On the Composability of Design Patterns

Hong Zhu, *Senior Member, IEEE* and Ian Bayley

Abstract—In real applications, design patterns are almost always to be found composed with each other. It is crucial that these compositions be validated. This paper examines the notion of validity, and develops a formal method for proving or disproving it, in a context where composition is performed with formally defined operators on formally specified patterns. In particular, for validity, we require that pattern compositions preserve the features, semantics and soundness of the composed patterns. The application of the theory is demonstrated by a formal analysis of overlap-based pattern compositions and a case study of a real pattern-oriented software design.

Index Terms—Design patterns, pattern composition, composibility, feature preservation, semantics preservation, soundness preservation, formal methods

1 Motivation

Design patterns encapsulate knowledge of reusable solutions to recurring design problems [1]. Since Gamma et al. published a catalogue of 23 basic OO design patterns [2], a large number of patterns in various specific design areas have been identified and documented [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. Many software tools have been developed, often as IDE plug-ins, to apply design patterns, or to recognise the correct uses of patterns at code level [16], [17], [18], [19], [20], [21] and at model level [22], [23], [24], [25], [26]. They are widely used in practice in almost all software development [27]. A pattern-oriented software design methodology is emerging [28], [29].

Empirical studies show that design patterns are often used wrongly, with a negative impact on software quality [27], [30], [31], though the exact meaning of appropriate application is still an open question. For example, Fig. 1 shows in diagrams c) to f), four different compositions of the Gamma et al.'s patterns [2] Composite and Adapter, with the latter indicated by shading. Are these valid and is there a way to prove that they are?

In this paper, we take a formal approach to the problem by proposing a mathematical definition of the notion of valid composition and instantiation of design patterns, and developing a formal theory that allows us to formally prove or disprove that a use of a design pattern is sound and valid. The applicability of the theory is demonstrated by applying it to the analysis of overlap-based pattern compositions as well as a case study with a real example of pattern-oriented design. It is based on our previous work on an algebra of design patterns [32] as well as on the work of many others on formalisation of design patterns [33], [34], [35], [36], [37], [38], [39], [40], [41].

• *The authors are with the Oxford Brookes University, Oxford OX33 1HX, United Kingdom. E-mail: {hzhu, ibayley}@brookes.ac.uk.*

Manuscript received 30 Sept. 2013; revised 10 Mar. 2015; accepted 31 May 2015. Date of publication 14 June 2015; date of current version 13 Nov. 2015. Recommended for acceptance by B.H.C. Cheng.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2445341

The remainder of the paper is organised as follows. Section 2 outlines our proposed approach and summarises the main contributions of this paper. Section 3 sets the foundation of the work by defining the mathematical notations and recalls the formal theory that the paper is based on. Section 4 examines the notion of valid pattern composition and instantiation. The notions of feature preservation, semantics preservation and soundness preservation are introduced and formally defined as conditions of valid pattern compositions and instantiations. Their interrelationships are studied. Section 5 is devoted to the verification of the validity of pattern compositions and instantiations expressed in terms of pattern operations. Section 6 applies the theory to overlap-based pattern composition operators. Section 7 reports a case study with a real example of pattern-oriented software design: a general request handling framework [42]. Finally, Section 8 concludes the paper with a comparison with related work and a discussion of future work.

2 The Proposed Approach

This section outlines our approach to the open problem of verifying that a composition and instantiation of design patterns is valid. We refine the problem to that of proving that a pattern composition and instantiation preserves three important qualities of the pattern:

- soundness, the existence of valid instances for the pattern, i.e., at least one design conforms to the pattern;
- semantics, the meaning of the pattern, which is the set of designs conforming to the pattern;
- features, the structural and behavioural properties of the pattern.

Another important quality of pattern specifications we will discuss is completeness, which means that it covers all the characteristic features of the pattern, no more no less.

In common with other researchers, we regard a design pattern as a predicate that asserts the existence of elements (e.g. classes) in the design, states structural properties in terms of how these elements are statically interconnected, and behavioural properties in terms of their dynamic interaction. Pattern compositions and instantiations are expressions

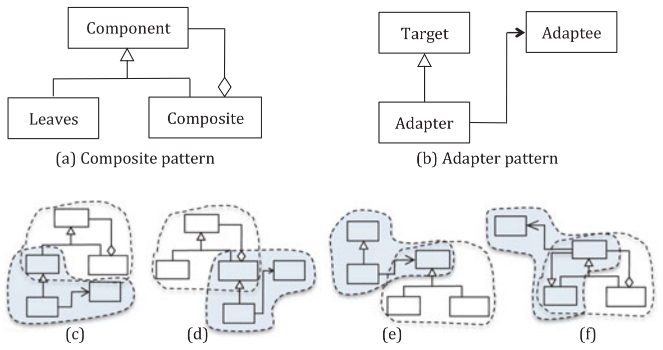


Fig. 1. Motivative examples of pattern compositions.

formed from the application of six pattern operators [32], [43], [44] to existing patterns. To determine validity, we investigate under what conditions the operators preserve soundness, semantics and features.

The main contributions of the paper are as follows.

- We formally define the notions of feature preservation, semantics preservation and soundness preservation, and thereby formalise the notion of valid composition. We also study the relationships between them.
- We present a formal method to enable software designers to prove or disprove the validity of pattern composition, by considering soundness preservation, semantics preservation and feature preservation. In particular, we prove that
  - all six operators are feature preserving,
  - operators that change the structural requirements are semantics preserving, and
  - operators that introduce new constraints fail to be soundness preserving only when the newly introduced constraints are in conflict with the semantics of the original pattern.
- We demonstrate the validity and applicability of the theory developed in this paper by two means:
  - a theoretical analysis of the validity conditions for pattern compositions based on overlaps [45],
  - a case study of a real pattern-oriented design.

3 Preliminaries

In this section, we first recall the logic underlying the formal specification of design patterns, then the pattern composition and instantiation operators [32].

3.1 Logics Underlying Pattern Specification and Reasoning

In the past few years, researchers have advanced several approaches to the formalisation of design patterns. In spite of the differences in their formalisms, the basic underlying ideas are quite similar. In particular, a pattern is usually specified using statements that constrain the structural features, and sometimes also the behavioural features, of its valid instances. The structural constraints are typically assertions that certain types of components exist and have a certain configuration. The behavioural constraints, on the other hand, detail the temporal order of messages exchanged between the components during the executions of an instance of the pattern. Note that negative information can also be included in pattern specifications, for example, to state the so-called *forbidden conditions*, such as that no associations are allowed between two particular components. Such negative conditions could be useful to validate the correct uses of patterns.

The various approaches to pattern formalisation differ in how they represent software systems and in how they formalise the predicate. For example, Eden's predicates are on the source code of object-oriented programs [40], [46], [47], [48] but they are limited to structural features. Taibi et al.'s approach in [38] is similar but he takes the further step of adding temporal logic for behavioural features. In contrast, our predicates are built up from primitive predicates on UML class and sequence diagrams [41]. These primitives are induced from the abstract syntax definition of UML diagrams in GEBNF, which is an extension of BNF for graphical modelling languages [49], [50]. Therefore, without loss of generality, a pattern specification is defined as follows.

Definition 1 (Formal specifications of design patterns). *A* formal specification of a design pattern *is an ordered pair $P = \langle Vars, Pred\rangle$, where $Pred$ is a predicate on the domain of software systems, and $Vars = \{v_1 : T_1, \ldots, v_n : T_n\}$ is a set of declarations for the variables that are free in the predicate $Pred$. Each v_i is a variable that represents a component in the pattern and T_i is that variable's corresponding type. A type can be a basic type Z of elements, such as class, method, attribute, message, lifeline, etc. in the design model, or $\mathbb{P}(Z)$ (i.e., a power set of Z), or $\mathbb{P}(\mathbb{P}(Z))$ to represent a set of sets of elements of the type Z, etc. Note that, for the sake of convenience, we do not allow the empty set $\emptyset$ to be an instance of a power set type $\mathbb{P}(T)$.*

The semantics of a specification is a ground predicate in the following form.

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

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

Often predicate $Pred$ is split into static and dynamic conditions as in [38] and [41]. It can also be specialised to particular representations of software systems such as program code, UML diagrams etc, though in this paper, for simplicity, we will just consider the latter for our concrete examples. The operators we use from [32], [43], [51] are also independent of the particular formalism, although the examples come from the previous work [41] and [52]. The theory developed in this paper is valid as far as the following notion of conformance is valid and the logic is consistent.

Give a specification of a design pattern, one can decide whether a concrete design conforms to the design pattern by demonstrating that the predicate is satisfied by the design. To prove such a conformance we just need to give an assignment α of variables in $Vars$ to elements in the design model m and evaluate $Pred(P)$ in the context of α. The evaluation of a predicate p in the context of an assignment α of variables in p to elements in a model m, denoted by $[\![p]\!]_\alpha^m$, is defined as usual in predicate logic. Thus, the definition is omitted for the sake of space. If the result of the evaluation $[\![Pred(P)]\!]_\alpha^m$ is $true$, we say that the model m satisfies the specification P, and write $m \models Spec(P)$.

Definition 2 (Conformance of a Design to a Pattern). *Let m be a model and $P = \langle Vars, Pred\rangle$ be a formal specification of a design pattern. The model m* conforms to *the design pattern as specified by P if and only if $m \models Spec(P)$. For the sake of simplicity, in the sequel we will also write $m \models P$ for $m \models Spec(P)$.*

Given a formal specification of a pattern P, we can also infer the properties of any system that conforms to it by deducing that $Spec(P) \Rightarrow q$ where q is a formula denoting a property of the model. In other words, every logical consequence of a formal specification is a property of every model that conforms to the pattern specified. This statement is true only if the logic interpretation of predicates is consistent with logic inference rules. Formally, we have the following proposition about the logic system underlying the formalism used for pattern specification.

Proposition 1 (Consistency of Specification Logic). *For all models m and predicates p and q on models, we have that $\vdash (p \Rightarrow q)$ and $m \models p$ imply that $m \models q$.*

Note that the logic system also has axioms about the atomic predicates of software systems. One such predicate is $\longrightarrow\!\!\triangleright$, where $X \longrightarrow\!\!\triangleright Y$ means that X is a subclass of Y. Two of its axioms are the transitivity and asymmetry properties below. $\forall X, Y, Z \in Class$,

$$(X \longrightarrow\!\!\triangleright Y) \wedge (Y \longrightarrow\!\!\triangleright Z) \Rightarrow X \longrightarrow\!\!\triangleright Z. \tag{2}$$

$$\neg(X \longrightarrow\!\!\triangleright Y \wedge Y \longrightarrow\!\!\triangleright X). \tag{3}$$

These well-formedness conditions are true for all valid UML models. For that reason, they can be used as axioms in reasoning about design patterns [26].

3.2 Relations and Operators on Design Patterns

Based on the formal logic underlying pattern specifications, we can define various relationships between patterns, one of which is the following specialisation relationship, which has been studied by a number of researchers in various contexts, such as [39], [53].

