Laws of Pattern Composition

Hong Zhu and Ian Bayley

Oxford Brookes University, Wheatley Campus,
Wheatley, Oxfordshire OX33 1HX, UK
Email: hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

Abstract. Design patterns are rarely used on their own. They are almost always to be found composed with each other in real applications. So it is crucial that we can reason about their compositions. In our previous work, we defined a set of operators on patterns so that pattern compositions can be represented as expressions on patterns. In this paper, we investigate the algebraic properties of these operators, prove a set of algebraic laws that they obey, and use the laws to show the equivalence of pattern compositions.

Keywords: Design patterns, Pattern composition, Formal methods, Algebraic laws, First order logic

1 Introduction

Design patterns are codified reusable solutions to recurring design problems [9, 1]. Many such patterns have been identified, documented, catalogued [6] and included in software tools [11, 16, 14]. Although each is specified separately, they are usually to be found composed with each other with overlaps except in trivial cases [17]. However, while the importance of pattern compositions has been widely recognised, it has not been studied intensively. This is perhaps partly because the patterns have been documented informally.

In the past few years, significant progress has been made by several researchers in the formalisation of design patterns. Several approaches have been advanced in the literature [15, 13, 19, 7, 10, 5]. In spite of the differences in the formalisms used by these approaches, the basic ideas underlying them are similar. In particular, a specification of a pattern usually consists of statements on the common structural features and, sometimes, behavioural features of its instances. The structural features of a pattern are typically specified by assertions on the existence of certain types of components in the pattern. The configuration of the elements is also described, in terms of the static relationship between them. The behavioural features are normally defined by assertions on the temporal orders of the messages exchanged between the components as manifested in the designs of systems. This formalisation lays a foundation for systematically and formally investigating the composition of design patterns.

However, very few authors have investigated composition formally. In [18], Taibi illustrated the concept of pattern composition in his framework of pattern formalisation with an example. In [3], we formally defined a universal pattern composition operator. In [22], we extended and revised the work, but took a radically different approach.

We replaced the single operator with a set of simpler operators that express composition when used together. A case study was also reported there to demonstrate the expressiveness of the operators. In this paper, we continue the work in this direction by investigating how to reason about pattern compositions, such as how to determine whether two pattern compositions are equivalent. We will prove a set of algebraic laws that these operators obey and demonstrate, with an example, how to prove equivalence of pattern compositions by equational reasoning.

The particular formalism that we will use in this paper to define operators and to prove their algebraic laws is that advanced in our previous work. This uses the first-order logic induced from the abstract syntax of UML defined in GEBNF [20, 21] to define both the structural and behavioural features of design patterns. In this way, we have already formally specified the 23 patterns in the classic Gang of Four (hereafter referred to as GoF) book [9], and we have specified variants too [2, 4, 5]. We have also constructed a prototype software tool to check whether a design represented in UML conforms to a pattern [23, 24]. It is worth noting that the definitions of the operations and the algebraic laws proved in this paper are independent of the formalism and thus can equally well be applied to others such as OCL [8], temporal logic [18], and so on, but the results may be less readable. In particular OCL would need to be applied at the meta-level to assert the existence of the required classes and methods.

The remainder of the paper is organised as follows. Section 2 reviews our approach to formalisation and lays the theoretical foundation for our proofs. Section 3 outlines the set of operations on design patterns. Section 4 presents the algebraic laws that they obey. Section 5 outlines the use of laws in equational reasoning about the equivalence of pattern compositions with an example. Section 6 concludes the paper with a discussion of related works and future work. For the sake of readability and space, the proofs of the algebraic laws are removed from the body of the paper and some are given in the appendix.

2 Background

This section briefly reviews our approach to the formal specification of design patterns. It is based on meta-modelling in the sense that each pattern is a subset of the design models having certain structural and behavioral features. Readers are referred to [2, 4, 23, 5] for details.

2.1 Meta-modelling in GEBNF

Our approach starts by defining the domain of all models with an abstract syntax written in the meta-notation Graphic Extension of BNF (GEBNF) [20]. GEBNF extends the traditional BNF notation with a ‘reference’ facility to define the graphical structure of diagrams. In addition, each syntactic element in the definition of a language construct is assigned an identifier (called a *field name*) so that a first-order language (FOL) can be induced from the abstract syntax definition [21].

For example, the following are some example syntax rules in GEBNF for the UML modelling language.

$$
\begin{array}{ll}
ClassDiag & ::= classes : Class^{+}, assocs, inherits, compag : Rel^{*} \\
Class & ::= name : String, [attrs : Property^{*}], [opers : Operation^{*}] \\
Rel & ::= [name : String], source : End, end : End \\
End & ::= node : \underline{Class}, [name : String], [mult : Multiplicity]
\end{array}
$$

The first line defines a class diagram as consisting of a non-empty set of classes and a collection of three relations on the set. Here $classes$, $assocs$, $inherits$ and $compag$ are field names. Each field name is a function. For example, $classes$ is a function from a $ClassDiag$ to the set of class nodes in the model. Functions $assocs$, $inherits$ and $compag$ are mappings from a class diagram to the sets of association, inheritance and composite/aggregate relations in the model. The non-terminal $Class$ in the definition of End is a reference occurrence. This means that the node at the end of a relation must be an existing class node in the diagram, not a newly introduced class node. The definitions of the class diagrams and sequence diagrams of UML in GEBNF can be found in [5]. Table 1 gives the functions used in this paper that are induced from these definitions as well as those that are based on them. A formal more detailed treatment of this can be found in [5].

