

Zhu, H and Bayley, I

On the Composability of Design Patterns

Zhu, H and Bayley, I (2015) On the Composability of Design Patterns. *IEEE Transactions on Software Engineering*, 41 (11). pp. 1138-1152.

This version is available: <https://radar.brookes.ac.uk/radar/items/259da36e-dfbd-4436-84a6-1bd3bfa86504/1/>

Available on RADAR: September 2016

Copyright © and Moral Rights are retained by the author(s) and/ or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This item cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder(s). The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

This document is the post print version of the journal article. Some differences between the published version and this version may remain and you are advised to consult the published version if you wish to cite from it.

On the Composability of Design Patterns

Hong Zhu, *Senior Member, IEEE*, and Ian Bayley
Oxford Brookes University, Oxford OX33 1HX, UK
Email: hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

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.

Keywords—Design Patterns, Pattern composition, Composability, Feature preservation, Semantics preservation, Soundness preservation, Formal methods.

I. 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]–[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]–[21] and at model level [22]–[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?

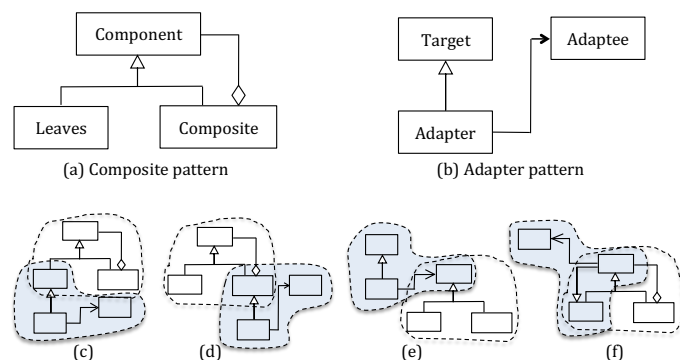


Fig. 1. Motivative examples of pattern compositions

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]–[41].

The remainder of the paper is organised as follows. Section II outlines our proposed approach and summarises the main contributions of this paper. Section III sets the foundation of the work by defining the mathematical notations and recalls the formal theory that the paper is based on. Section IV 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 V is devoted to the verification of the validity of pattern compositions and instantiations expressed in terms of pattern operations. Section VI applies the theory to overlap-based pattern composition operators. Section VII reports a case study with a real example of pattern-oriented software design: a general request handling framework [42]. Finally, Section VIII concludes the paper with a comparison with related work and a discussion of future work.

II. 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 (eg 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 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.

III. PRELIMINARIES

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

A. 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]–[48] but they

are limited to structural features. Taibi’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 $Vars = \{v_1 : T_1, \dots, v_n : T_n\}$ 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.

$$\exists v_1 : T_1 \dots \exists v_n : T_n \cdot (Pred) \quad (1)$$

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 $Pred$. \square

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 $\llbracket p \rrbracket_\alpha^m$, is defined as usual in predicate logic. Thus, the definition is omitted for the sake of space. If the result of the evaluation $\llbracket Pred(P) \rrbracket_\alpha^m$ 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 $m \models Spec(P)$. \square

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 $m \models q$. \square

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

$$(X \rightarrow Y) \wedge (Y \rightarrow Z) \implies X \rightarrow Z. \quad (2)$$

$$\neg(X \rightarrow Y \wedge Y \rightarrow X) \quad (3)$$

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].

B. 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 $P \preceq Q$, if for all models m , whenever m conforms to P , m also conforms to Q . Formally, $P \preceq Q \triangleq \forall m. (m \models P \Rightarrow m \models Q)$.

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

To establish that $P \preceq Q$, 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 , $m \models FALSE$. \square

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, $V = Vars(P) = \{x_0 : T_0, \dots, x_n : T_n\}$ and $Pred(P) = p(x_0, \dots, x_n)$.