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, m also conforms to Q. Formally, $P \preccurlyeq Q \triangleq \forall m \cdot (m \models P \Rightarrow m \models Q)$.*

Two patterns P and Q are equivalent, *written $P \approx Q$, if $P \preccurlyeq Q$ and $Q \preccurlyeq P$.*

To establish that $P \preccurlyeq Q$, one can use logic inference in predicate logic to prove that $Spec(P) \Rightarrow Spec(Q)$.

Specialisation is a partial order with $FALSE$ as bottom and $TRUE$ as top, where $TRUE$ and $FALSE$ are special patterns defined as follows.

Definition 4 ($TRUE$ and $FALSE$ patterns). *Pattern $TRUE$ is the pattern that satisfies the condition that for all models m, $m \models TRUE$. Pattern $FALSE$ is the pattern that satisfies the condition that for all models m, $m \models FALSE$.*

The operators on patterns introduced in [32] are as defined below; see the original for explanations, examples and case studies.

Definition 5 (Pattern Operators). *Let P and Q be any given patterns, $V = Vars(P) = \{x_0 : T_0, \ldots, x_n : T_n\}$ and $Pred(P) = p(x_0, \ldots, x_n)$.*

1) *Restriction. Let c be a predicate on V. $P[c]$ is the pattern such that $Vars(P[c]) = V$ and $Pred(P[c]) = p \wedge c$.*
2) *Superposition. Assume that $V \cap Vars(Q) = \emptyset$. $P{*}\, Q$, is the pattern that $Vars(P * Q) = V \cup Vars(Q)$ and $Pred(P * Q) = p \wedge Pred(Q)$.*
3) *Extension. Let $V \cap U = \emptyset$, and c be a predicate on $V \cup U$. $P\#(U \bullet c)$ is the pattern such that $Vars(P\#(U \bullet c)) = V \cup U$ and $Pred(P\#(U \bullet c)) = p \wedge c$,*
4) *Flattening. Assume $T_0 = \mathbb{P}(T)$ and $x'_0 \notin V$. $P \Downarrow x_0 \backslash x'_0$ is the pattern such that*

$$Vars(P \Downarrow x_0\backslash x'_0) = \{x'_0 : T, x_1 : T_1, \ldots, x_n : T_n\};$$
$$Pred(P \Downarrow x_0\backslash x'_0) = p(\{x'_0\}, x_1, \ldots, x_n).$$

5) *Generalisation. $P \Uparrow x_0\backslash x'_0$ is the pattern such that*

$$Vars(P \Uparrow x_0\backslash x'_0) = \{x'_0 : \mathbb{P}(T_0), x_1 : T_1, \ldots, x_n : T_n\},$$
$$Pred(P \Uparrow x_0\backslash x'_0) = \forall x_0 \in x'_0 \cdot Pred(P).$$

6) *Lifting. Let $X = \{x_0 \ldots, x_k\}$, $n > k > 0$, and $xs_i \notin V$ for $i = 1, \ldots, n$. $P \uparrow X$ is the pattern such that*

$$Vars(P \uparrow X) = \{xs_0 : \mathbb{P}(T_0), \ldots, xs_n : \mathbb{P}(T_n)\},$$
$$Pred(P \uparrow X) = \forall x_0 \in xs_0 \cdots \forall x_k \in xs_k \cdot$$
$$\exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot p(x_1, \ldots, x_n).$$

Informal explanations of the operators are as follows:

Restriction operator $P[c]$ imposes an additional condition c on an existing pattern P. A common use of restriction, as shown in our case studies [43], is in the form $P[u = v]$, where u and v are variables of the same type. An alternative form $P[u = a]$ where a is a constant element is also useful for instantiating a pattern.

Superposition $P * Q$ is a pattern containing both pattern P and pattern Q. Naming clashes in component variables can always be resolved by systematic renaming. Let $x \in Vars(P)$ and $x' \notin Vars(P)$. The *systematic renaming* of x to x', written as $P[x\backslash x']$, does not change the meaning of the pattern. That is, for all models m that $m \models P \Leftrightarrow m \models P[x\backslash x']$. Another approach, which we prefer, is to write $P.x$ to denote the variable x in pattern P. Thus, the variable $P.x$ can be easily distinguished from $Q.x$.

Extension $P\#(U \bullet c)$ introduces a set U of new components into the pattern P and links these components with the existing ones according to the predicate c.

Flattening $P \Downarrow x\backslash x'$ forces the component x in P always to be a singleton $\{x'\}$. When there is no risk of confusion, the name x' can be omitted.

Generalisation $P \Uparrow x\backslash x'$ is the opposite of flattening. It allows an element x in pattern P to be repeated one or many times. Both the generalisation and flattening operators can be overloaded to be applied to a set X of component variables.

Lifting $P \uparrow X$ results in a pattern P' that contains a varying number of instances of pattern P. For example, $Adapter \uparrow Target$ is the pattern that contains a number of

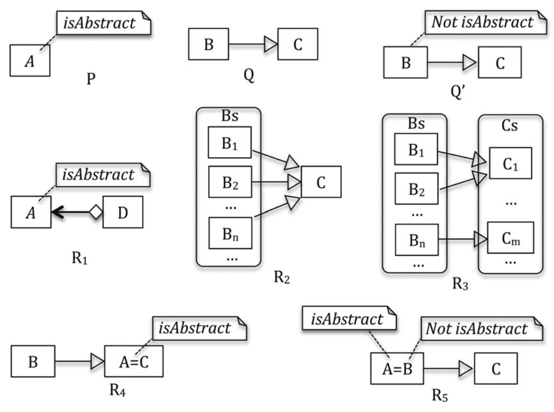


Fig. 2. Illustration of the patterns in Examples 1 to 4.

*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. The difference between lifting and generalisation is illustrated in Example 1.

Note that pattern specifications are closed formulae, containing no free variables. Although the names given to component variables improve readability significantly, 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 $P \Downarrow x\backslash x'$, and $P \Downarrow X$ to represent $P \Downarrow X\backslash X'$. For the lifting operator, when the key set X is singleton, we omit the set brackets for simplicity, so we write $P \uparrow x$ instead of $P \uparrow \{x\}$.

The following are some simple examples that illustrate the meanings of pattern operators. They are also used in the next section to illustrate the notions of validity of pattern compositions.

Example 1.

Consider patterns P and Q defined as below:

$$P = \langle \{A : Class\}, A.isAbstract \rangle$$
$$Q = \langle \{B : Class, C : Class\}, B \longrightarrow\!\!\!\!\triangleright\, C \rangle$$

where $X.isAbstract$ means that class X is an abstract class, $X \longrightarrow\!\!\!\!\triangleright\, Y$ means that class X is a subclass of Y. We have that

$$Spec(P) = \exists A : Class \cdot (A.isAbstract)$$
$$Spec(Q) = \exists B, C : Class \cdot (B \longrightarrow\!\!\!\!\triangleright\, C)$$

Consider the pattern compositions R_1 to R_4 defined as follows:

$$R_1 = P\#(D : Class \bullet D \diamond\!\!\longrightarrow A)$$
$$R_2 = Q \Uparrow B\backslash Bs$$
$$R_3 = Q \uparrow B\backslash Bs$$
$$R_4 = (P * Q)[A = C]$$

where $X \diamond\!\!\longrightarrow Y$ means class X contains class Y as a part, i.e., there is composite/aggregate relation from X to Y.

Informally, R_1 adds an additional component D to P and connects it to class A with an aggregation relation. R_2 generalises Q by allowing a number of classes $Bs = \{B_1, \ldots, B_n, \ldots\}$ to be C's subclasses instead of just one class B, whereas R_3 is the lifting of Q on the component B.

Note that, flattening is the inverse of generalization. Thus, Q is a flattening of R_2. R_4 is the composition of P and Q by unifying component A of pattern P with component C of pattern Q. These compositions are illustrated in Fig. 2.

From the definitions of the operators we immediately have:

$$Spec(R_1) = \exists A, D : Class \cdot (A.isAbstract \wedge D \diamond\!\!\longrightarrow A)$$
$$Spec(R_2) = \exists Bs : \mathbb{P}(Class), \exists C : Class \cdot (\forall B \in Bs \cdot (B \longrightarrow\!\!\!\!\triangleright\, C))$$
$$Spec(R_3) = \exists Bs, Cs : \mathbb{P}(Class) \cdot (\forall B \in Bs \cdot \exists C \in Cs \cdot (B \longrightarrow\!\!\!\!\triangleright\, C))$$
$$Spec(R_4) = \exists A, B : Class \cdot (A.isAbstract \wedge B \longrightarrow\!\!\!\!\triangleright\, A)$$

In [32], we proved a complete set of equational laws that the operators obey. Some of them are used in this paper to prove the theorems and in the case study. Thus, they are listed in the online Appendix[1].

4 The Notion of Validity

Our process to determine validity of compositions considers each of feature preservation, semantic preservation and soundness preservation in turn. We formally define these notions now and study the relationships between them.

4.1 Feature Preservation

If a pattern P has a certain feature, one would expect that a valid use of the pattern should also have the feature. The notion of feature preservation can be formally defined as follows.

Definition 6 (Feature Preservation). *A unary operator $\oplus$ on patterns is* feature preserving, *if, for any pattern P and any predicate p, pattern P has property p implies that $\oplus P$ also has property p. Formally,*

$$Spec(P) \vdash p \Rightarrow Spec(\oplus P) \vdash p.$$

A binary operator $\oplus$ on patterns is feature preserving, *if for any patterns P and Q and any predicate p, pattern P has the property p or pattern Q has the property p imply that $P \oplus Q$ also has property p. Formally,*

$$(Spec(P) \vdash p) \vee (Spec(Q) \vdash p) \Rightarrow (Spec(P \oplus Q) \vdash p).$$

The following lemma proves an important property of feature preservation operators.

Lemma 1 (Feature Preservation Lemma). *(a) An unary pattern operator $\oplus$ is feature preserving, if for all patterns P, $Spec(\oplus P) \Rightarrow Spec(P)$.*

(b) A binary pattern operator $\oplus$ is feature preserving, if for all patterns P and Q, $Spec(P \oplus Q) \Rightarrow Spec(P)$ and $Spec(P \oplus Q) \Rightarrow Spec(Q)$.

Proof. (a) Assume that for all patterns P, we have that $Spec(\oplus P) \Rightarrow Spec(P)$. Then, for any predicate p, $Spec(p) \vdash p$ implies that $Spec(\oplus P) \vdash p$ by the consistency of the logic. Thus, $\oplus$ preserves features.

