the result of composition generally useful, we will also rename the components so that their roles can be understood from their names. For example, we would like to name the component Leaf in Composite1, which overlaps with the Target in Adapter, to be the AdaptedLeaf. From the above analysis we can see that we need the following operators on design patterns: (a) specialisation of a pattern; (b) composition of pattern with an overlap; (c) renaming the components in a pattern.

Before we formally define the operations and study their properties, let’s see a slightly more complicated composition of design patterns.

<table>
<thead>
<tr>
<th>Specification 3 (Composite1 Pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Components</strong></td>
</tr>
<tr>
<td>- Component, Composite, Leaf ∈ classes,</td>
</tr>
<tr>
<td>- ops ⊆ Component.ops</td>
</tr>
<tr>
<td><strong>Static Conditions</strong></td>
</tr>
<tr>
<td>- CDR(Component), allAbstract(ops)</td>
</tr>
<tr>
<td>- Leaf → Component, ¬(Leaf ← Component)</td>
</tr>
<tr>
<td>- isInterface(Component),</td>
</tr>
<tr>
<td>- Composite → Component</td>
</tr>
<tr>
<td>- Composite ← * Component</td>
</tr>
<tr>
<td><strong>Dynamic Conditions</strong></td>
</tr>
<tr>
<td>- ∃m ∈ messages · (toClass(m) = Composite ∧ m.sig ∈ ops ⇒ ∃m' ∈ messages · calls(m, m') ∧ m'.sig = m.sig)</td>
</tr>
<tr>
<td>- ∃m ∈ messages · (toClass(m) ∈ Leaf ∧ m.sig ∈ ops ⇒ ¬∃m' ∈ messages · calls(m, m') ∧ m'.sig = m.sig)</td>
</tr>
</tbody>
</table>

When we compose Composite with Adapter pattern, we may want to adapt a number of leaf components in the Composite patterns. Each of the adapted leaves may have a different adapter. We can apply the above operations to define that there are two adapted leaves, three adapted leaves, etc. However, we cannot define a single pattern that can be applied to all these situations. To achieve this objective, we need an operation on patterns such that the result of an application of lifting on a pattern P is a pattern P' with the following property: that a design model m conforms to P if it contains at least one set of components that matches pattern P. This operation is called Lift in the sequel.

3.2 Specialisation

Let’s start with the specialisation operation on patterns.

<table>
<thead>
<tr>
<th>Definition 4 (Specialisation operation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let P be given pattern and c be a predicate defined on the components of P, a specialisation of P. A specialisation by imposing a constraint c, written as P[c], is the pattern obtained from P by imposing the predicate c as an additional condition of the pattern. Formally, Vars(P[c]) = Vars(P), Pred(P[c]) = (Pred(P) ∧ c) □</td>
</tr>
</tbody>
</table>

The following theorem states that by applying a specialisation operation [c] on a pattern P, the result is a pattern that is indeed a specialisation of P.

<table>
<thead>
<tr>
<th>Theorem 4 (Specialisation by constraints)</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all patterns P and valid predicates c, P[c] ⇒ P.</td>
</tr>
<tr>
<td><strong>Proof.</strong> By Definition 4, we have that Spec(P[c]) ≡ (Spec(P) ∧ c) ⇒ Spec(P).</td>
</tr>
<tr>
<td>By Theorem 3, we have that Spec(P[c]) ⇒ P. □</td>
</tr>
</tbody>
</table>
Example 1 The pattern Composite₁ can be formally defined as follows.

\[ \text{Composite₁ = \text{Composite}[\text{Leaves} = \{\text{Leaf}\}]}. \]

By inference in the first order logic, we can simplify Composite₁ to the specification in Spec. 3. □

3.3 Lifting

Informally, the application of the lifting operation on a pattern \( P \) results in a pattern \( P' \) such that a design model \( m \) conforms to \( P' \) if it contains at least one set of components that matches pattern \( P \). We will write \( P \uparrow \) to denote the result of a lifting on \( P \). So \( P \uparrow \) is the pattern in which a model contains a variable number of instances of pattern \( P \). For example, Adapter \( \uparrow \) is the pattern that the design contains a number of Targets of adapted classes. For each of them, the system has an Adapter class and an Adaptee class configured as in the Adapter pattern. To indicate that the classes Adaptees and Adapters are varying according to the component Targets, we write Adapter \( \uparrow \) Target. In other words, the component Target in the lifted pattern plays a role similar to the primary key in a relational database. Formally, we define the lift operation as follows.