Table 1. Some Functions Induced from GEBNF Syntax Definition of UML

ID	Domain	Function
Functions directly induced from GEBNF syntax definition of UML		
$classes$	Class diagram	The set of class nodes in the class diagram
$assocs$	Class diagram	The set of association relations in the class diagram
$inherits$	Class diagram	The set of inheritance relations in the class diagram
$compag$	Class diagram	The set of composite and aggregate relations in the class diagram
$name$	Class node	The name of the class
$attr$	Class node	The attributes contained in the class node
$opers$	Class node	The operations contained in the class node
sig	Message	The signature of the message
Functions defined based on induced functions		
$X \mathrel{-\!\!\triangleright}^{+} Y$	Class	Class X inherits class Y directly or indirectly
$X \longrightarrow^{+} Y$	Class	There is an association from class X to class Y directly or indirectly
$X \diamond\!\!\longrightarrow^{+} Y$	Class	There is an composite or aggregate relation from X to Y directly or indirectly
$isInterface(X)$	Class	Class X is an interface
$CDR(X)$	Class	No messages are send to a subclass of X from outside directly
$subs(X)$	Class	The set of class nodes that are subclasses of X
$calls(x, y)$	Operation	Operation x calls operation y
$isAbstract(op)$	Operation	Operation op is abstract
$fromClass(m)$	Message	The class of the object that message m is sent from
$toClass(m)$	Message	The class of the object that message m is sent to
$X \approx Y$	Operation	Operations X and Y share the same name

2.2 Formal specification of patterns

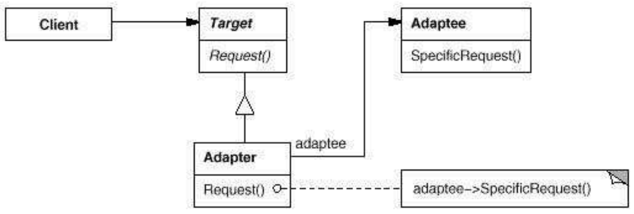
Given a formal definition of the domain of models, we can for each pattern, define a predicate in first-order logic to constrain the models such that each model that satisfies the predicates is an instance of the pattern.

Definition 1. *(Formal specification of DPs)*
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 static class diagrams and dynamic 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 $V = \{v_1 : T_1, \cdots, v_n : T_n\}$. The semantics of the specification is the closed formula in the following form.

$$\exists v_1 : T_1 \cdots \exists v_n : T_n \cdot (Pr_s \wedge Pr_d) \tag{1}$$

In the sequel, we write $Spec(P)$ to denote the predicate (1) above, $Vars(P)$ for the set of variables declared in V, and $Pred(P)$ for the predicate $Pr_s \wedge Pr_d$.

For example, Fig. 1 shows the specification of the Object Adapter design pattern. The class diagram from the GoF book has been included for the sake of readability.



Specification 1 *(Object Adapter Pattern)*
Components

1. $Target, Adapter, Adaptee \in classes$,
2. $requests \subseteq Target.opers$,
3. $specreqs \subseteq Adaptee.opers$

Static Conditions

1. $Adapter \longrightarrow\!\!\triangleright^{+} Target$, $Adapter \longrightarrow^{+} Adaptee$,
2. $CDR(Target)$

Dynamic Conditions

1. $\forall o \in requests \cdot \exists o' \in specreqs \cdot (calls(o, o'))$

Fig. 1. Specification of Object Adapter Pattern

Fig. 2 gives the specification of the $Composite$ pattern. Both patterns will be used throughout the paper.

2.3 Reasoning about patterns

We often want to show that a concrete design really conforms to a design pattern. This is a far from trivial task for some other formalisation approaches. For us though, the

Specification 2 *(Composite)*
Components

1. $Component, Composite \in classes$,
2. $Leaves \subseteq classes$,
3. $ops \subseteq Component.opers$

Static Conditions

1. $ops \neq \emptyset$
2. $\forall o \in ops.isAbstract(o)$,
3. $\forall l \in Leaves \cdot (l \mathrel{-\!\!\!\triangleright}^{+} Component \wedge \neg(l \diamond\!\!\longrightarrow^{+} Component))$
4. $isInterface(Component)$
5. $Composite \mathrel{-\!\!\!\triangleright}^{*} Component$
6. $Composite \diamond\!\!\longrightarrow^{+} Component$
7. $CDR(Component)$

Dynamic Conditions

1. *any call to $Composite$ causes follow-up calls*
$$\forall m \in messages \cdot \exists o \in ops \cdot (toClass(m) = Composite \wedge m.sig \approx o \Rightarrow$$
$$\exists m' \in messages \,.\, calls(m, m') \wedge m'.sig \approx m.sig)$$
2. *any call to a leaf does not*
$$\forall m \in messages \cdot \exists o \in ops \cdot toClass(m) \in Leaves \wedge m.sig \approx o \Rightarrow$$
$$\neg\exists m' \in messages \,.\, calls(m, m') \wedge m'.sig \approx m.sig)$$

Fig. 2. Specification of the Composite Pattern

use of predicate logic makes it easy and we formally define the conformance relation as follows.

Let m be a model and pr be a predicate. We write $m \models pr$ to denote that predicate pr is true in model m. Readers are referred to [21] for the formal definition of $m \models pr$.

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 $m \models Spec(P)$.* □

To prove such a conformance we just need to give an assignment α of variables in V to elements in m and evaluate $Pred(P)$ in the context of α. If the result is $true$, then the model satisfies the specification. This is formalised in the following lemma.