1. The Appendix can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109//TSE.2015.2445341.

(b) Let p be any given property. If we have that $Spec(P) \vdash p$, by proposition 1, we have that $Spec(P) \Rightarrow p$. Because $Spec(P \oplus Q) \Rightarrow Spec(P)$, we have that $Spec(P \oplus Q) \Rightarrow p$. If we have that $Spec(Q) \vdash p$, then, by Proposition 1 we have that $Spec(Q) \Rightarrow p$. Because $Spec(P \oplus Q) \Rightarrow Spec(Q)$, we have that $Spec(P \oplus Q) \Rightarrow p$. Thus, we have that

$$(Spec(P) \vdash p) \vee (Spec(Q) \vdash p) \Rightarrow (Spec(P \oplus Q) \vdash p).$$

That is, $\oplus$ preserves features. □

An important property of pattern specifications is completeness, which means that it should capture all aspects of the design. If the specification is incomplete, a design may wrongly be regarded as an instance of the pattern, leading to a false positive. The formal definition of completeness is as follows.

Definition 7 (Completeness of Pattern Specification). *Let $P = \langle Vars, Pred \rangle$ be a formal specification of a given pattern, Thm be a set of statements on the properties that all instances of the pattern should possess. The specification P is* complete *with respect to Thm, if for all $p \in Thm$, we have that $Spec(P) \vdash p$.*

Because design patterns are documented informally and represent empirical knowledge, the completeness of a formal specification can only be judged manually, perhaps with the aid of examples. However, we would want a composition to preserve completeness when its components do. More formally,

Definition 8 (Completeness Preservation). *A unary operator $\oplus$ on patterns is* completeness preserving, *if, for any pattern P and set Thm of statements, P is complete with respect to Thm implies that $\oplus P$ is also complete with respect to Thm.*

A binary operator $\oplus$ on patterns is completeness preserving, *if for any patterns P and Q and any sets Thm_P and Thm_Q of statements, P is complete w.r.t. Thm_P and Q is complete w.r.t. Thm_Q imply that $P \oplus Q$ is complete w.r.t. $Thm_P \cup Thm_Q$.*

Fortunately, completeness preservation is guaranteed by feature preservation, as the following lemma states

Lemma 2 (Completeness Preservation Lemma). *(a) An unary pattern operator $\oplus$ is completeness preserving, if it is feature preserving.*

(b) A binary pattern operator $\oplus$ is completeness preserving, if it is feature preserving.

Proof. (a) Let P be any pattern specification that is complete w.r.t. a given set of statements Thm. By Definition 7, we have that for all $p \in Thm$, $Spec(P) \Rightarrow p$. Because $\oplus$ is feature preserving, we have that $Spec(\oplus(P)) \Rightarrow p$. Therefore, statement (a) of the lemma is true.

(b) Similarly, let P and Q be any pattern specifications complete w.r.t. sets of statements Thm_P and Thm_Q, respectively. By Definition 7, we have that

$$\forall p \in Thm_P \cdot (Spec(P) \Rightarrow p), \tag{4}$$

$$\forall q \in Thm_Q \cdot (Spec(Q) \Rightarrow q). \tag{5}$$

Now, let $s \in Thm_P \cup Thm_Q$. So $s \in Thm_P$ or $s \in Thm_Q$. If $s \in Thm_P$, by (4) and statement (b) of Lemma 1, we have that $Spec(P \oplus Q) \Rightarrow s$. Similarly, if $s \in Thm_Q$ then by (5) and statement (b) again, $Spec(P \oplus Q) \Rightarrow s$. So for all statements $s \in Thm_P \cup Thm_Q$ implies that $Spec(P \oplus Q) \Rightarrow s$. This means that $\oplus$ preserves completeness. □

Example 2. Consider the patterns in Example 1. It is easy to see that $Spec(R_1) \Rightarrow Spec(P)$. Thus, we can prove that for all p, $Spec(P) \Rightarrow p$ implies that $Spec(R_1) \Rightarrow p$ by transitivity of $\Rightarrow$. This means that for all properties p that pattern P has, pattern R_1 also has property p. In other words, pattern R_1 preserves the features of pattern P. Similarly, we also have

$$Spec(R_2) \Rightarrow Spec(Q)$$
$$Spec(R_3) \Rightarrow Spec(Q)$$
$$Spec(R_4) \Rightarrow Spec(P) \textit{ and } Spec(R_4) \Rightarrow Spec(Q).$$

This means that patterns R_2 and R_3 preserve the features of pattern Q, and pattern R_4 preserves the features of both patterns P and Q.

Note we do not allow the empty set $\emptyset$ to be an instance of power set type $\mathbb{P}(T)$, since if $Bs = \emptyset$ then $Spec(R_2)$ and $Spec(R_3)$ are vacuously true, even if $Spec(Q)$ is false. So this requirement is necessary for R_2 and R_3 to be feature preserving.

4.2 Preservation of Semantics

The semantics of a pattern is the set of designs that conform to it. More formally, we have

Definition 9 (Denotational Semantics of Patterns). *Let P be a pattern specification. The* denotational semantics *(or simply* semantics*) of P, denoted by $[\![P]\!]$, is the set of models m that satisfy the specification. Formally,*

$$[\![P]\!] \triangleq \{m | m \models Spec(P)\}.$$

By the above definition, it is easy to see that, for all patterns P and Q, we have

$$P \approx Q \Leftrightarrow [\![P]\!] = [\![Q]\!], \tag{6}$$

$$P \preccurlyeq Q \Leftrightarrow [\![P]\!] \subseteq [\![Q]\!]. \tag{7}$$

Some operators preserve the denotational semantics while changing the structural requirements, while others introduce new restrictions, and thereby change the semantics. Semantics preservation is formally defined as follows.

Definition 10 (Semantics Preservation Property). *A unary operator $\oplus$ on patterns is* semantics preserving *if for all patterns P we have that $[\![P]\!] = [\![\oplus P]\!]$.*

A binary operator $\oplus$ on patterns is semantics preserving *if, for all patterns P and Q, we have $[\![P \oplus Q]\!] = [\![P]\!] \cap [\![Q]\!]$.*

Obviously, a unary operator $\oplus$ preserves semantics, if and only if for all models m, $(m \models P) \Leftrightarrow (m \models \oplus P)$. For a binary operator $\oplus$, the operator $\oplus$ preserves semantics if and only if for any patterns P and Q, we have for all models m, $(m \models P \oplus Q) \Leftrightarrow ((m \models P) \wedge (m \models Q))$.

Example 3. Consider patterns P, Q and R_4 defined in Example 1. We have that

$$\begin{aligned}
& m \models Spec(R_4) \\
& \quad \Leftrightarrow m \models \exists A, B : Class \cdot (A.isAbstract \wedge B \longrightarrow\!\!\triangleright\, A) \\
& \quad \Rightarrow m \models \exists A : Class \cdot (A.isAbstract) \\
& \qquad \wedge \exists A, B : Class \cdot (B \longrightarrow\!\!\triangleright\, A) \\
& \quad \Leftrightarrow m \models \exists A : Class \cdot (A.isAbstract) \\
& \qquad \wedge m \models \exists A, B : Class \cdot (B \longrightarrow\!\!\triangleright\, A) \\
& \quad \Leftrightarrow m \models Spec(P) \wedge m \models Spec(Q)
\end{aligned}$$

Therefore, $[\![R_4]\!] \subseteq [\![P]\!] \cap [\![Q]\!]$.

On the other hand, $[\![R_4]\!] \neq [\![P]\!] \cap [\![Q]\!]$, because

$$\begin{aligned}
& \exists A : Class \cdot (A.isAbstract) \wedge \exists A, B : Class \cdot (B \longrightarrow\!\!\triangleright\, A) \\
& \not\Rightarrow \exists A, B : Class \cdot (A.isAbstract \wedge B \longrightarrow\!\!\triangleright\, A).
\end{aligned}$$

Therefore, pattern R_4 does not preserve the semantics of patterns P and Q, even though, as we saw in Example 2, it preserves their features.

4.3 Preservation of Soundness

A design pattern is sound if it has at least one instance. For example, the $FALSE$ pattern is not sound because it cannot be satisfied. Any operation $\oplus$ is soundness preserving if when applied to a sound pattern P it gives a sound pattern $\oplus P$.

Definition 11 (Soundness Preservation Property). *A unary operator $\oplus$ on patterns is* soundness preserving *if for any pattern P we have*

$$\exists m \cdot (m \models P) \Rightarrow \exists m \cdot (m \models \oplus P).$$

A binary operator $\oplus$ on patterns is soundness preserving *if for any patterns P and Q we have*

$$(\exists m \cdot m \models P) \wedge (\exists m \cdot m \models Q) \Rightarrow \exists m \cdot m \models (P \oplus Q).$$

The following lemma is useful.

Lemma 3. *If a pattern operator preserves semantics, it also preserves soundness.*

Proof. Here, we only give the proof for unary pattern operators. The proof for binary operators is very similar.

Let $\oplus$ be a unary operator that preserves semantics. By definition of semantics preservation, for all patterns P and models m, we have that $m \models P \Leftrightarrow m \models \oplus P$. If P is sound, i.e. there is a model m such that $m \models P$, then, we have that $m \models \oplus P$. That is, $\oplus P$ is also sound. Thus, $\oplus$ preserves soundness. □

Example 4. Let pattern P be as defined in Example 1. Let pattern Q' be the following.

$$\langle \{B, C : Class\}, (B \longrightarrow\!\!\triangleright\, C \wedge \neg B.isAbstract) \rangle$$

Then, we have that

$$Spec(Q') = \exists B, C : Class \cdot (B \longrightarrow\!\!\triangleright\, C \wedge \neg B.isAbstract)$$

Although P and Q' are sound, their composition might not be. For example, pattern R_5 is not sound,

$$R_5 = P * Q'[A = B]$$

because the following is not satisfiable.

$$\begin{aligned}
Spec(R_5) = \exists A, C : Class \cdot ((A \longrightarrow\!\!\triangleright\, C) \wedge \\
\neg A.isAbstract \wedge A.isAbstract),
\end{aligned}$$

From Examples 2 to 4, we can see that not all pattern compositions preserve semantics nor even soundness. The next section analyses which operators preserves these properties.

Knowledge of this will make validity much easier to determine without recourse again to logic as required above.

5 Analysis of Pattern Operators

Now we analyse the preservation properties of the operators, proving a set of general theorems. The lengthier proofs are given in online Appendix 2, available in the online supplemental material.

5.1 Feature Preservation Properties

Theorem 1 (Feature Preservation of Pattern Operators). *The restriction, extension, flattening, generalisation, superposition and lifting operators all preserve features.*

Note that, for all patterns P and Q, if $Spec(P) \Rightarrow Spec(Q)$, we have that $P \preccurlyeq Q$. Therefore, the feature preservation theorem means that applying any of the six pattern operators will not increase the set of instances of the pattern. This is because each of these operators either introduces additional constraints on the instances, or modifies the structure of the pattern without changing its semantics.

As shown in the case studies and examples given in [43], pattern compositions are expressions formed from patterns and the six operators. Using Theorem 1, by induction on the structure of expressions, we can prove all such pattern expressions are feature preserving. Thus, we have the following theorem.

Theorem 2 (Feature Preservation of Expressions). *For any expression E made up by applying the six operators to patterns P_i, for each i we have that $Spec(E) \Rightarrow Spec(P_i)$. This means that E preserves the features of P_i.*

Informally, Theorem 2 guarantees that any expression made up from the operators preserves features. We regard this as essential for the correctness of using patterns.