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

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

$$Pred(P \Downarrow x_0 \backslash x'_0) = p(\{x'_0\}, x_1, \dots, x_n).$$

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

$$Vars(P \Uparrow x_0 \backslash x'_0) = \{x'_0 : \mathbb{P}(T_0), x_1 : T_1, \dots, x_n : T_n\},$$

$$Pred(P \Uparrow x_0 \backslash x'_0) = \forall x_0 \in x'_0. Pred(P).$$

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

$$Vars(P \Uparrow X) = \{xs_0 : \mathbb{P}(T_0), \dots, xs_n : \mathbb{P}(T_n)\},$$

$$Pred(P \Uparrow X) = \forall x_0 \in xs_0 \dots \forall x_k \in xs_k \cdot$$

$$\exists x_{k+1} \in xs_{k+1} \dots \exists x_n \in xs_n \cdot p(x_1, \dots, x_n).$$

\square

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 *Targets* 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 $P \Downarrow x \setminus x'$, and $P \Downarrow X$ to represent $P \Downarrow X \setminus X'$. 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.

$$\begin{aligned} P &= \langle \{A : \text{Class}\}, A.\text{isAbstract} \rangle \\ Q &= \langle \{B : \text{Class}, C : \text{Class}\}, B \rightarrow C \rangle \end{aligned}$$

where $X.\text{isAbstract}$ means that class X is an abstract class, $X \rightarrow Y$ means that class X is a subclass of Y . We have that

$$\begin{aligned} \text{Spec}(P) &= \exists A : \text{Class} \cdot (A.\text{isAbstract}) \\ \text{Spec}(Q) &= \exists B, C : \text{Class} \cdot (B \rightarrow C) \end{aligned}$$

Consider the pattern compositions R_1 to R_4 defined as follows:

$$\begin{aligned} R_1 &= P \# (D : \text{Class} \bullet D \diamond \rightarrow A) \\ R_2 &= Q \uparrow B \setminus B_s \\ R_3 &= Q \uparrow B \setminus B_s \\ R_4 &= (P * Q)[A = C] \end{aligned}$$

where $X \diamond \rightarrow Y$ 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 $B_s = \{B_1, \dots, B_n, \dots\}$ 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.

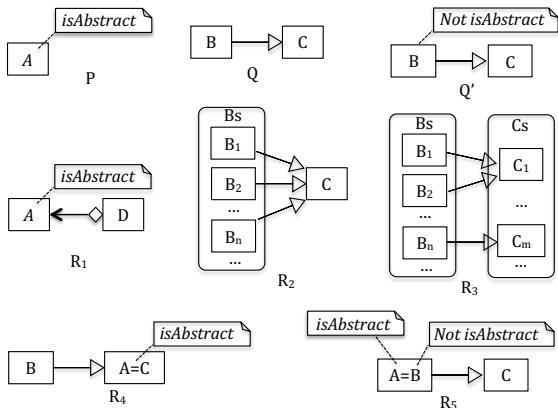


Fig. 2. Illustration of the Patterns in Examples 1 to 4

From the definitions of the operators we immediately have:

$$\begin{aligned} \text{Spec}(R_1) &= \exists A, D : \text{Class} \cdot (A.\text{isAbstract} \wedge D \diamond \rightarrow A) \\ \text{Spec}(R_2) &= \exists B_s : \mathbb{P}(\text{Class}), \exists C : \text{Class} \cdot \\ &\quad (\forall B \in B_s \cdot (B \rightarrow C)) \\ \text{Spec}(R_3) &= \exists B_s, C_s : \mathbb{P}(\text{Class}) \cdot \\ &\quad (\forall B \in B_s \cdot \exists C \in C_s \cdot (B \rightarrow C)) \\ \text{Spec}(R_4) &= \exists A, B : \text{Class} \cdot (A.\text{isAbstract} \wedge B \rightarrow A) \end{aligned}$$

□

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.