Lemma 1. *(Validity of conformance proofs)*
A model m conforms to a design pattern specified by predicate P if and only if there is an assignment α from $Vars(P)$ to the elements in m such that $Eva_\alpha(m, Pred(P)) = true$. □

A software tool has been developed that employs the first order logic theorem prover *SPASS*. With it, proofs of conformance can be performed automatically [23, 24].

Given a formal specification of a pattern P, we can infer the properties of any system that conforms to it. Using the inference rules of first-order logic, we can deduce that $Spec(P) \Rightarrow q$ where q is a formula denoting a property of the model. Intuitively, we expect that all models that conform to the specification should have this property and the following lemma formalises this intuition.

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

In other words, every logical consequence of a formal specification is a property of all the models that conform to the pattern specified.

There are several different kinds of relationships between patterns. Many of them can be defined as logical relations and proved in first-order logic. Specialisation and equivalence are examples of them.

Definition 3. *(Specialisation relation between patterns)*
Let P and Q be design patterns. Pattern P is a specialisation *of Q, written $P \preccurlyeq Q$, if for all models m, whenever m conforms to P, then, m also conforms to Q.* □

Definition 4. *(Equivalence relation between patterns)*
Let P and Q be design patterns. Pattern P is equivalent *to Q, written $P = Q$, if $P \preccurlyeq Q$ and $Q \preccurlyeq P$.* □

By Lemma 1, we can use inference in first-order logic to show specialisation.

Lemma 3. *(Validity of proofs of specialisation relation)*
Let P and Q be two design patterns. Then, we have that

1. *$P \preccurlyeq Q$, if $Spec(P) \Rightarrow Spec(Q)$, and*
2. *$P = Q$, if $Spec(P) \Leftrightarrow Spec(Q)$.* □

Furthermore, by Definition 1 and Lemma 3, we can prove specialisation and equivalence relations between patterns by inference on the predicate parts alone if their variable sets are equal.

Lemma 4. *(Validity of proofs of predicate relation)*
Let P and Q be two design patterns with $Vars(P) = Vars(Q)$. Then $P \preccurlyeq Q$ if $Pred(P) \Rightarrow Pred(Q)$, and $P = Q$ if $Pred(P) \Leftrightarrow Pred(Q)$. □

Specialisation is a pre-order with bottom $FALSE$ and top $TRUE$ defined as follows.

Definition 5. *(TRUE and FALSE patterns)*
Pattern $TRUE$ is the pattern such that for all models m, $m \models TRUE$. Pattern $FALSE$ is the pattern such that for no model m, $m \models FALSE$. □

In summary, therefore, and letting P, Q and R be any given patterns, we have the following.

$$P \preccurlyeq P \tag{2}$$

$$(P \preccurlyeq Q) \wedge (Q \preccurlyeq R) \Rightarrow (P \preccurlyeq R) \tag{3}$$

$$FALSE \preccurlyeq P \preccurlyeq TRUE \tag{4}$$

3 Operators on Design Patterns

In this section, we review the set of operators on patterns defined in [22]. The restriction operator was first introduced in [3], where it was called the *specialisation* operator.

Definition 6. *(Restriction operator)*
Let P be a given pattern and c be a predicate defined on the components of P. A restriction of P with constraint c, written $P\,[c]$, is the pattern obtained from P by imposing the predicate c as an additional condition of the pattern. Formally,

1. $Vars(P[c]) = Vars(P)$,
2. $Pred(P[c]) = (Pred(P) \wedge c)$. □

For example, the pattern $Composite_1$ is the variant of the $Composite$ pattern that has only one leaf:

$$Composite_1 = Composite[\#Leaves = 1].$$

Many more examples are given in the case studies reported in [22]. A frequently occuring use is in expressions of the form $P[u = v]$ for pattern P and variables u and v of the same type. This is the pattern obtained from P by unifying components u and v and making them the same element.

The restriction operator does not introduce any new components into the structure of a pattern, but the following operators do.

Definition 7. *(Superposition operator)*
Let P and Q be two patterns. Assume that the component variables of P and Q are disjoint, i.e. $Vars(P) \cap Vars(Q) = \emptyset$. The superposition *of P and Q, written $P * Q$, is defined as follows.*

1. $Vars(P * Q) = Vars(P) \cup Vars(Q)$;
2. $Pred(P * Q) = Pred(P) \wedge Pred(Q)$. □

Informally, $P * Q$ is the minimal pattern (i.e. that with the fewest components and weakest conditions) containing both P and Q without overlap. The definition has the requirement that component variables be disjoint, but we can always systematically rename the variables to make them disjoint and the notation with which we will do so is as follows. Let $x \in Vars(P)$ be a component of pattern P and $x' \notin Vars(P)$. The systematic renaming of x to x' is written as $P[x' := x]$. Obviously, for all models m, we have that $m \models P \Leftrightarrow m \models P[x' := x]$ because $Spec(P)$ is a closed formula. In the sequel, we assume that renaming is made implicitly before two patterns are superposed when there is a naming conflict between them.

Definition 8. *(Extension operator)*
Let P be a pattern, V be a set of variable declarations that are disjoint with P's component variables (i.e. $Vars(P) \cap V = \emptyset$), and c be a predicate with variables in $Vars(P) \cup V$. The extension of pattern P with components V and linkage condition c, written as $P\#(V \bullet c)$, is defined as follows.

1. $Vars(P\#(V \bullet c)) = Vars(P) \cup V$;

2. $Pred(P\#(V \bullet c)) = Pred(P) \wedge c$. $\square$

For any predicate p, let $p[x \backslash e]$ denote the result of replacing all free occurrences of x in p with expression e.