5.2 Semantics Preservation Properties

Theorem 3 (Semantics Preservation Properties). *Superposition, lifting and generation operators preserve semantics. That is, for all patterns P and Q, all sets $X \subseteq Vars(P)$, we have that for all models m,*

$$(m \models P * Q) \Leftrightarrow ((m \models P) \wedge (m \models Q)), \tag{8}$$

$$(m \models (P \uparrow X)) \Leftrightarrow (m \models P), \tag{9}$$

$$(m \models (P \Uparrow X)) \Leftrightarrow (m \models P). \tag{10}$$

An immediate corollary is the following.

Corollary 1. *For all patterns P and Q, we have that*

1) $[\![P * Q]\!] = [\![P]\!] \cap [\![Q]\!]$;
2) $[\![P \Uparrow x]\!] = [\![P]\!]$, *for all* $x \in Vars(P)$;
3) $[\![P \uparrow x]\!] = [\![P]\!]$, *for all* $x \in Vars(P)$.

These operators change the structure of the pattern without affecting conformance. They are usually applied, as seen in [43], in preparation for restriction and extension, which do affect conformance since they add constraints.

Theorem 4. *(Semantics of Restriction, Extension and Flattening) Let P be any given pattern, V a set of variables disjoint to $Vars(P)$, and c a given predicate. We have that*

$$[\![P[c]]\!] = \{m | m \in [\![P]\!] \wedge m \models c\}, \quad (11)$$

$$[\![P\#(V \bullet c)]\!] = \{m | m \in [\![P]\!] \wedge m \models \exists V \cdot c\}, \quad (12)$$

$$[\![P \Downarrow x]\!] = \{m | m \in [\![P]\!] \wedge m \models (||x|| = 1).\} \quad (13)$$

Note that Theorem 4 implies that $[\![P[c]]\!] \subseteq [\![P]\!]$, $[\![P\#(V \bullet c)]\!] \subseteq [\![P]\!]$, and $[\![P \Downarrow x]\!] \subseteq [\![P]\!]$. Using Theorem 3 as well, and induction on the structure of pattern expressions, we obtain:

Corollary 2. *For any expression E made up by applying the six operators to patterns P_i, for each i, we have that $[\![E]\!] \subseteq [\![P_i]\!]$.*

5.3 Soundness Preservation Properties

While each of the six operators preserve features, some do not preserve soundness. For example, restriction does not because $P[false]$ cannot be sound even if P is sound. However, Lemma 3 tells us that semantics preserving operators also are soundness preserving so we conclude:

Corollary 3. *The superposition, lifting and generalisation operators preserve soundness.*

Restriction, extension and flattening do not, however, as the following counterexamples show.

Example 5 (Counterexamples of Soundness Preservation).
(1) *Restriction.* Suppose $[\![P]\!] \neq \emptyset$. But $[\![P[false]]\!] = \emptyset$ because $P[false] \approx FALSE$.
(2) *Extension.* Suppose $[\![P]\!] \neq \emptyset$ again. But $[\![P\#\ (V \bullet false)]\!] = \emptyset$ because $P\#(V \bullet false) \approx FALSE$.
(3) *Flattening.* Suppose $P = \langle\{v : \mathbb{P}(Class)\}, ||v|| \geq 2\rangle$. Then designs exist that satisfy P so $[\![P]\!] \neq \emptyset$. However, from the definition of the flatten operator, $P \Downarrow v = \langle\{v' : Class\}, (||\{v'\}|| \geq 2)\rangle$. But $||\{v'\}|| \geq 2$ is not satisfiable so $[\![P \Downarrow v]\!] = \emptyset$.

From Theorem 4, we obtain the following conditions for these operators to lose soundness.

Corollary 4 (Conditions of losing soundness).
Let P be any given pattern. We have that

1) *$P[c]$ is not sound, if $Pred(P) \Rightarrow \neg c$.*
2) *$P\#(V \bullet c)$ is not sound if $Pred(P) \Rightarrow \neg\exists V \cdot c$.*
3) *$P \Downarrow x$ is not sound if $Pred(P) \Rightarrow (||x|| \neq 1)$.*

Informally, semantics is lost if a conflicting condition is introduced. These conditions are necessary as well as sufficient if the logic system is complete in the sense that $c \neq false$ implies that there is a model m such that $m \models c$, so these conditions are the strongest that one can get.

5.4 An Example

We now conclude the section by applying these theorems to our original motivating example of Fig. 1.

- *Feature Preservation.*

The compositions (c), (d) and (f) in Fig. 1 can be formally expressed using the operators as follows.

$$\begin{aligned}(c) &= Composite * Adapter[Leaves = Target]\\ (d) &= Composite * Adapter[Composite = Target]\\ (f) &= Composite * Adapter[Leaves = Target\\ &\quad \wedge Component = Adapter]\end{aligned}$$

So by Theorem 2, all the features of Composite and Adapter are present in these compositions. This is not true of (e), however, because the structural feature $Composite \diamond\!\longrightarrow Component$ is missing. So, (e) is not valid, and thus, cannot even be written as an expression.

- *Semantics Preservation.*

By Theorem 4, we have the semantics of (c)

$$\begin{aligned}[\![(c)]\!] &= \{m | m \in [\![Composite * Adapter]\!] \wedge m\\ &\models (Leaves = Target)\} \subseteq [\![Composite * Adapter]\!]\\ &= [\![Composite]\!] \cap [\![Adapter]\!]\end{aligned}$$

As $[\![Composite]\!] \neq [\![Adapter]\!]$, we have $[\![(c)]\!] \subset [\![Adapter]\!]$ and $[\![(c)]\!] \subset [\![Composite]\!]$. So (c) does not preserve semantics but instead restricts the semantics with a further condition. Compositions (d) and (f) are similar.

Informally, this means that the composition does not completely preserve the semantics of the composted patterns, but restricts the semantics with an additional condition. This is what one would expect.

In the same way, we can also prove a similar property for compositions (d) and (f).

- *Soundness Preservation.*

By Corollary 4 we have that composition (c) is not sound, if

$$Pred(Composite * Adapter) \Rightarrow \neg(Leaves = Target).$$

However, this is not provable. Since the logic system is complete, and Composite and Adapter are sound, we have that composition (c) is sound. Compositions (d) and (e) are also sound for similar reasons, but (f) is not. By Theorem 4, the semantics of (f) is

$$\begin{aligned}&\{m | m \in [\![Composite * Adapter]\!]\\ &\wedge m \models Leaves = Target \wedge Component = Adapter\}.\end{aligned}$$

Assume that a software system m satisfies the specifications of Composite and Adapter patterns as well as the conditions $Leaves = Target$ and $Component = Adapter$. Because

$$\begin{aligned}&Pred(Composite) \Rightarrow Leaves \longrightarrow\!\triangleright\ Component,\\ &Pred(Adapter) \Rightarrow Adapter \longrightarrow\!\triangleright\ Target,\end{aligned}$$

we have that $Leaves \longrightarrow\!\triangleright Adapter$ and $Adapter \longrightarrow\!\triangleright\ Leaves$. This contradicts the axioms about inheritance relation between classes, i.e., Equ. (3). So we have

$$Pred(Composite * Adapter) \Rightarrow$$
$$\neg(Leaves = Target \wedge Component = Adapter).$$

In summary, compositions (c) and (d) are valid. However, (e) and (f) are not, because (e) is not feature-preserving though it is implementable, and (f) is not sound and thus not implementable.

Note that proving the conditions of lost soundness can be performed by employing a theorem prover such as SPASS.[2] The details of using SPASS in the proof of the example above is given in online Appendix 3, available in the online supplemental material.

In conclusion, the validity of a pattern composition can be determined as follows. First, represent it using the six pattern operators. If this can be done then the composition is feature-preserving. Then, determine whether semantics and soundness are preserved. This is best done by applying the theorems we proved in this section rather than using the formal definitions directly. This is demonstrated in the next section in the analysis of overlap-based pattern compositions.

6 Analysis of Overlap-Based Compositions

In our previous work [45], pattern compositions are formally defined in terms of overlaps between components in the patterns composed. Three types of overlaps were identified. In this section, we re-express them using the pattern operators. By doing so, we can deduce their validity properties from the theorems proved above.

6.1 Expression of Overlaps in Pattern Operators

To define the notion of overlap, suppose that patterns P and Q are composed together in the form $P \otimes Q$. Then, if a model m conforms to this composition then m also conforms both to P and to Q, provided that the composition is sound. By the definition of conformance, we must have assignments α_1 and α_2 such that $[\![Pred(P)]\!]^m_{\alpha_1} = true$ and $[\![Pred(Q)]\!]^m_{\alpha_2} = true$. There is an *overlap* between two assignments if there is an element of the model m assigned to two variables, one in $Vars(P)$ and the other in $Vars(Q)$. There are three types of overlaps, distinguished by whether the variables are elements (one-to-one), sets of elements (many-to-many) or one of each (many-to-one or one-to-many). The following defines composition with various types of overlaps using the pattern operators.

Definition 12 (Composition with One-to-One Overlap). *Let P and Q be design patterns. Let $v \in Vars(P)$ and $u \in Vars(Q)$ be variables of the same type T, i.e., $v, u : T$. Then, the* composition of P and Q with one-to-one overlap *v —u, written $P\langle v$ —$u\rangle Q$, is defined as follows:*

$$P\langle v \text{ —} u\rangle Q \triangleq (P * Q)[v = u].$$

Definition 13 (Composition with Many-to-Many Overlap). *Let P and Q be design patterns. Let $vs \in Vars(P)$ and $us \in Vars(Q)$ be variables assigned to sets of model elements of the same type $\mathbb{P}(T)$, i.e., $vs, us : \mathbb{P}(T)$. Then, the* composition of P and Q with many-to-many overlap *$vs \succ\!\!\prec us$, written $P\langle vs \succ\!\!\prec us\rangle Q$, is defined as follows:*

$$P\langle vs \succ\!\!\prec us\rangle Q \triangleq (P * Q)[vs \cap us \neq \emptyset].$$

For example, in Definition 12, T could be the type $Class$, and then v and u would be classes. In Definition 13, vs and us would be sets of classes.

Alternative formulations of many-to-many overlaps are possible, by instantiating the general form below for R bound to $\subseteq$, $\subset$ and $=$.

$$P\langle vs \succ\!\!\prec_R us\rangle Q \triangleq (P * Q)[vs \; R \; us].$$

Theorem 5 (Ordering among Many-to-Many Compositions). *For all patterns P and Q, we have that*

$$(P\langle vs \succ\!\!\prec_\subset us\rangle Q) \preccurlyeq (P\langle vs \succ\!\!\prec_\subseteq us\rangle Q) \quad (14)$$

$$(P\langle vs \succ\!\!\prec_= us\rangle Q) \preccurlyeq (P\langle vs \succ\!\!\prec_\subseteq us\rangle Q) \quad (15)$$

$$(P\langle vs \succ\!\!\prec_\subseteq us\rangle Q) \preccurlyeq (P\langle vs \succ\!\!\prec us\rangle Q) \quad (16)$$

Proof. The ordering relations follow the algebraic laws of the pattern operators (See [32] for details) and the fact that $(vs \subset us) \Rightarrow (vs \subseteq us)$, $(vs = us) \Rightarrow (vs \subseteq us)$ and $(vs \subseteq us) \Rightarrow (vs \cap us \neq \emptyset)$. □

The third sort of composition is defined as follows.