IV. 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.

A. 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,

$$\text{Spec}(P) \vdash p \Rightarrow \text{Spec}(\oplus P) \vdash p.$$

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,

$$(\text{Spec}(P) \vdash p) \vee (\text{Spec}(Q) \vdash p) \Rightarrow (\text{Spec}(P \oplus Q) \vdash p). \quad \square$$

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 , $\text{Spec}(\oplus P) \Rightarrow \text{Spec}(P)$.

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

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

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

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

That is, \oplus preserves features. \blacksquare

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 $Spec(P) \vdash p$. \square

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. $Thm_P \cup Thm_Q$. \square

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

$$\forall p \in Thm_P \cdot (Spec(P) \Rightarrow p), \quad (4)$$

$$\forall q \in Thm_Q \cdot (Spec(Q) \Rightarrow q). \quad (5)$$

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. \blacksquare

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

$$Spec(R_2) \Rightarrow Spec(Q)$$

$$Spec(R_3) \Rightarrow Spec(Q)$$

$$Spec(R_4) \Rightarrow Spec(P) \text{ and } Spec(R_4) \Rightarrow Spec(Q).$$

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. \square

B. 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 $\llbracket P \rrbracket$, is the set of models m that satisfy the specification. Formally,

$$\llbracket P \rrbracket \triangleq \{m \mid m \models Spec(P)\}. \quad \square$$

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

$$P \approx Q \Leftrightarrow \llbracket P \rrbracket = \llbracket Q \rrbracket, \quad (6)$$

$$P \preceq Q \Leftrightarrow \llbracket P \rrbracket \subseteq \llbracket Q \rrbracket. \quad (7)$$

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 $\llbracket P \rrbracket = \llbracket \oplus P \rrbracket$.

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

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

$$m \models Spec(R_4)$$

$$\Leftrightarrow m \models \exists A, B : Class \cdot (A.isAbstract \wedge B \rightarrow A)$$

$$\Rightarrow m \models \exists A : Class \cdot (A.isAbstract)$$

$$\wedge \exists A, B : Class \cdot (B \rightarrow A)$$

$$\Leftrightarrow m \models \exists A : Class \cdot (A.isAbstract) \wedge$$

$$m \models \exists A, B : Class \cdot (B \rightarrow A)$$

$$\Leftrightarrow m \models Spec(P) \wedge m \models Spec(Q)$$

Therefore, $\llbracket R_4 \rrbracket \subseteq \llbracket P \rrbracket \cap \llbracket Q \rrbracket$.

On the other hand, $\llbracket R_4 \rrbracket \neq \llbracket P \rrbracket \cap \llbracket Q \rrbracket$, because

$$\begin{aligned} \exists A : \text{Class} \cdot (A.\text{isAbstract}) \wedge \exists A, B : \text{Class} \cdot (B \rightarrow A) \\ \not\equiv \exists A, B : \text{Class} \cdot (A.\text{isAbstract} \wedge B \rightarrow A). \end{aligned}$$

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. \square

C. 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

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

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. \blacksquare

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

$$\{\{B, C : \text{Class}\}, (B \rightarrow C \wedge \neg B.\text{isAbstract})\}$$

Then, we have that

$$\text{Spec}(Q') = \exists B, C : \text{Class} \cdot (B \rightarrow C \wedge \neg B.\text{isAbstract})$$

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.

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

\square

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.

V. 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.

A. Feature Preservation Properties

Theorem 1: (Feature Preservation of Pattern Operators)

The restriction, extension, flattening, generalisation, superposition and lifting operators all preserve features. \square

Note that, for all patterns P and Q , if $\text{Spec}(P) \Rightarrow \text{Spec}(Q)$, we have that $P \preceq Q$. 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 $\text{Spec}(E) \Rightarrow \text{Spec}(P_i)$. This means that E preserves the features of P_i . \square

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.

B. 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 $X \subseteq \text{Vars}(P)$, we have that for all models m ,

$$(m \models P * Q) \Leftrightarrow ((m \models P) \wedge (m \models Q)), \quad (8)$$

$$(m \models (P \uparrow X)) \Leftrightarrow (m \models P), \quad (9)$$

$$(m \models (P \uparrow X)) \Leftrightarrow (m \models P). \quad (10)$$

\square

An immediate corollary is the following.