Now we can define the flatten operator as follows.

Definition 9. *(Flatten Operator)*
Let P be a pattern, $xs : \mathbb{P}(T)$ be a variable in $Vars(P)$ and $x : T$ be a variable not in $Vars(P)$. Then the flattening of P on variable x, written $P \Downarrow xs \backslash x$, is defined as follows.

1. $Vars(P \Downarrow xs \backslash x) = (Vars(P) - \{xs : \mathbb{P}(T)\}) \cup \{x : T\}$,
2. $Pred(P \Downarrow xs \backslash x) = Pred(P)[xs \backslash \{x\}]$. $\square$

Note that $\mathbb{P}(T)$ is the power set of T, and thus, $xs : \mathbb{P}(T)$ means that variable xs is a set of elements of type T. For example, $Leaves \subseteq classes$ in the specification of the $Composite$ pattern is the same as $Leaves : \mathbb{P}(classes)$. Applying the flatten operator on $Leaves$, the $Composite_1$ pattern can be equivalently expressed as follows.

$$Composite \Downarrow Leaves \backslash Leaf$$

As an immediate consequence of this definition, we have the following property. For $x_1 \neq x_2$ and $x'_1 \neq x'_2$,

$$(P \Downarrow x_1 \backslash x'_1) \Downarrow x_2 \backslash x'_2 = (P \Downarrow x_2 \backslash x'_2) \Downarrow x_1 \backslash x'_1. \quad (5)$$

Therefore, we can overload the $\Downarrow$ operator to a set of component variables. Let X be a subset of P's component variables all of power set type, i.e. $X = \{x_1 : \mathbb{P}(T_1), \cdots, x_n : \mathbb{P}(T_n)\} \subseteq Vars(P), n \geq 1$ and $X' = \{x'_1 : T_1, \cdots, x'_n : T_n\}$ such that $X' \cap Vars(P) = \emptyset$. Then we write $P \Downarrow X \backslash X'$ to denote $P \Downarrow x_1 \backslash x'_1 \Downarrow \cdots \Downarrow x_n \backslash x'_n$.

Note that our pattern specifications are closed formulae, containing no free variables. Although the names given to component variables greatly improve readability, they have no effect on semantics so, in the sequel, we will often omit new variable names and write simply $P \Downarrow x$ to represent $P \Downarrow x \backslash x'$. Also, we will use plural forms for the names of lifted variables, e.g. xs for the lifted form of x, and similarly for sets of variables, e.g. XS for the lifted form of X.

Definition 10. *(Generalisation operator)*
Let P be a pattern, $x : T$ be a variable in $Vars(P)$ and $xs : \mathbb{P}(T)$ be a variable not in $Vars(P)$. Then the generalisation *of P on variable x, written $P \Uparrow x \backslash xs$, is defined as follows.*

1. $Vars(P \Uparrow x \backslash xs) = (Vars(P) - \{x : T\}) \cup \{xs : \mathbb{P}(T)\}$,
2. $Pred(P \Uparrow x \backslash xs) = \forall x \in xs \cdot Pred(P)$. $\square$

We will use the same syntactic sugar for $\Uparrow$ as we do for $\Downarrow$. In other words, we will often omit the new variable name and write $P \Uparrow x$, and thanks to an analogue of Equation 5, we can and will promote the operator $\Uparrow$ to sets.

For example, by applying the generalisation operator to $Composite_1$ on the component $Leaf$, we can obtain the pattern $Composite$. Formally,

$$Composite = Composite_1 \Uparrow Leaf \backslash Leaves.$$

The lift operator was first introduced in our previous work [3], but in [22] it is revised so that it only allows lifting class components. Let $CVars(P)$ be the set of variables of patterns P that range over classes, and $OPred(P)$ be the predicate obtained from $Pred(P)$ by the existentially quantifying at the outermost the remaining variables not in $CVars(P)$, i.e. those in $Vars(P) - CVars(P)$, which are the declarations of the operations. Then, we can define lifting as follows.

Definition 11. *(Lift Operator)*
Let P be a pattern and $CVars(P) = \{x_1 : T_1, \cdots, x_n : T_n\}$, $n > 0$. Let $X = \{x_1, \cdots, x_k\}$, $1 \leq k < n$, be a subset of the variables in the pattern. The lifting of P with X as the key, written $P \uparrow X$, is the pattern defined as follows.

1. $Vars(P \uparrow X) = \{xs_1 : \mathbb{P}T_1, \cdots, xs_n : \mathbb{P}T_n\}$,
2. $Pred(P \uparrow X) = \forall x_1 \in xs_1 \cdots \forall x_k \in xs_k \cdot \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot OPred(P)$. □

When the key set is singleton, we omit the set brackets for simplicity, so we write $P \uparrow x$ instead of $P \uparrow \{x\}$.

For example, $Adapter \uparrow Target$ is the following pattern.

$$\begin{aligned} Vars(Adapter \uparrow Target) &= \{Targets, Adapters, Adaptees \subseteq classes\} \\ Pred(Adapter \uparrow Target) &= \forall Target \in Targets \cdot \exists Adapter \in Adapter \cdot \\ &\qquad \exists Adaptee \in Adaptees \cdot OPred(Adapter). \end{aligned}$$

Fig. 3 spells out the components and predicates of the pattern.