Definition 14 (Composition with One-to-Many Overlap). *Let P and Q be design patterns. Let $v \in Vars(P)$ be a variable assigned to a model element and let $us \in Vars(Q)$ be a variable assigned to sets of model elements of the type of v; i.e., $v : T$ and $us : \mathbb{P}(T)$. Then, the* composition of P and Q with one-to-many overlap *$v \text{ —}\!\!\prec us$, written $P\langle v \text{ —}\!\!\prec us\rangle Q$, is defined as follows:*

$$P\langle v \text{ —}\!\!\prec us\rangle Q \triangleq (P * Q)[v \in us]$$

Naturally, a composition with many-to-one overlap can also be defined by symmetry. The version in [45] however is slightly more complex in that P is first lifted to duplicate its class components. It is defined as follows.

Definition 15 (Composition with Lifted One-to-Many Overlap). *Let P and Q be design patterns. Let $v \in Vars(P)$ be a variable assigned to a model element and let $us \in Vars(Q)$ be a variable assigned to sets of model elements of the type of v; i.e., $v : T$ and $us : \mathbb{P}(T)$. Then, the* lifted composition of P and Q with one-to-many overlap *$v \text{ —}\!\!\prec us$ is defined as follows:*

$$P\langle v_\uparrow \text{ —}\!\!\prec_\subseteq us\rangle Q \triangleq (P \uparrow (v \backslash vs) * Q)[vs \subseteq us]$$

Many alternatives to this are possible. Lifting could be replaced by generalisation, for example, duplicating only the generalised component. Also, the constraints $vs \subseteq us$ could be specialised to $vs = us$, $vs \subset us$, etc.

$$P\langle v_\uparrow \text{ —}\!\!\prec_\subset us\rangle Q \triangleq (P \uparrow (v \backslash vs) * Q)[vs \subset us]$$
$$P\langle v_\uparrow \text{ —}\!\!\prec_= us\rangle Q \triangleq (P \uparrow (v \backslash vs) * Q)[vs = us]$$
$$P\langle v_\Uparrow \text{ —}\!\!\prec_\subseteq us\rangle Q \triangleq (P \Uparrow (v \backslash vs) * Q)[vs \subseteq us]$$
$$P\langle v_\Uparrow \text{ —}\!\!\prec_\subset us\rangle Q \triangleq (P \Uparrow (v \backslash vs) * Q)[vs \subset us]$$
$$P\langle v_\Uparrow \text{ —}\!\!\prec_= us\rangle Q \triangleq (P \Uparrow (v \backslash vs) * Q)[vs = us]$$

2. http://www.spass-prover.org

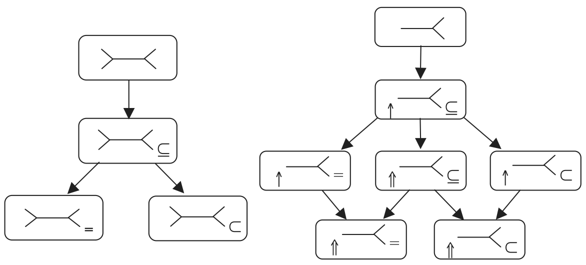


Fig. 3. Relationships between compositions with overlaps.

By applying the algebraic laws we can easily prove that these compositions have the following relationships.

Theorem 6. *For all patterns P and Q, we have that*

$$P\langle v_{\uparrow} \multimap_{R} us\rangle Q \preccurlyeq P\langle v_{\Uparrow} \multimap_{R} us\rangle Q,$$
$$P\langle v_{\uparrow} \multimap_{\subseteq} us\rangle Q \preccurlyeq P\langle v \multimap us\rangle Q.$$

where R is one of the relations $\subseteq$, $\subset$ and $=$.

Note that, by definition of many-to-many overlaps and one-to-many overlaps, the ordering relations given in Theorem 5 also hold among $P\langle v_{\uparrow} \multimap_{R} us\rangle Q$ for R to be $\subseteq$, $\subset$ and $=$, and among $P\langle v_{\Uparrow} \multimap_{R} us\rangle Q$.

The above $\preccurlyeq$ relationships between these composition operators are summarised in Fig. 3, where nodes represent various composition operators and an arrow from node A to node B means $A \preccurlyeq B$. On the right-hand side of Fig. 3 are the ordering relations given in Theorem 5. On the left-hand side are the $\preccurlyeq$ relationships between the one-to-many overlap composition operators.

6.2 Validity of Overlap-Based Compositions

By the theorems of feature preservation, semantics preservation and soundness preservation of the operators used to define the composition with overlaps, we know at once that the validity of overlap-based composition follows from their definitions.

Theorem 7 (Validity of Overlap Compositions).

(I) One-to-One Overlaps. *For a one-to-one overlap composition $P\langle v —u\rangle Q$, we have that*

1) *it preserves features;*
2) *its semantics $[\![P\langle v —u\rangle Q]\!]$ is:*

$$\{m \mid m \in [\![P]\!] \cap [\![Q]\!] \wedge m \models (v = u)\};$$

3) *it loses soundness, if $Pred(P) \wedge Pred(Q) \Rightarrow \neg\,(v = u)$.*

(II) Many-to-Many Overlaps. *For a many-to-many overlap composition $P\langle vs \succ\!\!\multimap_{R} us\rangle Q$, we have that*

1) *it always preserves features;*
2) *its semantics is:*

$$\{m \mid m \in [\![P]\!] \cap [\![Q]\!] \wedge m \models (vs\ R\ us)\};$$

3) *it loses soundness, if*

$$Pred(P) \wedge Pred(Q) \Rightarrow \neg(vs\ R\ us),$$

where $(vsRus)$ is $(vs \subseteq us)$, $(vs \subset us)$, $(vs = us)$, or $(vs \cap us \neq \emptyset)$.

(III) One-to-Many overlaps. *For a one-to-many overlap composition $P\langle v_{\dagger} \multimap_{R} us\rangle Q$, where $\dagger$ is either $\uparrow$ or $\Uparrow$ and R is the same as in (II), we have that*

1) *it always preserves features;*
2) *its semantics is*

$$\{m \mid m \in [\![P]\!] \cap [\![Q]\!] \wedge m \models (vs\ R\ us)\};$$

3) *it loses soundness, if*

$$(Pred(P\dagger v/vs) \wedge Pred(Q) \Rightarrow \neg(vs\ R\ us)).$$

Proof. Here, we only give the proof of (I). The proofs for (II) and (III) are very similar.

For 1), as shown in the previous section, a one-to-one overlap composition $P\langle v —u\rangle Q$ can be expressed with the pattern operators as follows:

$$P\langle v —u\rangle Q = (P * Q)[v = u] \qquad (Def.12)$$

Therefore, by Theorem 2, such a one-to-one overlap composition preserves features.

For 2), we have that

$$\begin{aligned} &[\![P\langle v —u\rangle Q]\!] \\ &\quad = [\![(P * Q)[v = u]]\!] (Def.12) \\ &\quad = \{m | m \in [\![P * Q]\!] \wedge m \vdash v = u\} && (Thm.4) \\ &\quad = \{m | m \in [\![P]\!] \cap [\![Q]\!] \wedge m \vdash v = u\} && (Thm.3) \end{aligned}$$

For 3), by Corollary 4 and Definition 12, a one-to-one overlap composition $P\langle v —u\rangle Q$ loses its soundness, if $Pred(P * Q) \Rightarrow \neg(v = u)$. By Definition 5, we have that $Pred(P * Q) = Pred(P) \wedge Pred(Q)$. Thus, statement 3) is true. □

7 A Case Study

In this section we report a case study in which a pattern-oriented design approach is used to develop a general request handling framework RHF [42].

Pattern-oriented design is a process of repeatedly recognising a design problem, identifying a design pattern to solve it and then applying the pattern by instantiating it and composing it to the design. Table 1 summarises the five design decisions that result in the design depicted in Fig. 4.[3] In [32], it is demonstrated that these design decisions can be formally expressed using pattern operators. When the formal design is compared manually with the original design depicted in Fig. 5, mismatches between them were detected.

Now, as a further contribution, we demonstrate that the theory developed in this paper will not only enable us to identify the differences between the original design and the formal design as detected in our previous case study [32], but will also enable us to formally prove that the differences are indeed errors in the manual design and that the formal design is valid. Moreover, the validity proofs can be automated by employing a theorem prover.

3. Note that, there are two different versions of Command Processor pattern in the literature by the same group of authors [9], [54]. The one used in [42] is the one given in [9].

TABLE 1
Design Decisions Made in the Design of Request Handling Framework

Design problem	Solution
The requests to the system are issued by the clients, who may be human users or other computer systems. Such requests must be objectified.	Apply the Command pattern that consists of an abstract class Command which declares a set of abstract methods to execute client requests. A set of ConcreteCommand subclasses implement these methods.
Multiple clients issue requests independently. A central component should coordinate the handling of these requests.	Use the CommandProcessor pattern to provide such coordination. The clients pass concrete commands to a CommandProcessor component for further handling and execution. It is inserted in between client and the Command class.
The system need to support undoing the actions performed in response to requests.	Use Memento pattern. The Memento component maintains copies of the states of the Originator, which is the Application class. *The* Caretaker *component creates a memento*, holds it over time, and if needed, passes it back to the Originator.
Requests from client must be logged. Requests from different users may be logged differently.	Apply Strategy pattern. The CommandProcessor passes the requests it received to a logging context, i.e. the context role in Strategy, which implements the invariant parts of the logging service and delegates the customer-specific logging aspects to the ConcreteStrategy component in Strategy.
The system should support compound commands, which are aggregates of other commands executed in a particular order.	Use the Composite pattern with atomic commands as the Leaves and compound commands as the Composite. Thus, add a new class CompoundCommand and an whole-part relation from this new class to the Command class.

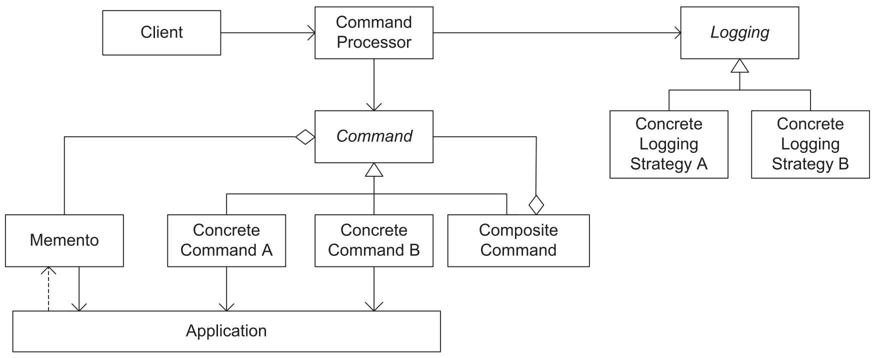


Fig. 4. Design of request handling framework as derived from the formal definition of RHF.