Definition 5 (Lift Operation)
Let \( P \) be a pattern and \( \text{Vars}(P) = \{v₁ : T₁, v₂ : T₂, \ldots, vₙ : Tₙ\} \), \( n > 0 \) and \( \text{Spec}(P) = p(v₁, v₂, \ldots, vₙ) \). Let \( V = \{v₁, v₂, \ldots, vₖ\}, 1 ≤ k < n, \) be a subset of the variables of the pattern. The lift of \( P \) with \( V \) as the key, written \( P \uparrow V \), is the pattern defined as follows.

\[
\text{Vars}(P \uparrow V) = \{vₛ₁ : \psi T₁, vₛ₂ : \psi T₂, \ldots, vₛₙ : \psi Tₙ\}
\]

\[
\text{Pred}(P \uparrow V) = ∀v₁ \in vₛ₁ \cdot ∀v₂ \in vₛ₂ \cdot ∀vₖ \in vₛₖ \cdot \exists vₖ₊₁ \in vₛₖ₊₁ \cdot \forall v_n \in vₛ_n \cdot p(v₁, v₂, vₖ, vₙ)
\]

\[
∧ (vₛ₁ ≠ \emptyset) ∧ \cdots ∧ (vₛₖ ≠ \emptyset)
\]

□

Example 2 For example, Adapter \( \uparrow \{\text{Target}\} \) is the pattern as follows.

\[
\text{Vars} (\text{Adapter} \uparrow \{\text{Target}\}) = \\
\{\text{Targets, Adapters, Adaptees} \subseteq \text{classes}\}
\]

\[
\text{Pred} (\text{Adapter} \uparrow \{\text{Target}\}) = \\
∀\text{Target} \in \text{Targets} \cdot ∃\text{Adapter} \in \text{Adapter} \cdot ∃\text{Adaptee} \in \text{Adaptees} \cdot \text{Spec(Adapter)}.
\]

□

Theorem 5 (Lifting Operation)
For all patterns \( P \) and models \( m \), we have \( m \models (P \uparrow V) \), where \( V \) is any non-empty subset of \( \text{Vars}(P) \).

Proof. (⇒) Let \( m \models P \). By definition 2, there is an assignment \( α \) such that \( \text{Eval}_α(P, m) = \text{true} \). Let \( α(vᵢ) = aᵢ \). We have that \( p(a₁, a₂, \ldots, aₙ) \) is true in model \( m \).

Define \( α'(vₛᵢ) = \{aᵢ\} \), for \( i = 1, 2, \ldots, n \). We have that, \( vₛᵢ ≠ \emptyset \) for all \( i = 1, 2, \ldots k \). It is also easy to see that the following predicate is also true under assignment \( α' \).

\[
∀v₁ \in vₛ₁ \cdot ∀v₂ \in vₛ₂ \cdot ∀vₖ \in vₛₖ \cdot
\exists vₖ₊₁ \in vₛₖ₊₁ \cdot ∀v_n \in vₛ_n \cdot p(v₁, v₂, vₖ, vₙ)
\]

Therefore, we have that \( m \models (P \uparrow V) \).

(⇐) Let \( m \models (P \uparrow V) \). By definition 2, there is an assignment \( α \) such that \( \text{Eval}_α(P \uparrow V, m) = \text{true} \). Let \( α(vₛᵢ) = Aᵢ \). We have that for all \( a₁ \in A₁, \ldots, aₖ \in Aₖ \) the following predicate is true in model \( m \).

\[
∀vₖ₊₁ \cdot ∀v_n \cdot p(a₁, \ldots, aₖ, vₖ₊₁, \ldots, vₙ)
\]

Let \( a_k₊₁, \ldots, a_n \) be the witness of \( vₖ₊₁, \ldots, vₙ \) in the above. Then, we have that \( p(a₁, a₂, \ldots, aₙ) \) is true in model \( m \). Define assignment \( α'(vᵢ) = aᵢ \), \( i = 1, 2, \ldots, n \). We have that \( \text{Eval}_α'(p(v₁, v₂, \ldots, vₙ), m) = \text{true} \). That is, \( m \models P \). □

3.4 Overlaps

Consider a situation in which a design model \( m \) conforms to two design patterns \( P \) and \( Q \) at the same time. Thus, by the definition of conformance and Theorem 1, we have that there is an assignment \( α₁ \) such that \( \text{Eval}_α₁(P, m) = \text{true} \), and at the same time there is an assignment \( α₂ \) such that \( \text{Eval}_α₂(Q, m) = \text{true} \).