Specification 3 *(Lifted Object Adapters Pattern)*
Components

1. $Targets, Adapters, Adaptees \subseteq classes$,

Conditions

1. $\forall Adaptee \in Adaptees \cdot \exists specreqs \in Adaptee.opers$,
2. $\forall Target \in Targets \cdot \exists requests \in Target.opers$,
3. $\forall Target \in Targets \cdot CDR(Target)$,
4. $\forall Target \in Targets \cdot \exists Adapter \in Adapters, Adaptee \in Adaptees \cdot$
   (a) $Adapter \longrightarrow\!\triangleright Target$,
   (b) $Adapter \longrightarrow Adaptee$,
   (c) $\forall o \in Target.requests \cdot \exists o' \in Adaptee.specreqs \cdot (calls(o, o')))$

Fig. 3. Specification of Lifted Object Adapter Pattern

Informally, lifting a pattern P results in a pattern P' that contains a number of instances of P. For example, $Adapter \uparrow Target$ is the pattern that contains a number of $Target$s of adapted classes. Each of these has a dependent $Adapter$ and $Adaptee$ class configured as in the original $Adapter$ pattern. In other words, the component $Target$ in the lifted pattern plays a role similar to the *primary key* in a relational database.

4 Algebraic Laws of the Operations

This section studies the algebraic laws that the operators obey. For the sake of space, we only give some proofs in the appendix.

4.1 Laws of restriction

Let $vars(p)$ denote the set of free variables in a predicate p. For all predicates c, c_1, c_2 such that $vars(c), vars(c_1)$ and $vars(c_2) \subseteq Vars(P)$, the following equalities hold.

$$P[c_1] \preccurlyeq P[c_2],\ if\ c_1 \Rightarrow c_2 \tag{6}$$
$$P[c] \preccurlyeq P[true] \tag{7}$$
$$P[c][c] = P[c] \tag{8}$$
$$P[c_1][c_2] = P[c_2][c_1] \tag{9}$$
$$P[c_1][c_2] = P[c_1 \wedge c_2] \tag{10}$$
$$P[true] = P \tag{11}$$
$$P[false] = FALSE \tag{12}$$

4.2 Laws of superposition

For all patterns P and Q, we have the following equations.

$$P * Q \preccurlyeq P \tag{13}$$

$$Q \preccurlyeq P \Rightarrow P * Q = Q \tag{14}$$

From this and reflexivity of $\preccurlyeq$, it follows that superposition is idempotent.

$$P * P = P \tag{15}$$

It also follows from (14) that $TRUE$ is the unit of superposition since it is the top in $\preccurlyeq$. Similarly, $FALSE$ is the zero of superposition since it is the bottom in $\preccurlyeq$.

$$P * TRUE = TRUE * P = P \tag{16}$$
$$P * FALSE = FALSE * P = FALSE \tag{17}$$

Superposition is also commutative and associative.

$$P * Q = Q * P \tag{18}$$
$$(P * Q) * R = P * (Q * R) \tag{19}$$

4.3 Laws of extension

The extension operation has the following properties.

Let U be any set of component variables that is disjoint to $Vars(P)$, and c_1, c_2 be any given predicates such that $vars(c_i) \subseteq Vars(P) \cup U$, $i = 1, 2$. We have the following inequalities.

$$P\#(U \bullet c_1) \preccurlyeq P\#(U \bullet c_2),\ if\ c_1 \Rightarrow c_2 \tag{20}$$

$$P\#(U \bullet c_1) \preccurlyeq P \tag{21}$$

Let U and V be any sets of component variables that are disjoint to $Vars(P)$ and to each other, c_1 and c_2 be any given predicates such that $vars(c_1) \subseteq Vars(P) \cup U$ and $vars(c_2) \subseteq Vars(P) \cup V$. We have equalities.

$$P\#(U \bullet c_1)\#(V \bullet c_2) = P\#(U \cup V \bullet c_1 \wedge c_2) \tag{22}$$

$$P\#(U \bullet c_1)\#(V \bullet c_2) = P\#(V \bullet c_2)\#(U \bullet c_1) \tag{23}$$

4.4 Laws of flattening and generalisation

Let $X, Y \subseteq Vars(P)$ and $X \cap Y = \emptyset$.

$$(P \Downarrow X) \Downarrow Y = P \Downarrow (X \cup Y) \tag{24}$$

$$(P \Uparrow X) \Uparrow Y = P \Uparrow (X \cup Y) \tag{25}$$

4.5 Laws connecting several operators

For all predicates c such that $vars(c) \subseteq Vars(P)$, we have that

$$P[c] * Q = (P * Q)[c]. \tag{26}$$

For all $X \subseteq Vars(P)$, we have that

$$(P \Uparrow X) * Q = (P * Q) \Uparrow X, \tag{27}$$

$$(P \Downarrow X) * Q = (P * Q) \Downarrow X. \tag{28}$$

Let $X \subseteq Vars(P) \cup Vars(Q)$. From (24), (27) and (28), we can prove that

$$(P * Q) \Uparrow X = (P \Uparrow X_P) * (Q \Uparrow X_Q), \tag{29}$$

$$(P * Q) \Downarrow X = (P \Downarrow X_P) * (Q \Downarrow X_Q), \tag{30}$$

where $X_P = X \cap Vars(P)$, $X_Q = X \cap Vars(Q)$.

For all sets of variables X such that $X \cap vars(P) = \emptyset$ and all predicates c such that $Vars(c) \subseteq (Vars(P) \cup X)$, we have that

$$P\#(X \bullet c) = P\#(X \bullet True)[c]. \tag{31}$$

$$P\#(X \bullet c) = P[\exists X \cdot c], \tag{32}$$

where $\exists X \cdot c = \exists x_1 : T_1 \cdots \exists x_k : T_k \cdot c$, if $X = \{x_1 : T_1, \cdots, x_k : T_k\}$.