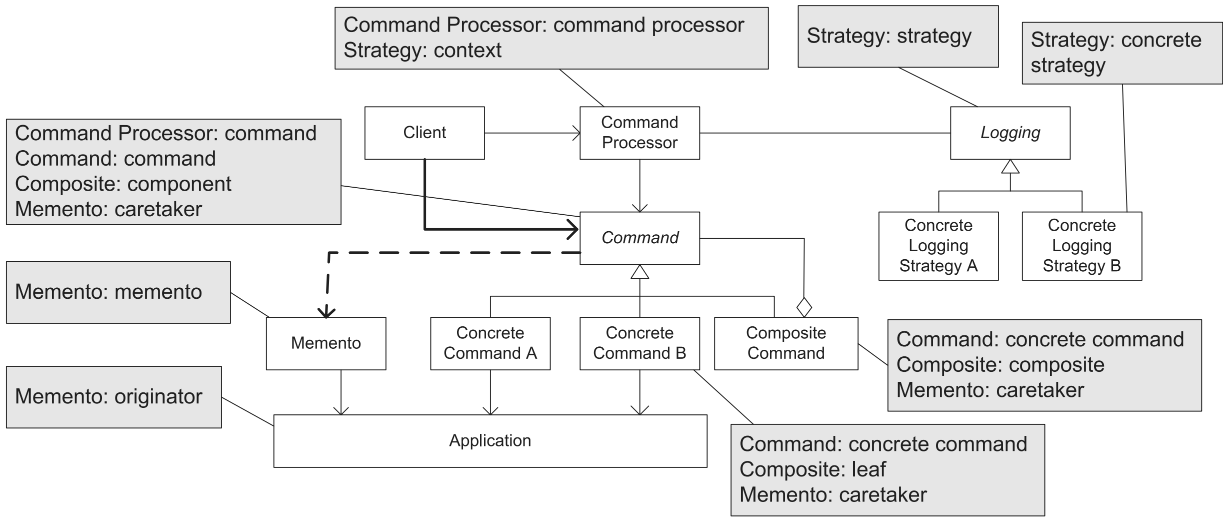


Fig. 5. Original design of request handling framework as in [42].

7.1 Feature Preservation

As pointed out in [32], there is a mistake in the original design. That mistake is that, in the definition of the Memento pattern, the *originator creates a state* and passes it to the caretaker component, which then holds the state and passes it back to the originator when needed [2]. However, in the design presented in [42], the caretaker creates the states. Therefore, this feature in the Memento pattern is not preserved in the design.

7.2 Semantics Preservation

Another problem with the original design is that it has a structural feature that $Client \longrightarrow Command$. By Theorem 3, we have that

$$m \in [\![RHF_O]\!] \Rightarrow m \models (Client \longrightarrow Command).$$

where RHF_O denotes the original design presented in [42]. Informally, this means the design allows the client to send requests directly to the command, bypassing command processor, and therefore not logging. We believe this is not what the designer intended to do, so it is a semantics error and is removed from the revised version of the design.

Fixing the above two problems led to the revised design depicted in Fig. 4.

On the other hand, the design decisions given in Table 1 can be formally expressed using pattern operators as follows, where RHF is the final result.

$$\begin{aligned}
RHF_1 &\triangleq Command[Invoker = Client,\\
&\quad Receiver\backslash Application]\\
RHF_2 &\triangleq RHF_1 * CommandProcessor\\
&\quad [Command = Component\\
&\quad \wedge Client = CommandProcessor]\\
RHF_3 &\triangleq RHF_2 * Memento\\
&\quad [Originator = Application,\\
&\quad Command \longrightarrow Caretaker]\\
RHF_4 &\triangleq RHF_3 * Strategy\\
&\quad [Context\backslash LoggingContext, Strategy\backslash Logging,\\
&\quad ConcreteStrategies\backslash ConcreteLoggingStrategies]\\
&\quad [CommandProcessor \longrightarrow LoggingContext]\\
RHF_5 &\triangleq RHF_4 * Composite\\
&\quad [Leaves = ConcreteCommands\\
&\quad \wedge Component = Command]\\
&\quad [Composite\backslash CompositeCommand]\\
RHF &\triangleq RHF_5[Caretaker = Command]\\
&\quad [CommandProcessor = LoggingContext]
\end{aligned}$$

By applying algebraic laws, we can rewrite this to the following, which exactly matches the diagram in Fig 4.

$$\begin{aligned}
&RHF \approx TRUE\\
&\quad \#(\{Client, Application, CommandProcessor, Logging,\\
&\quad Command, CompositeCommand, Memento : Class,\\
&\quad ConcreteLoggingStrategies,\\
&\quad ConcreteCommands : \mathbb{P}(Class)\}\\
&\bullet ((Client \longrightarrow CommandProcessor)\\
&\quad \wedge \forall CC \in ConcreteCommands \cdot (CC \longrightarrow Application\\
&\quad \wedge CC \longrightarrow\!\!\triangleright\ Command \wedge \neg\ isAbstract(CC))\\
&\quad \wedge (CommandProcessor \longrightarrow Command)\\
&\quad \wedge (Command \ \diamond\!\!\longrightarrow Memento)\\
&\quad \wedge (Application \longrightarrow memento)\\
&\quad \wedge (CommandProcessor \ \diamond\!\!\longrightarrow Logging)\\
&\quad \wedge \forall CL \in ConcreteLoggingStrategies \cdot (CL \longrightarrow\!\!\triangleright\ Logging)\\
&\quad \wedge isInterface(Command)\\
&\quad \wedge isInterface(Logging)\\
&\quad \wedge (CompositeCommand \longrightarrow\!\!\triangleright^{*} Command)\\
&\quad \wedge (CompositeCommand \ \diamond\!\!\longrightarrow {}^{+}Command)))
\end{aligned}$$

Therefore, by Theorem 2, we can conclude that the revised design is feature preserving.

7.3 Soundness Preservation

By applying the algebraic laws of pattern operators [32], we can also prove that

$$\begin{aligned}
RHF \approx (&Command * CommandProcessor * Memento *\\
&Strategy * Composite)[Connection]
\end{aligned}$$

where $Connection$ is the conjunction of the following predicates.

$$\begin{aligned}
&Command = Caretaker,\\
&Command = Component,\\
&Originator = Application,\\
&Command.Client = CommandProcessor,\\
&Originator = Application,\\
&Leaves = ConcreteCommands,\\
&Component = Command,\\
&CommandProcessor = Context,\\
&Caretaker = Command
\end{aligned}$$

By Corollary 4, the revised design loses its soundness if the following is true.

$$\begin{aligned}
&Pred(Command * CommandProcessor * Memento *\\
&Strategy * Composite) \Rightarrow \neg Connection.
\end{aligned}$$

Using the theorem prover SPASS, we can show this is not true; see online Appendix 3, available in the online supplemental material, for details. So, soundness isn't lost.

In conclusion, we have demonstrated again how to analyse the validity of a pattern composition. In our previous case study [32], we found two differences between the original manual design and our formal design. In this paper, we can confirm that the differences are errors in the manual design; one is feature preservation error and the other is a semantics error. We have also proved that our revised design is valid in terms of its preservation of feature and soundness.

8 Conclusion

Although each pattern is specified separately, they are usually to be found composed with each other [55]. Thus, pattern composition plays a crucial role in the effective use of design knowledge, whereas wrongly used patterns may impose a negative impact on software quality. In this paper, we formalised the notion of the validity of pattern compositions and instantiations by defining feature preservation, semantics preservation and soundness preservation. We studied these properties for the operators proposed in [32], [43]. The theory is applied to the theoretical analysis of pattern compositions represented as overlaps between patterns and a case study of a real pattern-oriented design, thereby demonstrating their utility in formally proving the validity of designs. Where there is an error, we can distinguish feature preservation problems from semantic errors and soundness lost.

8.1 Comparison with Related Works

Existing related work can be classified into two categories: (a) the representation of pattern composition and instantiation, and (b) the validation of pattern applications when they are composed and instantiated.

8.1.1 Representation of Pattern Compositions and Instantiations

Compositions can be represented either visually or formally.

- *Visual representation*

This is usually informal [42], [56]. Visual notations such as the Venn diagram with *Pattern:Role* annotation proposed by Vlissides [57] have been widely used in practice to show the component parts of the composition. Dong et al. [58] developed both static and dynamic techniques for visualizing the applications of design patterns. They defined UML profiles and implemented a tool, deployed as a web service, that represents the application of patterns in UML diagrams. This is done by UML profiles to attach information to designs through stereotypes, tagged values, and constraints. Such information is delivered dynamically with the movement of the user's mouse cursor on the screen. Their experiments show that this dynamic delivery helps to reduce the apparent complexity of the design. More recently, Smith [59] proposed the PIN notation (Pattern Instance Notation) to represent the same information in a hierarchical manner.

- *Formal representation of pattern compositions*

Very few authors have studied pattern compositions formally despite the large number of works on formalisation of design patterns. Dong et al. and Taibi do so in [60] and [61], respectively.

In Dong et al.'s approach, a composition of two patterns is a pair of mappings each of which link components from a pattern to the result pattern. Formally, for a composition P of $P_1, \ldots, P_n$, Dong et al. define a composition mapping $C : Vars(P_1) \times \cdots \times Vars(P_n) \to Vars(P)$ that associates names of component pattern P_i to those of P, which is mathematically equivalent to a set of *name mappings* $C_i : Vars(P_i) \to Vars(P), i = 1, \ldots, n$.

Dong et al. demonstrated how the structural and behavioural properties of the composite pattern can be derived from the original patterns and applied this to the study of security design patterns [62]. Let each P_i have a set θ_i of properties and the composition have the set θ of properties. The derivation of the properties of a composed pattern is actually a mapping M that extends C. It translates the sentences in each θ_i to θ, preserving the types of variables.

In [63], Dong et al. define instantiation again as a mapping, but from components in the pattern (e.g. classes, attributes, methods) to corresponding instances in the actual system.

Taibi [61], [64] took a very similar approach to Dong et al., but instead of defining mappings between the components of composed patterns, he directly renames the components and combines the predicates from the pattern specification. The variables in the predicates of the patterns to be composed are substituted with new variables of the result pattern or with constants to represent instantiation. Formally, for patterns P_1 and P_2 with properties φ_1 and φ_2 their composition is given by

$$Subst\{v_1 \backslash t_1, \ldots, v_n \backslash t_n\}(\varphi_1 \wedge \varphi_2)$$

where the terms t_i are either variables or constants.