An overlap between two assignments in the conformance means that there is at least one element in model \( m \) involved in both assignments \( α₁ \) and \( α₂ \). Such an involvement can be in one of the following three cases.

Case 1: One to one overlap. In this situation, there is one element in the model \( m \) assigned to a variable of pattern \( P \) by \( α₁ \) and at the same time the same element is also assigned to a variable in pattern \( Q \) by \( α₂ \). This element in the design model \( m \) plays two roles; one for each of the design patterns. In this case, we say the overlap is an one-to-one overlap. Let \( v \in \text{Vars}(P) \) and \( v' \in \text{Vars}(Q) \) be the variables that are assigned to the same element in model \( m \). Therefore, the model \( m \) does not only conform to \( P \) and \( Q \) (thus satisfy the condition \( P \land \text{Pred}(Q) \)), but also satisfies the condition that \( v = v' \). We will also name the new role that the overlap element plays in the composition, say \( v'' \). Therefore, we also have the equations that \( v'' = v \) and \( v'' = v' \) for the composition pattern. These equations are called the overlap constraints.

Case 2: One to many overlap. In this situation, there is an element in the model \( m \) assigned to a variable of pattern \( P \) by \( α₁ \), and at the same time there is a set \( E \) of elements such that \( E \) is assigned to a variable \( v'' \) in pattern \( Q \). In this case, we say the overlap is an one-to-many overlap. Assume
that the variable v is of type T, e.g., the type Class. Then, the variable v’ must be of type T, e.g., a subset of classes. Because variables v and v’ are of different types, we cannot simply write the overlap constraints as equations. Instead, we will lift the pattern P, thus variable v becomes variable vs, which is of type T. Assume that the new role that the overlap part plays is called v". The overlap constraints are v" ♦ vs and v" ⊆ v’.

Case 3: Many to many overlap. In this case, there is an element in the model m in a set E of elements assigned to variable v in pattern P by α1, and at the same time it is in another set E’ of elements that are assigned to variable v’ in pattern Q by α2. In this case, both variables must be of the same type of PT. In this case, we say that the overlap is a many-to-many overlap. Assume that the new role of the overlap part played in the composition pattern is called v".

Then, the overlap constraints are v" ⊆ v and v" ⊆ v’.

The following definition introduces a notation to represent various types of overlaps [3]. For the sake of simplicity, we assume that Vars(P) ∩ Vars(Q) = ∅. This can always be achieved by systematically renaming the variables in Vars(P) and their free occurrences in Pred(P).

**Definition 6 (Overlaps)**

Let P and Q be two patterns. We write vnew := (v : P # v’ : Q) to denote an overlap between the component v in pattern P with component v’ in pattern Q, and the overlap-part is named as vnew, where vnew ≠ Vars(P) ∪ Vars(Q). When there is no risk of confusion, we will omit the names of patterns and write simply vnew := (v # v’).

An overlap vnew := (v # v’) is called an one-to-one overlap and written vnew := v − v’, if both v and v’ are of type T. It is called a many-to-many overlap and written vnew := v >−< v’, if both v and v’ are of type PT.

In both one-to-one and many-to-many overlaps, we say that the variables v and v’ are equally ordered. The overlap is called a many-to-one overlap and written vnew := v − v’ or vnew := v −< v’, if both v and v’ are of type PT.

In this case, we say that v is higher ordered in the overlap and v’ lower ordered.

Let o be a set of overlaps as defined above between pattern P and Q. We write Vars(P) to denote the subset of variables in Vars(P) that occurred in o. We say that P is lower ordered, if there is a variable in Vars(P) that is lower ordered.

We can now define composition with an overlap.