For all $x \in Vars(P)$ such that $x : \mathbb{P}(T)$, we have that

$$P \Downarrow (x \backslash x') = P\#(\{x' : T\} \bullet (x = \{x'\}). \tag{33}$$

For all $X \subseteq Vars(P)$ and $X' \cap Vars(P) = \emptyset$, we have that

$$P \Uparrow X \backslash X' = (P \uparrow X \backslash X') \Downarrow (V - X'), \tag{34}$$

$$(P \Uparrow X \backslash X') \Downarrow (X' \backslash X) = P. \tag{35}$$

where $V = Vars(P \uparrow X)$.

From (34) and (35), we can prove that for all $x \in Vars(P)$,

$$(P \uparrow x) \Downarrow V = P, \tag{36}$$

where $V = Vars(P \uparrow x)$.

Let $X \subseteq Vars(P)$, we have that

$$(P \uparrow X) * Q = ((P * Q) \uparrow X) \Downarrow Vars(Q). \tag{37}$$

Let c be a predicate that $vars(c) \subseteq X \cup V \subseteq Vars(P)$, we have that

$$((P[c] \uparrow X) \Downarrow VS) = ((P \uparrow X) \Downarrow VS)[c'], \tag{38}$$

where $c' = \forall x_1 : xs_1, \cdots, \forall x_k : xs_k \cdot c$, $\{x_1, \cdots, x_k\} = vars(c) \cap X$.

5 Examples

In this section, we demonstrate the uses of the laws to prove the equivalence of pattern compositions.

We first consider the composition of $Composite$ and $Adapter$ in such a way that one of the $Leaves$ in the $Composite$ pattern is the $Target$ in the $Adapter$ pattern. This leaf is renamed as the $AdaptedLeaf$. The definition for the composition using the operators is as follows:

$$\begin{aligned} &OneAdaptedLeaf \triangleq \\ &\quad (Adapter * Composite)[Target \in Leaves][AdaptedLeaf := Target] \end{aligned}$$

Then, we lifted the adapted leaf to enable several of these $Leaves$ to be adapted. That is, we lift the $OneAdaptedLeaf$ pattern with $AdaptedLeaf$ as the key and then flatten those components in the composite part of the pattern (i.e. the components in the $Composite$ pattern remain unchanged). Formally, this is defined as follows.

$$\begin{aligned} &(OneAdaptedLeaf \uparrow (AdaptedLeaf \backslash AdaptedLeaves)) \\ &\quad \Downarrow \{Composites, Components, Leaveses\} \end{aligned} \tag{39}$$

By the definitions of the operators, we derive the predicates of the pattern in Specification 4 after some simplification in the first order logic.

Specification 4 *(ManyAdaptedLeaves)*
Components

1. $Component, Composite \in classes$,
2. $Leaves, AdaptedLeaves, Adapters, Adaptees \subseteq classes$,
3. $ops \subseteq Component.opers$

Static Conditions

1. $ops \neq \emptyset$
2. $\forall o \in ops.isAbstract(o)$,
3. $\forall l \in Leaves.(l \mathrel{-\!\!\triangleright}^{+} Component \wedge \neg(l \diamond\!\!\longrightarrow^{+} Component))$
4. $\forall l \in AdaptedLeaves.(l \mathrel{-\!\!\triangleright}^{+} Component \wedge \neg(l \diamond\!\!\longrightarrow^{+} Component))$
5. $isInterface(Component)$,
6. $Composite \mathrel{-\!\!\triangleright}^{+} Component$
7. $Composite \diamond\!\!\longrightarrow^{*} Component$
8. $CDR(Component)$
9. $\forall Adaptee \in Adaptees \cdot (\exists specreqs \in Adaptee.opers$,
10. $\forall AdLeaf \in AdaptedLeaves \cdot \exists requests \in AdLeaf.opers$,

Dynamic Conditions

1. *any call to $Composite$ causes follow-up calls*
$$\forall m \in messages \cdot \exists o \in ops \cdot (toClass(m) = Composite \wedge m.sig \approx o \Rightarrow$$
$$\exists m' \in messages\ .\ calls(m, m') \wedge m'.sig \approx m.sig)$$
2. *any call to a leaf or an adapted leaf does not*
$$\forall m \in messages \cdot (\exists o \in ops \cdot (toClass(m) \in Leaves \cup AdaptedLeaves \wedge$$
$$m.sig \approx o) \Rightarrow \neg\exists m' \in messages\ .\ calls(m, m') \wedge m'.sig \approx m.sig))$$
3. $\forall AdLeaf \in AdaptedLeaves \cdot \exists Adapter \in Adapters, Adaptee \in Adaptees \cdot$
   (a) $Adapter \mathrel{-\!\!\triangleright} AdLeaf$,
   (b) $Adapter \longrightarrow Adaptee$,
   (c) $\forall o \in AdLeaf.requests \cdot \exists o' \in Adaptee.specreqs \cdot (calls(o, o')))$

An alternative way of expressing the composition is first to lift the $Adapter$ with $target$ as the key and then to superposition it to the $Composite$ patterns so that many leaves can be adapted. Formally,

$$\begin{aligned}&ManyAdaptedLeaves \triangleq\\ &\quad(((Adapter \uparrow (Target \backslash Targets)) * Composite)[Targets \subseteq Leaves]\\ &\qquad[AdaptedLeaves := Targets]\end{aligned}$$

We now apply the algebraic laws to prove that expression Equ. (39) is equivalent to the definition of $ManyAdaptedLeaves$.