Mathematically speaking, these substitutions are equivalent to the name mappings of Dong et al. Both must preserve types of variables for the resulting formulae to be well-typed and for that reason both approaches can only express one-to-one and many-to-many overlaps, but not one-to-many overlaps.

Both of these approaches effectively specify how components from the composed patterns overlap in the composite pattern. In our previous work [45], a pattern composition operator was formally defined based on the notion of overlaps between the elements of composed patterns. There are three types of overlap: one-to-one, many-to-many and one-to-many. Dong and Taibi's approaches can handle the first two but cannot easily be extended to one-to-many overlaps because the latter requires linking component names of different types and therefore cannot be defined as mappings between component names (Dong's approach), nor as renaming of component identifiers (Taibi's approach).

In [32], [43], [51], we developed a formal calculus of design patterns, consisting of:

- *A set of operators on design patterns* in which pattern compositions and instantiations can be expressed.
- *A set of algebraic laws* that these operators obey so that two different compositions can be proven equal.
- *A normalisation process* that transforms pattern expressions into a normal form. The process always terminates with a unique normal form up to logic equivalence.

As shown in [43], these operators are expressive enough to capture all pattern compositions suggested by Gamma et al. [2], and the normalisation process with algebraic laws can be used in a pattern oriented design processes, as demonstrated in [32] with a case study based on a real software design example. In this paper, we further proved the expressiveness of the set of six operators by using them to express the overlap-based operator.

8.1.2 Validation of Pattern Compositions and Instantiations

The impact on software quality, both positive and negative, of using design patterns has been studied empirically, for example, by Huston [65], Prechelt et al. [66], Khomh and Guéhéneuc [27] and Mouratidou et al. [31], etc.

Wendorff [30] observed that there are two different ways in which patterns can be misused.

1) the pattern's intent might not fit the project's requirements. Research efforts to address this include Hsueh et al.'s quantitative approach [67], which uses quality metrics to measure the improvement effectiveness, and Ampatzoglou et al.'s methodology [68] of impact assessment.
2) the pattern may be misapplied by software developers who misunderstand the rationale. This is the subject of this paper and few have addressed it.

Dong et al. [60] were perhaps the first who studied the 'correctness' of compositions of design patterns. Given their definition of a pattern composition as a set $C = <C_1, \ldots, C_n>$ of mappings from patterns $P_1, \ldots, P_n$ to be composed to the result pattern P, they proposed the following *faithfulness conditions* to ensure that pattern composition makes sense.

1) the mappings must agree on shared objects and parts,
2) it must not be possible to infer new facts about the patterns being composed from the result of their composition, and
3) all the properties of the composed patterns must also be true in the resultant composite pattern.

Their faithfulness conditions were formally defined as follows. Let θ_i be the properties of pattern P_i that are being composed, θ be the set of properties of the pattern P of the result of composition, and M be the mapping for translating properties of P_i to properties of P. Then,

1) for all variables x_1 and x_2, $C(x_1) = C(x_2)$ implies that $Type(x_1) = Type(x_2) = Type(C(x_1))$;
2) for every sentence S, if $S \in \theta_i$ then $M(S) \in \theta$;
3) if $S \notin \theta_i$ then $M(S) \notin \theta$.

In [60], Dong et al. also showed how to verify these conditions with an example where Composite is composed with Iterator. However, there was no theory or method for proving their faithfulness conditions in general.

Condition (1) above ensures that the results of the translation of formulas are well formed. However, it limits the pattern compositions to be only valid for one-to-one overlaps. Condition (2), that composition should not lose properties, is similar to our feature preservation condition, but weaker because the compositions are limited to one-to-one overlaps. Condition (3), means that composition should not gain properties, but it is very difficult if not impossible to prove. Moreover, it is not necessary, as argued by Taibi and Ngo [64]: "*while the second condition of faithfulness is relevant to component, it is not always necessary in the case of patterns*". In fact, a composition of patterns may well introduce additional properties as shown in our case studies.

Taibi and Ngo [64] also observed the faithfulness conditions (1) and (2) of Dong but only informally explained why his example satisfied condition (2). There is a lack of formal methods either for proving or for disproving faithfulness.

Our feature preservation property ensures that no features are lost from the original pattern, whereas our semantic preservation property ensures that no features are added. This latter property may be too strong, as extra features are often wanted, so soundness preservation is used instead as a minimal requirement that the added features do not cause a conflict. Dong and Taibi do not have such a condition. But what distinguishes our approach even more is that they have no systematic methods to prove their faithfulness conditions, whereas we have the general theorems about when soundness is preserved and how semantics are changed when patterns are composed. In addition, we can apply algebraic laws for the operators and automated theorem provers to prove feature preservation and soundness as demonstrated in the case study.

Interactions and conflicts between patterns were also discussed by Bottoni et al. for a different approach to pattern formalization [69]. Their pattern formalization approach is general for specifying patterns of all types of models, including OO designs, workflow models, etc. Their approach is graphical but formally based on category theory. They express patterns as triples of graphs (source, target and correspondence). These represent, respectively, the structure or configuration of the pattern, the roles of the pattern, giving the vocabulary of the application domain, and the mapping from this structure to these roles. Pattern satisfaction, composition and expansion were all defined as graph operations. Graphs also represent constraints with constraint satisfaction defined in terms of graph matching. Our power set types are represented in their notation by variable parts, visualized as triangles. They discuss pattern composition informally with an example and identify three types of conflicts.

- *Fatal conflicts*, which result in unsatisfiable compositions, i.e., loss of soundness.
- *Conflicts affecting satisfaction*, which change parts of elements in the design that constitute an instance of the pattern.
- *Conflicts between invariants*, which change the semantics of the invariants.

Obviously, the second and third types of conflict cannot be considered to be invalid pattern compositions. It is unclear however how to validate a pattern composition, e.g. to prove that it is satisfiable without a conflict.

8.2 Future Work

It would be useful to have tools to prove soundness for specific compositions and to support equational reasoning on them. Our case study employed the automated theorem prover SPASS and it indicates that it is feasible to design and implement such a tool.

The composition of OO design patterns has also been studied in the context of aspect-oriented programming (AOP) [70], in which overlap based compositions can be implemented by employing a crosscutting mechanism. Cacho et al. demonstrated in an empirical study that under certain conditions such blending of design patterns could achieve a better modularity than by simply merging statements, methods and/or classes in overlapped pattern components using traditional OO programming languages. The notions of feature preservation, semantics preservation and soundness preservation for the validity of pattern compositions proposed in this paper should be applicable to such an implementation of pattern compositions. The pattern operators express pattern composition and instantiation at a high level of abstraction, and thus they are independent of the way that pattern compositions are implemented. Therefore, the theory presented in this paper should also be applicable to the blending of design patterns. It is worth conducting some empirical study to demonstrate how to apply the theory to prove or disprove the validity of pattern blending in practice.

An interesting research questions is: how expressive are the pattern operators? In our previous work [43], we demonstrated that the six pattern operators can express all pattern compositions documented by Gamma et al. in [2]. In

[32], we demonstrated that they can express the design decisions made in a real pattern-oriented design process of the general request handling framework. In this paper, we have also demonstrated that all overlap-based compositions can be expressed using the six operators. However, how to formally define the notion that a set of operators is complete and to prove or disprove that the set of six operators is complete still remains open for future work.

REFERENCES

[1] P. Coad, "Object-oriented patterns," *Commun. ACM*, vol. 35, no. 9, pp. 152–159, Sep. 1992.

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

[3] M. Grand, *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1*. New York, NY, USA: Wiley, 2002.

[4] M. Grand, *Patterns in Java, Volume 2*. New York, NY, USA: Wiley, 1999.

[5] M. Grand, *Java Enterprise Design Patterns*. New York, NY, USA: Wiley, 2002.

[6] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Jun. 2003.

[7] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2003.

[8] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley, 2004.

[9] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture. Vol 4: A Pattern Language for Distributed Computing*. West Sussex, England: Wiley, 2007.

[10] M. Voelter, M. Kircher, and U. Zdun, *Remoting Patterns*. West Sussex, England: Wiley, 2004.

[11] M. Schumacher, E. Fernandez, D. Hybertson, and F. Buschmann, *Security Patterns: Integrating Security and Systems Engineering*. West Sussex, England: Wiley, 2005.

[12] C. Steel, *Applied J2EE Security Patterns: Architectural Patterns & Best Practices*. Upper Saddle River, NJ, USA: Prentice-Hall, 2005.

[13] L. DiPippo and C. D. Gill, *Design Patterns for Distributed Real-Time Systems*. Secaucus, NJ, USA: Springer-Verlag, 2005.

[14] B. P. Douglass, *Real Time Design Patterns: Robust Scalable Architecture for Real-time Systems*. Boston, MA, USA: Addison-Wesley, 2002.

[15] R. S. Hanmer, *Patterns for Fault Tolerant Software*. West Sussex, England: Wiley, 2007.

[16] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proc. 22nd Int. Conf. Softw. Eng.*, Orlando, Florida, USA, May 2002, pp. 338–348.

[17] D. Hou and H. J. Hoover, "Using SCL to specify and check design intent in source code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 404–423, Jun. 2006.

[18] N. N. Shi and R. Olsson, "Reverse engineering of design patterns from Java source code," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Tokyo, Japan, Sept. 2006, pp. 123–134.

[19] A. Blewitt, A. Bundy, and I. Stark, "Automatic verification of design patterns in Java," in *Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Long Beach, California, USA, Nov. 2005, pp. 224–232.

[20] D. Maplesden, J. Hosking, and J. Grundy, "Design pattern modelling and instantiation using DPML," in *Proc. 4th Int. Conf. Tools Pacific.*, Darlinghurst, Australia, 2002, pp. 3–11.

[21] J. Dong, Y. Zhao, and T. Peng, "Architecture and design pattern discovery techniques-a review," in *Proc. Int. Conf. Softw. Eng. Res. Practice*, Jun. 2007, pp. 621–627.

[22] D.-K. Kim and L. Lu, "Inference of design pattern instances in UML models via logic programming," in *Proc. 11th Int. Conf. Eng. Complex Comput. Syst.*, Stanford, California, USA, Aug. 2006, pp. 47–56.

[23] D.-K. Kim and W. Shen, "An approach to evaluating structural pattern conformance of UML models," in *Proc. ACM Symp. Applied Comput.*, Seoul, Korea, Mar. 2007, pp. 1404–1408.

[24] D.-K. Kim and W. Shen, "Evaluating pattern conformance of UML models: A divide-and-conquer approach and case studies," *Softw. Quality J.*, vol. 16, no. 3, pp. 329–359, 2008.

[25] H. Zhu, I. Bayley, L. Shan, and R. Amphlett, "Tool support for design pattern recognition at model level," in *Proc. 33rd Annu. IEEE Int. Comput. Softw. Appl. Conf.*, Seattle, Washington, USA, Jul. 2009, pp. 228–233.

[26] H. Zhu, L. Shan, I. Bayley, and R. Amphlett, "A formal descriptive semantics of UML and its applications," in *UML 2 Semantics and Applications*, K. Lano, Ed. New York, NY, USA: Wiley, Nov. 2009.