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

- 1) $\llbracket P * Q \rrbracket = \llbracket P \rrbracket \cap \llbracket Q \rrbracket$;
- 2) $\llbracket P \uparrow x \rrbracket = \llbracket P \rrbracket$, for all $x \in \text{Vars}(P)$;
- 3) $\llbracket P \uparrow x \rrbracket = \llbracket P \rrbracket$, for all $x \in \text{Vars}(P)$. \square

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 $\text{Vars}(P)$, and c a given predicate. We have that

$$\llbracket P[c] \rrbracket = \{m \mid m \in \llbracket P \rrbracket \wedge m \models c\}, \quad (11)$$

$$\llbracket P\#(V \bullet c) \rrbracket = \{m \mid m \in \llbracket P \rrbracket \wedge m \models \exists V \cdot c\}, \quad (12)$$

$$\llbracket P \Downarrow x \rrbracket = \{m \mid m \in \llbracket P \rrbracket \wedge m \models (|x| = 1)\}. \quad (13)$$

□

Note that Theorem 4 implies that $\llbracket P[c] \rrbracket \subseteq \llbracket P \rrbracket$, $\llbracket P\#(V \bullet c) \rrbracket \subseteq \llbracket P \rrbracket$, and $\llbracket P \Downarrow x \rrbracket \subseteq \llbracket P \rrbracket$. 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 $\llbracket E \rrbracket \subseteq \llbracket P_i \rrbracket$. □

C. Soundness Preservation Properties

While each of the six operators preserve features, some do not preserve soundness. For example, restriction does not because $P[\text{false}]$ 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 $\llbracket P \rrbracket \neq \emptyset$. But $\llbracket P[\text{false}] \rrbracket = \emptyset$ because $P[\text{false}] \approx \text{FALSE}$.

(2) *Extension.* Suppose $\llbracket P \rrbracket \neq \emptyset$ again. But $\llbracket P\#(V \bullet \text{false}) \rrbracket = \emptyset$ because $P\#(V \bullet \text{false}) \approx \text{FALSE}$.

(3) *Flattening.* Suppose $P = \{\{v : \mathbb{P}(\text{Class})\}, \|\{v\| \geq 2\}\}$. Then designs exist that satisfy P so $\llbracket P \rrbracket \neq \emptyset$. However, from the definition of the flatten operator, $P \Downarrow v = \{\{v' : \text{Class}\}, (\|\{v'\| \geq 2)\}\}$. But $\|\{v'\| \geq 2\}$ is not satisfiable so $\llbracket P \Downarrow v \rrbracket = \emptyset$. □

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 $\text{Pred}(P) \Rightarrow \neg c$.
- 2) $P\#(V \bullet c)$ is not sound if $\text{Pred}(P) \Rightarrow \neg \exists V \cdot c$.
- 3) $P \Downarrow x$ is not sound if $\text{Pred}(P) \Rightarrow (\|\{x\| \neq 1)$. □

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 $c \neq \text{false}$ implies that there is a model m such that $m \models c$, so these conditions are the strongest that one can get.

D. 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.

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

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 $\text{Composite} \diamond \rightarrow$

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)

$$\begin{aligned} \llbracket (c) \rrbracket &= \{m \mid m \in \llbracket \text{Composite} * \text{Adapter} \rrbracket \\ &\quad \wedge m \models (\text{Leaves} = \text{Target})\} \\ &\subseteq \llbracket \text{Composite} * \text{Adapter} \rrbracket \\ &= \llbracket \text{Composite} \rrbracket \cap \llbracket \text{Adapter} \rrbracket \end{aligned}$$

As $\llbracket \text{Composite} \rrbracket \neq \llbracket \text{Adapter} \rrbracket$, we have $\llbracket (c) \rrbracket \subset \llbracket \text{Adapter} \rrbracket$ and $\llbracket (c) \rrbracket \subset \llbracket \text{Composite} \rrbracket$. 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