First, by (37), we can rewrite $ManyAdaptedLeaves$ to the following expression, where $V_C = \{Composites, Components, Leaveses\}$.

$$\begin{aligned}&((Adapter * Composite) \uparrow (Target \backslash Targets) \Downarrow V_C\\ &\quad[Targets \subseteq Leaves]\,[Adaptedleaves := Targets]\end{aligned} \tag{40}$$

Because $Leavses$ is in V_C and $Targets \subseteq Leaves$ is equivalent to

$$\forall Target \in Targets \cdot (Target \in Leaves),$$

by (38), we have that

$$\begin{aligned}&((Adapter * Composite) \uparrow (Target \backslash Targets) \Downarrow V_C)\,[Targets \subseteq Leaves] \quad (41)\\ &= ((Adapter * Composite)[Target \in Leaves]) \uparrow (Target \backslash Targets) \Downarrow V_C\end{aligned}$$

Now, renaming $Target$ to $AdaptedLeaf$ and $Targets$ to $AdaptedLeaves$ in expression on the right-hand-side of (41), we have the following.

$$\begin{aligned}&((Adapter * Composite)[Target \in Leaves][AdaptedLeaf := Target])\\ &\qquad \uparrow (AdaptedLeaf \backslash AdaptedLeaves) \Downarrow V_C \qquad (42)\end{aligned}$$

By substituting the definition of $OneAdaptedLeaf$ into Equ. (42), we obtain (39).

6 Conclusion

In this paper, we proved a set of algebraic laws that the operators on design patterns obey and we demonstrated their use in proving the equivalence of pattern compositions. These operators and algebraic laws form a formal calculus of design patterns that enable us to reasoning about pattern compositions. Although the calculus is developed in our own formalisation framework, we believe that they can be easily adapted to others, such as that of Eden's approach, which also uses first-order logic but no specification of behavioural features [10], that of Taibi's approach, which is a mixture of first-order logic and temporal logic [19], and that of [12], etc. as well as the approaches based on graphic meta-modelling languages, such as RBML [8] and DPML [14]. However, the definitions of the operators and proofs of the laws are more concise and readable in our formalism.

For future work, we are investigating the uses of theorem provers for automated reasoning about the compositions of design patterns based on the theory developed in this paper. We are also investigating the completeness of the algebraic laws.

References

1. Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall, 2nd edn. (June 2003)
2. Bayley, I., Zhu, H.: Formalising design patterns in predicate logic. In: Proc. of SEFM'07. pp. 25–36. IEEE Computer Society, (Sept 2007)
3. Bayley, I., Zhu, H.: On the composition of design patterns. In: Proc. of QSIC 2008. pp. 27–36. IEEE Computer Society, (Aug 2008)
4. Bayley, I., Zhu, H.: Specifying behavioural features of design patterns in first order logic. In: Proc. of COMPSAC'08. pp. 203–210. IEEE Computer Society, (Aug 2008)
5. Bayley, I., Zhu, H.: Formal specification of the variants and behavioural features of design patterns. Journal of Systems and Software 83(2), 209–221 (Feb 2010)

6. Buschmann, F., Henney, K., Schmidt, D.C.: Past, present, and future trends in software patterns. IEEE Software 24(4), 31–37 (2007)
7. Eden, A.H.: Formal specification of object-oriented design. In: International Conference on Multidisciplinary Design in Engineering, Montreal, Canada (November 2001)
8. France, R.B., Kim, D.K., Ghosh, S., Song, E.: A UML-based pattern specification technique. IEEE Trans. Softw. Eng. 30(3), 193–206 (2004)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
10. Gasparis, E., Nicholson, J., Eden, A.H.: LePUS3: An object-oriented design description language. In: Proc. of Diagrams'08. Lecture Notes in Computer Science, vol. 5223, pp. 364–367. Springer (September 2008)
11. Hou, D., Hoover, H.J.: Using SCL to specify and check design intent in source code. IEEE Trans. Softw. Eng. 32(6), 404–423 (June 2006)
12. Lano, K., Bicarregui, J.C., Goldsack, S.: Formalising design patterns. In: BCS-FACS Northern Formal Methods Workshop, Ilkley, UK (September 1996)
13. Lauder, A., Kent, S.: Precise visual specification of design patterns. In: Proc. of ECOOP'98 Lecture Notes in Computer Science Vol. 1445. pp. 114–134, Springer (1998)
14. Mapelsden, D., Hosking, J., Grundy, J.: Design pattern modelling and instantiation using DPML. In: Proc. of CRPIT '02. pp. 3–11. Australian Computer Society, Inc. (2002)
15. Mikkonen, T.: Formalizing design patterns. In: Proc. of ICSE'98. pp. 115–124. IEEE CS (April 1998)
16. Nija Shi, N., Olsson, R.: Reverse engineering of design patterns from Java source code. In: Proc. of ASE'06. pp. 123–134 (September 2006)
17. Riehle, D.: Composite design patterns. In: Proc. of OOPSLA'97. pp. 218–228 (1997)
18. Taibi, T.: Formalising design patterns composition. Software, IEE Proceedings 153(3), 126–153 (June 2006)
19. Taibi, T., Check, D., Ngo, L.: Formal specification of design patterns-a balanced approach. Journal of Object Technology 2(4) (July-August 2003)
20. Zhu, H., Shan, L.: Well-formedness, consistency and completeness of graphic models. In: Proc. of UKSIM'06. pp. 47–53 (April 2006)
21. Zhu, H.: On the theoretical foundation of meta-modelling in graphically extended BNF and first order logic. In: Proc. of TASE 2010. IEEE CS Press, Taipei, Taiwan (August 2010).
22. Zhu, H., Bayley, I.: A formal language of pattern composition. In: Proc. of PATTERNS 2010. Lisbon, Portugal (November 2010) (In Press)
23. Zhu, H., Bayley, I., Shan, L., Amphlett, R.: Tool support for design pattern recognition at model level. In: Proc. of COMPSAC'09. pp. 228–233. IEEE CS (July 2009)
24. Zhu, H., Shan, L., Bayley, I., Amphlett, R.: A formal descriptive semantics of UML and its applications. In: Lano, K. (ed.) UML 2 Semantics and Applications. John Wiley & Sons, Inc. (November 2009)