[27] F. Khomh and Y.-G. Guéhéneuc, "Do design patterns impact software quality positively?" in *Proc. 12th Eur. Conf. Softw. Maintenance Reeng.*, Athens, Greece, Apr. 1-4 2008, pp. 274–278.

[28] B. Venners. (2005, May). How to use design patterns: A conversation with Erich Gamma, Part I. [Online]. Available: http://www.artima.com/lejava/articles/gammadp.html

[29] S. M. Yacoub and H. H. Ammar, "UML support for designing software systems as a composition of design patterns," in *Proc. 4th Int. Conf. Unified Modeling Lang.-Modeling Lang., Concepts Tools*, Toronto, Canada, Oct. 2001, pp. 149–165.

[30] P. Wendorff, "Assessment of design patterns during software reengineering: Lessons learned from a large professional project," in *Proc. 5th Eur. Conf. Softw. Maintenance Reeng.*, Lisbon, Portugal, Mar. 2001, pp. 77–84.

[31] M. Mouratidou, V. Lourdas, A. Chatzigeorgiou, and C. K. Georgiadis, "An assessment of design patterns' influence on a Java-based E-commerce application," *J. Theoretical Applied Electron. Commerce Res.*, vol. 5, no. 1, pp. 25–38, Apr. 2010.

[32] H. Zhu and I. Bayley, "An algebra of design pattern composition," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, article 23, Jul. 2013.

[33] K. Lano, J. C. Bicarregui, and S. Goldsack, "Formalising design patterns," in *Proc. BCS-FACS Northern Formal Methods Workshop*, Ilkley, UK, Sep. 1996, p. 11.

[34] T. Mikkonen, "Formalizing design patterns," in *Proc. 20th Int. Conf. Softw. Eng.*., Apr. 1998, pp. 115–124.

[35] J. Dong, P. S. C. Alencar, and D. D. Cowan, "Correct composition of design components," in *Proc. 4th Int. Workshop Component-Oriented Programm. Conjunction ECOOP*, 1999, p. 186.

[36] A. Lauder and S. Kent, "Precise visual specification of design patterns," in *Proc. 12th Eur. Conf. Object-Oriented Programm.*, 1998, pp. 114–134.

[37] A. H. Eden, "Formal specification of object-oriented design," in *Proc. Int. Conf. Multidisciplinary Des. Eng.*, Montreal, Canada, Nov. 2001, pp. 21–22.

[38] T. Taibi, D. Check, and L. Ngo, "Formal specification of design patterns-a balanced approach," *J. Object Technol.*, vol. 2, no. 4, pp. 127–140, Jul.-Aug. 2003.

[39] I. Bayley and H. Zhu, "Formalising design patterns in predicate logic," in *Proc. 5th IEEE Int. Conf. Softw. Eng. Formal Methods*, London, UK, Sep. 2007, pp. 25–36.

[40] E. Gasparis, A. H. Eden, J. Nicholson, and R. Kazman, "The design navigator: Charting Java programs," in *Proc. Companion 30th Int. Conf. Softw. Eng.*, 2008, pp. 945–946.

[41] I. Bayley and H. Zhu, "Formal specification of the variants and behavioural features of design patterns," *J. Syst. Softw.*, vol. 83, no. 2, pp. 209–221, Feb. 2010.

[42] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture. Vol.5: On Patterns and Pattern Languages*. West Sussex, England: Wiley, 2007.

[43] I. Bayley and H. Zhu, "A formal language of pattern composition," in *Proc. 2nd Int. Conf. Pervasive Patterns*, Lisbon, Portugal, Nov. 2010, pp. 1–6.

[44] I. Bayley and H. Zhu, "A formal language for the expression of pattern compositions," *Int. J. Adv. Softw.*, vol. 4, no. 3&4, pp. 354–366, 2011.

[45] I. Bayley and H. Zhu, "On the composition of design patterns," in *Proc. 8th Int. Conf. Quality Softw.*, Oxford, UK, Aug. 2008, pp. 27–36.

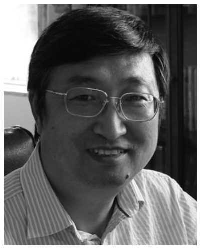
[46] A. H. Eden, *Codecharts: Roadmaps and Blueprints for Object-Oriented Programs*. Hoboken, NJ, USA: Wiley, 2011.

[47] A. H. Eden, E. Gasparis, J. Nicholson, and R. Kazman, "Modeling and visualizing object-oriented programs with codecharts," *Formal Methods Syst. Des.*, vol. 42, no. 1, p. 1–28, 2013.

[48] J. Nicholson, A. H. Eden, E. Gasparis, and R. Kazman, "Automated verification of design patterns: A case study," *Sci. Comput. Programm.*, vol. 80, no. B, p. 211–222, Feb. 2014.

[49] H. Zhu, "On the theoretical foundation of meta-modelling in graphically extended BNF and first order logic," in *Proc. 4th IEEE Symp. Theoretical Aspects Softw. Eng.*, Taipei, Taiwan, Aug. 2010, pp. 95–104.

[50] H. Zhu, "An institution theory of formal Meta-modelling in graphically extended BNF," *Frontiers Comput. Sci.*, vol. 6, no. 1, pp. 40–56, 2012.
[51] H. Zhu and I. Bayley, "Laws of pattern composition," in *Proc. 12th Int. Conf. Formal Eng. Methods*, Shanghai, China, Nov. 17-19 2010, pp. 630–645.
[52] I. Bayley and H. Zhu, "Specifying behavioural features of design patterns," Dept. of Comput., Oxford Brookes University, Oxford, UK, Tech. Rep. TR-08-01, 2008.
[53] P. Bottoni, E. Guerra, and J. de Lara, "Towards a formal notion of interaction pattern," in *Proc. IEEE Symp. Vis. Lang. Human-Centric Comput.*, Leganes, Spain, Sep. 2010, pp. 235–239.
[54] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture. Vol. 1: A System of Patterns*. West Sussex, England: Wiley, 1996.
[55] W. B. McNatt and J. M. Bieman, "Coupling of design patterns: Common practices and their benefits," in *Proc. 25th Comput. Softw. Appl. Conf.*, Oct. 2001, pp. 574–579.
[56] D. Riehle, "Composite design patterns," in *Proc. ACM SIGPLAN Conf. Object-Oriented Programm. Syst., Lang. Appl.*, Atlanta, Georgia, Oct. 5-9 1997, pp. 218–228.
[57] J. Vlissides, "Notation, notation, notation," *C++ Report*, pp. 48–51, Apr. 1998.
[58] J. Dong, S. Yang, and K. Zhang, "Visualizing design patterns in their applications and compositions," *IEEE Trans. Softw. Eng.*, vol. 33, no. 7, pp. 433–453, Jul. 2007.
[59] J. M. Smith, "The pattern instance notation: A simple hierarchical visual notation for the dynamic visualization and comprehension of software patterns," *J. Vis. Lang. Comput.*, vol. 22, no. 5, pp. 355–374, Oct. 2011.
[60] J. Dong, P. S. Alencar, and D. D. Cowan, "Ensuring structure and behavior correctness in design composition," in *Proc. IEEE 7th Annu. Int. Conf. Workshop Eng. Comput. Based Syst.*, Edinburgh, Scotland, Apr. 2000, pp. 279–287.
[61] T. Taibi, "Formalising design patterns composition," *IEE Proc. Softw.*, vol. 153, no. 3, pp. 126–153, Jun. 2006.
[62] J. Dong, T. Peng, and Y. Zhao, "Automated verification of security pattern compositions," *Inf. Softw. Technol.*, vol. 52, no. 3, p. 274–295, Mar. 2010.
[63] J. Dong, T. Peng, and Y. Zhao, "On instantiation and integration commutability of design pattern," *The Comput. J.*, vol. 54, no. 1, pp. 164–184, Jan. 2011.
[64] T. Taibi and D. C. L. Ngo, "Formal specification of design pattern combination using BPSL," *Inf. Softw. Technol.*, vol. 45, no. 3, pp. 157–170, Mar. 2003.
[65] B. Huston, "The effects of design pattern application on metric scores," *J. Syst. Softw.*, vol. 58, no. 3, pp. 261–269, Sep. 2001.
[66] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta, "A controlled experiment in maintenance: Comparing design patterns to simpler solutions," *IEEE Trans. Softw. Eng.*, vol. 27, no. 12, pp. 1134–1144, Dec. 2001.
[67] N.-L. Hsueh, P.-H. Chu, and W. Chu, "A quantitative approach for evaluating the quality of design patterns," *The J. Syst. Softw.*, vol. 81, pp. 1430–1439, 2008.
[68] A. Ampatzoglou, G. Frantzeskou, and I. Stamelos, "A methodology to assess the impact of design patterns on software quality," *Inf. Softw. Technol.*, vol. 54, no. 4, pp. 331–346, Apr. 2012.
[69] P. Bottoni, E. Guerra, and J. de Lara, "A language-independent and formal approach to pattern-based modelling with support for composition and analysis," *Inf. Softw. Technol.*, vol. 52, no. 8, pp. 821–844, 2010.
[70] N. Cacho, C. Santanna, E. Figueiredo, F. Dantas, A. Garcia, and T. Batista, "Blending design patterns with aspects: A quantitative study," *J. Syst. Softw.*, vol. 98, pp. 117–139, 2014.



Hong Zhu received the BSc, MSc, and PhD degrees in computer science from Nanjing University, China, in 1982, 1984, and 1987, respectively. He worked at Nanjing University as a lecturer, associate professor, and then a full professor from August 1987 to November 1998. From October 1990 to December 1994 while on leave from Nanjing University, he was a research fellow at Brunel University and the Open University, United Kingdom. He joined Oxford Brookes University, United Kindom, in November 1998 as a senior lecturer in computing and became a professor of computer science in October 2004. He chairs the Applied Formal Methods Research Group of the Department of Computing and Communication Technologies. His research interests are in the area of software development methodologies, including formal methods, agent-orientation, automated software development, foundation of software engineering, software design, modeling and testing methods, Software-as-a-Service, etc. He has published two books and more than 180 research papers in journals and international conferences. He has been a conference program committee chair of IEEE SOSE 2012 and IEEE ICWS 2015, etc., a conference general chair of IEEE SOSE 2013, IEEE MobileCloud 2014, etc. He is a member of the editorial board of the journal of *Software Testing, Verification and Reliability*, *Software Quality Journal*, *International Journal of Big Data Intelligence*, and the *International Journal of Multi-Agent and Grid Systems*. He is a senior member of the IEEE, IEEE Computer Society, a member of British Computer Society, ACM, and China Computer Federation.

Ian Bayley studied for the MEng degree in computing (mathematical foundations and formal methods) at Imperial College London and received the DPhil degree in computation at the Balliol College of Oxford University with research into the semantics of functional programming languages. Since 2005, he has been a lecturer at Oxford Brookes University. His research interests include software engineering, formal methods, and programming paradigms.