**Definition 7 (Composition with an overlap)** Let o be a set of overlaps between pattern P and Q. The composition of P and Q with overlap o, written P ⊗ o Q, is a pattern defined as follows.

\[ \text{Vars}(P \otimes_o Q) = \text{Vars}(P') \cup \text{Vars}(Q') \cup \text{NewVars}(o), \]

Pred(P ⊗ o Q) = Pred(P’) ∧ Pred(Q’) ∧ OC(o).

Here U’ = U ⊃ Vars(U) if U is lower ordered in o; otherwise U’ = U, U = P, Q. NewVars(o) is the set of new variables given in overlap o. Predicate OC(o) is the conjunction of overlap constraints. Formally, let o = \{u_1 := v_1#v_1', \ldots, u_n := v_n#v_n' \}.

\[ \text{OC}(o) = \text{OC}(u_1 := v_1#v_1') \land \cdots \land \text{OC}(u_n := v_n#v_n'), \]

\[ \text{Pred}(P ⊗ o Q) = \text{Pred}(P') \land \text{Pred}(Q') \land \text{OC}(o), \]

\[ \forall \text{Pred, } \exists \text{ AdaptedLeaf } \in \text{Adapters } \land \text{Target } \in \text{Targets } \land \text{Adaptee } \in \text{Adaptees } \land \text{Pred}(\text{Adapter}) \]

In the above definition, variables vs and vs’ are the corresponding lifted variables of v and v’, respectively.

**Example 3** Now, we return to the motivating example of Composite1 and Adapter with the overlap that the Leaf component in Composite1 is the same as component Target in the Adapter pattern. We name this overlapped component AdaptedLeaf in the result pattern of the composition. This overlap can be specified as AdaptedLeaf := Leaf # Target. It is an one-to-one overlap. Let o1 be this overlap. Thus, the composition with o1 is the pattern Comp1Adapter = Composite1 ⊗ o1 Adapter. By Definition 7, we have that

\[ \text{Vars(Comp1Adapter)} = \{\text{Composite}, \text{Component}, \text{Leaf}, \text{Adapter}, \text{Adaptee}, \text{Target}, \text{AdaptedLeaf}\} \]

\[ \text{Pred(Comp1Adapter)} = \text{Pred(Composite)} \land \text{Pred(Adaptee)} \land \text{(AdaptedLeaf = Target)} \land \text{(AdaptedLeaf = Leaf)} \]

**Example 4** For the composition of Adapter with Composite (rather than Composite1), the overlap o2 is TLeafs := Leaf ≠ Target. This is a one-to-many overlap with Target as the lower ordered variable.

Let CompAdapter = Composite ⊗ o2 Adapter. By Definition 7, we have that

\[ \text{Vars(CompAdapter)} = \{\text{Composite}, \text{Component } \in \text{classes}, \text{Leafs}, \text{TLeafs}, \text{Adapters}, \text{Adaptees}, \text{Targets } \subseteq \text{classes}\} \]

\[ \text{Pred(CompAdapter)} = \text{(TLeafs } \subseteq \text{Leafs) } \land \text{Pred(Adaptee)} \land \text{Pred(Composite)} \land \text{(TLeafs } \not= \text{Target)} \land \forall \text{Adaptee } \in \text{Adaptees } \land \exists \text{Target } \in \text{Targets } \land \exists \text{Adaptee } \in \text{Adaptees } \land \text{Pred(Adaptee)} \]

\[ \square \]
As one would expect, we have the following theorem about composition as defined above.

**Theorem 6 (Composition of Patterns)**
Let $P$ and $Q$ be any given patterns and $o$ any overlap between $P$ and $Q$. For all models $m$, $m \models P \otimes_o Q$ implies that $m \models P$ and $m \models Q$.

**Proof.** Let $o$ be any given overlap between patterns $P$ and $Q$. If $P$ is lower ordered, by Definition 7, we have that $\text{Pred}(P \otimes_o Q) \Rightarrow \text{Pred}(P \uparrow \text{Vars}_P(o))$. Therefore, by Theorem 3, we have that for all models $m$, $m \models \text{Pred}(P \otimes_o Q)$ implies that $m \models \text{Pred}(P \uparrow \text{Vars}_P(o))$. By Theorem 5, we have that for all models $m$, $m \models \text{Pred}(P \uparrow \text{Vars}_P(o))$ if and only if $m \models \text{Pred}(P)$. Thus, we have that for all models $m$, $m \models \text{Pred}(P \otimes_o Q)$ implies that $m \models \text{Pred}(P)$.