Appendix. Proofs of the Algebraic Laws

In this appendix, we give some proofs of the algebraic laws.
Proof of Laws of Restriction:

For Law (6), let P be any given pattern, and c_1, c_2 be any predicates such that $vars(c_i) \subseteq Vars(P)$, $i = 1, 2$. By Definition 6, we have $Vars(P[c_i]) = Vars(P)$, and $Pred(P[c_i]) = Pred(P) \wedge c_i$, for $i = 1, 2$. Assume that $c_1 \Rightarrow c_2$. Then, we have that $Pred(P[c_1]) = Pred(P) \wedge c_1 \Rightarrow Pred(P) \wedge c_2 \equiv Pred(P[c_2])$. So by Lemma 4, we have that $P[c_1] \preccurlyeq P[c_2]$.

Similarly, we can prove that $Pred(P[true]) \equiv Pred(P)$ and $Pred(P[c_1][c_2]) \equiv Pred(P[c_1 \wedge c_2]$, thus, Law (10) and (11) are true by Lemma 4.

Law (7) is the special case of (6) where c_2 is $true$. For (8), we have that $c \wedge c \equiv c$. Thus, it follows from (10).

Law (12) holds because $Pred(P[false])$ cannot be satisfied by any models. $\square$

For the majority of laws, the variable sets on the two sides of the law can be proven to be equal. Therefore, by Lemma 4, the proof of the law reduces to the proof of the equivalence or implication between the predicates. However, for some laws, these variable sets are not equal. In such cases, we use Lemma 3. The following is an example of such a proof.

Proof of Law (13):

Let P and Q be patterns with

$$Vars(P) = \{x_1, \ldots, x_m\}, Vars(Q) = \{y_1, \ldots, y_n\}.$$

Assume that

$$Vars(P) \cap Vars(Q) = \emptyset. \tag{43}$$

$$\begin{array}{ll}
Spec(P * Q) & \\
= \exists x_1, \ldots x_m, y_1 \ldots y_n \cdot Pred(P) \wedge Pred(Q), & < Def.\ 1 > \\
\equiv \exists x_1, \ldots, x_m \cdot Pred(P) \wedge \exists y_1 \ldots y_n \cdot Pred(Q), & < (43) > \\
\Rightarrow \exists x_1, \ldots, x_m \cdot Pred(P), & < FOL > \\
= Spec(P), & < Def.\ 1 >
\end{array}$$

Thus, by Lemma 3, we have that $(P * Q) \preccurlyeq P$. $\square$

The following of the proof of Law (38), which involves three operators.

Proof of Law (38):

First, we prove that the variable sets on the two sides of the equation are equal.

Let $Y = Vars(P) - (X \cup V)$. Then, we have that $Vars(P) = X \cup Y \cup V$. By definition of the operators, it is easy to see that

$$Vars(lhs) = (((XS \cup YS \cup VS) - VS) \cup V) = (XS \cup YS \cup V) = Vars(rhs).$$

Thus, we only need to prove the predicates of the two sides are equivalent. Let $X = \{x_1, \cdots, x_k\}$, $Y = \{y_1, \cdots, y_n\}$ and $V = \{v_1, \cdots, v_m\}$.

By the definitions of the operators, we have that $Pred(lhs)$ is

$$\begin{array}{l}
\forall x_1 \in xs_1 \ldots x_k \in xs_k \cdot \exists y_1 \in ys_1 \ldots y_n \in ys_n \cdot \\
\quad \exists v_1 \in vs_1 \ldots v_m \in vs_m \cdot (Pred(P) \wedge c)[vs_1 \backslash \{v_1\}] \ldots [vs_m \backslash \{v_m\}] \\
\equiv \forall x_1 \in xs_1 \ldots x_k \in xs_k \cdot \exists y_1 \in ys_1 \ldots y_n \in ys_n \cdot \\
\quad \exists v_1 \in vs_1 \ldots v_m \in vs_m \cdot (Pred(P)[vs_1 \backslash \{v_1\}] \ldots [vs_m \backslash \{v_m\}] \\
\quad \wedge c[vs_1 \backslash \{v_1\}] \ldots [vs_m \backslash \{v_m\}]) \\
\equiv \forall x_1 \in xs_1 \ldots x_k \in xs_k \cdot \exists y_1 \in ys_1 \ldots y_n \in ys_n \cdot (Pred(P) \wedge c)
\end{array}$$

Because $vars(c) \cap Y = \emptyset$, the above is equivalent to the following.

$$\begin{array}{l}
\forall x_1 \in xs_1 \ldots x_k \in xs_k \cdot \exists y_1 \in ys_1 \ldots y_n \in ys_n \cdot Pred(P) \\
\quad \wedge \forall x_1 \in xs_1 \ldots x_k \in xs_k \cdot c
\end{array}$$

This is $Pred(rhs)$. By Lemma 4, the law holds. $\square$