Similarly, when $P$ is not lower ordered, by Definition 7, we have that $\text{Pred}(P \otimes_o Q) \Rightarrow \text{Pred}(P)$. By Theorem 3, we have that $m \models \text{Pred}(P \otimes_o Q)$ implies that $m \models \text{Pred}(P)$.

Therefore, in all cases, we have that for all models $m$, $m \models P \otimes_o Q$ implies that $m \models P$. The same proof applies to $Q$. Therefore the theorem is true. $\square$

4 Example and case study

This section illustrates how the notion of composition with overlaps can be apply to define compositions of design patterns and reports a case study on the expressiveness of the composition operators defined in this paper.

4.1 Example: The MVC Pattern

As a more complicated example of pattern composition, we now show that the MVC pattern can be formed by composing Composite, Strategy and Observer patterns. The specifications of Strategy and Observer are given below.

**Specification 4 (Strategy Pattern)**

**Components:**
- $\text{Context}, \text{Strategy} \in \text{classes}$,
- $\text{conInt} \in \text{Context}.\text{opers}$,
- $\text{algInt} \in \text{Strategy}.\text{opers}$

**Static Conditions**
- $\text{Context} \mapsto \text{Strategy}, \text{isAbstract}(\text{algInt})$

**Dynamic Conditions**
- $\text{callsHook}(\text{conInt}, \text{algInt})$

We first compose Composite to Strategy with a many-to-one overlap $o_3$.

$$o_3 = \{ \text{ContextLeafs} := \text{Leafs} \rightarrow \text{Context}, \text{OpContext} := \text{operation} \rightarrow \text{conInt} \}.$$ This lifts the variables Context and Strategy to sets and introduces a new component ContextLeafs. This yields the following pattern.

$$\text{Pred}(\text{Composite} \otimes_o \text{Strategy}) =$$

$$\text{Pred}(\text{Composite}) \land$$

$$\text{Pred}(\text{Strategy} \uparrow \{ \text{Context} \}) \land$$

$$\text{Pred}(\text{ContextLeafs} \subseteq \text{Contexts}) \land$$

$$\text{Pred}(\text{ContextLeafs} \not= \emptyset) \land$$

$$\text{Pred}(\text{ContextLeafs} \subseteq \text{Leafs})$$

However, because of the lifting of the variable Strategy to Strategies, the composition pattern has several Strategy classes, each the root of a separate hierarchy. This seems strange. In the Java API, the single class ActionListener is used. So we shall assume for simplicity that every class in the set Strategies is equal to a variable Strategy. This can be formally expressed as a constraint on the composition of the patterns by a predicate $\text{Strategies} = \{ \text{Strategy} \}$, thus we have

$$\text{Composite} \otimes_o \text{Strategy}[\text{Strategies} = \{ \text{Strategy} \}]$$

It is easy to prove that it is equivalent to the following pattern PVC by inference in the first order logic.

$$\text{Vars}(\text{PVC}) = \{ \text{Leafs} \subseteq \text{classes}, \text{Composite}, \text{Component} \in \text{classes} \}$$

$$\text{Pred}(\text{PVC}) = (\text{Strategy} \in \text{Leafs}) \land$$

$$\text{Pred}(\text{Composite}) \land \text{Pred}(\text{Strategy})$$

**Specification 5 (Observer Pattern)**

**Components:**
- $\text{Subject}, \text{Observer} \in \text{classes}$,
- $\text{update} \in \text{opers}(\text{Observer})$

**Static Conditions**
- $\text{Subject} \rightarrow* \text{Observer}$,
- $\neg\text{update}.\text{isLeaf}$,
- $\forall O \in \text{subs}(\text{Observer})$. $(\exists S \in \text{subs}(\text{Subject}) . O \rightarrow S)$

**Dynamic Conditions**
- $\exists ms \in \text{messages} . \neg ms.\text{sig}.\text{isQuery} \land$ toClass($ms$) $\in \text{subs}(\text{Subject}) \Rightarrow$
- $\exists m \in \text{messages} . m.\text{sig} = \text{notify} \land$
  fromLL($m$) = toLL($m$) = toLL($ms$)$\land$
- $\exists mu, mg . mu.\text{sig} = \text{update} \land \text{isQuery}(mg.\text{sig})$
  $\land$ toLL($mu$) = l $\land$ calls($m, mu$)$\land$
  calls($mu, mg$) $\land$ toLL($mg$) = toLL($ms$)$\land$

Then, we further compose this to Observer with the following overlap $o_4$. 

<table>
<thead>
<tr>
<th>Specification 5 (Observer Pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Components:</strong></td>
</tr>
<tr>
<td>$\text{Subject}, \text{Observer} \in \text{classes}$,</td>
</tr>
<tr>
<td>$\text{update} \in \text{opers}(\text{Observer})$</td>
</tr>
<tr>
<td><strong>Static Conditions</strong></td>
</tr>
<tr>
<td>$\exists ms \in \text{messages} . \neg ms.\text{sig}.\text{isQuery} \land$ toClass($ms$) $\in \text{subs}(\text{Subject}) \Rightarrow$</td>
</tr>
<tr>
<td>$\exists m \in \text{messages} . m.\text{sig} = \text{notify} \land$ fromLL($m$) = toLL($m$) = toLL($ms$)$\land$</td>
</tr>
<tr>
<td>$\exists mu, mg . mu.\text{sig} = \text{update} \land \text{isQuery}(mg.\text{sig}) \land$ toLL($mu$) = l $\land$ calls($m, mu$)$\land$</td>
</tr>
<tr>
<td>calls($mu, mg$) $\land$ toLL($mg$) = toLL($ms$) $\land$</td>
</tr>
</tbody>
</table>
suggests that the create and configure a particular Bridge [9] (p161). This the notion of overlap.

In [9], possible compositions of patterns were suggested in the Related Patterns section of each pattern. Such discussions were informal and little details were given. However, having formally defined the notion of compositions of patterns, we can now formalise all of these suggestions using the notion of overlap.

For example, it is stated that an Abstract Factory can create and configure a particular Bridge [9] (p161). This suggests that the Abstract and Implementor components in the Bridge pattern are the AbstractProducts in the AbstractFactory pattern. Thus, the overlap is

\[ o = \{ \text{AbstractProduct} \rightarrow \text{Abstractation, } \text{AbstractProduct} \rightarrow \text{Implementor} \} \]

By composing AbstractFactory pattern and Bridge pattern with the overlap \( o \), we can see that the Client’s dependences on Abstract and Implementor components in the Bridge pattern can be satisfied because of the constraints on AbstractProduct in the AbstractFactory pattern. All other conditions relate AbstractProducts to the other classes in the AbstractFactory pattern and so are not relevant.

It is also suggested that Iterator can be used to implement the recursion in Composite pattern [9]. The overlap is between the ConcreteAggregates in Aggregate pattern and the Component in Composite pattern. In our formal specification given in [4], ConcreteAggregate was implicitly referred to as \( \text{subs(Aggregate)} \). To enable composition with overlaps be defined, we first apply the specialisation operation to Iterator, i.e. define Iterator’ as follows.

\[ \text{Iterator’}[\text{ConcreteAggregates} = \text{subs(Aggregate)}] \]

Then, we compose Iterator’ to Composite with the overlap ConcreteAggregates \( \rightarrow \text{Component} \).

Table 2 summarises the overlaps between GoF patterns, where Command’ is defined in a similar way to Iterator’.

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>{AbsProducts \rightarrow \text{Abs, } Abstr, + Bridge \rightarrow \text{AbsProducts} \rightarrow \text{Impl} }</td>
</tr>
<tr>
<td>Composite + Decorator</td>
<td>{Component \rightarrow \text{Component, } Composite \rightarrow \text{Decorator, operation} \rightarrow \text{operation} }</td>
</tr>
<tr>
<td>Component + Flyweight</td>
<td>{Component \rightarrow \text{Flyweight} }</td>
</tr>
<tr>
<td>Iterator’ + Composite</td>
<td>{ConcreteAggregates \rightarrow \text{Component} }</td>
</tr>
<tr>
<td>Visitor + Composite</td>
<td>{Component \rightarrow \text{Element} }</td>
</tr>
<tr>
<td>Abstract Factory + Facade</td>
<td>{AbstractFactory \rightarrow \text{Facade} }</td>
</tr>
<tr>
<td>Facade + Singleton</td>
<td>{Facade \rightarrow \text{Singleton} }</td>
</tr>
<tr>
<td>State + Flyweight</td>
<td>{State \rightarrow \text{Flyweight} }</td>
</tr>
<tr>
<td>Strategy + Flyweight</td>
<td>{Strategy \rightarrow \text{Flyweight} }</td>
</tr>
<tr>
<td>Chain of Resp + Composite</td>
<td>{Chain of Resp \rightarrow \text{Composite} }</td>
</tr>
<tr>
<td>Command + Composite</td>
<td>{Command \rightarrow \text{Component} }</td>
</tr>
<tr>
<td>Memento + Command’</td>
<td>{Caretaker \rightarrow \text{ConcreteCommands} }</td>
</tr>
<tr>
<td>Prototype + Command</td>
<td>{Prototype \rightarrow \text{Command} }</td>
</tr>
<tr>
<td>Iterator + Factory Method</td>
<td>{Iterator \rightarrow \text{Product} }</td>
</tr>
<tr>
<td>Iterator’ + Memento</td>
<td>{ConcreteIterators \rightarrow \text{Caretaker} }</td>
</tr>
<tr>
<td>Mediator + Observer</td>
<td>{Colleague \rightarrow \text{Observer, Mediator} \rightarrow \text{Subject} }</td>
</tr>
<tr>
<td>State + Singleton</td>
<td>{State \rightarrow \text{Singleton} }</td>
</tr>
<tr>
<td>State + Flyweight</td>
<td>{State \rightarrow \text{Flyweight} }</td>
</tr>
<tr>
<td>Strategy + Flyweight</td>
<td>{Strategy \rightarrow \text{Flyweight} }</td>
</tr>
</tbody>
</table>

5 Conclusion

In this paper, we further developed our method for the formal specification of design patterns by adding a formal definition of the composition of design patterns. The properties of compositions are proved. A case study of the relationships between design patterns as suggested by GoF [9] is reported. As demonstrated in this paper, formal reasoning about design patterns and their compositions can be naturally supported by the formal deduction in first order logic, which is well understood, and well supported by software tools such as theorem provers.

There are a large number of papers published in the literature on formalisation of software design patterns. Existing
work can be classified into two categories. The first category proposes special-purpose formal languages or semiformal graphic modeling languages in order to define patterns rigorously [14, 6, 7]. The second category, to which our work belongs, simply employs or adapts existing formal or semi-formal languages [12, 18, 19, 15, 10, 13, 8, 20].

While each of these approaches are demonstrated with examples, it remains an open question whether they can be used to specify all design patterns. Our approach has been proven to be expressive by the successful specification of all 23 design patterns in the GoF catalog [2, 5]. In the existing work, few have attempted to facilitate formal reasoning about design patterns even though formalisation is perceived to have the potential to support formal reasoning. Generally speaking, for the works in the first category summarized above, to support formal reasoning of design patterns, a formal logic must be devised and its logic properties, such as semantic consistency must be proven and a special purpose formal reasoning tool must be developed. Thus, a great effort must be made before they can become practically useful. For works in the second category, those using the UML meta-modeling facility suffer from the ambiguity and informal definition of UML. Even the basic notion that a concrete design conforms to a pattern has to be defined with great length [8]. Few works on formal definition of pattern composition has been reported in the literature. One such work is by Taibi [18]. He discussed the composition of design patterns specified in the balanced pattern specification language BPSL. His definition did not handle the most complicated form of pattern composition like the many-to-many kind of overlaps, however.

In comparison with existing works, the formal specifications of design patterns in our approach are natural and easier to understand. Our approach is highly expressive. As demonstrated in this paper, reasoning about design patterns on various kinds of properties and relationship as well as their compositions can be naturally supported by the well-understood first order logic. Such reasoning can also be supported by software tools such as first order logic theorem provers. We are investigating the uses of theorem provers for automated reasoning about design patterns based on the theory developed in this paper.

References

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