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

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

$$\begin{aligned} \{m \mid m \in \llbracket \text{Composite} * \text{Adapter} \rrbracket \\ \wedge m \models \text{Leaves} = \text{Target} \wedge \text{Component} = \text{Adapter}\}. \end{aligned}$$

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

$$\begin{aligned} \text{Pred}(\text{Composite}) &\Rightarrow \text{Leaves} \rightarrow \text{Component}, \\ \text{Pred}(\text{Adapter}) &\Rightarrow \text{Adapter} \rightarrow \text{Target}, \end{aligned}$$

we have that $\text{Leaves} \rightarrow \text{Adapter}$ and $\text{Adapter} \rightarrow \text{Leaves}$. This contradicts the axioms about inheritance relation between classes, i.e. Equ. (3). So we have

$$\begin{aligned} \text{Pred}(\text{Composite} * \text{Adapter}) &\Rightarrow \\ &\neg(\text{Leaves} = \text{Target} \wedge \text{Component} = \text{Adapter}). \end{aligned}$$

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 SPASS¹. The details of using SPASS in the proof of the example above is given in online Appendix 3.

In conclusion, the validity of a pattern composition can be determined as follows. First, represent it using the six pattern

¹<http://www.spass-prover.org>

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.

VI. 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.

A. 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 $\llbracket \text{Pred}(P) \rrbracket_{\alpha_1}^m = \text{true}$ and $\llbracket \text{Pred}(Q) \rrbracket_{\alpha_2}^m = \text{true}$. There is an *overlap* between two assignments if there is an element of the model m assigned to two variables, one in $\text{Vars}(P)$ and the other in $\text{Vars}(Q)$. 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 $v \in \text{Vars}(P)$ and $u \in \text{Vars}(Q)$ 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:

$$P\langle v - u \rangle Q \triangleq (P * Q)[v = u]. \quad \square$$

Definition 13: (Composition with Many-to-Many Overlap)

Let P and Q be design patterns. Let $vs \in \text{Vars}(P)$ and $us \in \text{Vars}(Q)$ 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 $vs \succ us$* , written $P\langle vs \succ us \rangle Q$, is defined as follows:

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

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 $=$.

$$P\langle vs \succ_R us \rangle Q \triangleq (P * Q)[vs R us].$$

Theorem 5: (Ordering among Many-to-Many Compositions)

For all patterns P and Q , we have that

$$(P\langle vs \succ_C us \rangle Q) \preceq (P\langle vs \succ_{\subseteq} us \rangle Q) \quad (14)$$

$$(P\langle vs \succ_{=} us \rangle Q) \preceq (P\langle vs \succ_{\subseteq} us \rangle Q) \quad (15)$$

$$(P\langle vs \succ_{\subseteq} us \rangle Q) \preceq (P\langle vs \succ us \rangle Q) \quad (16)$$

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 $v \in \text{Vars}(P)$ be a variable assigned to a model element and let $us \in \text{Vars}(Q)$ 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 $v \prec us$* , written $P\langle v \prec us \rangle Q$, is defined as follows:

$$P\langle v \prec us \rangle Q \triangleq (P * Q)[v \in us] \quad \square$$

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 $v \in \text{Vars}(P)$ be a variable assigned to a model element and let $us \in \text{Vars}(Q)$ 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 $v \prec us$* is defined as follows:

$$P\langle v_{\uparrow} \prec_{\subseteq} us \rangle Q \triangleq (P \uparrow (v \setminus vs) * Q)[vs \subseteq us] \quad \square$$

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.

$$P\langle v_{\uparrow} \prec_C us \rangle Q \triangleq (P \uparrow (v \setminus vs) * Q)[vs \subset us]$$

$$P\langle v_{\uparrow} \prec_{=} us \rangle Q \triangleq (P \uparrow (v \setminus vs) * Q)[vs = us]$$

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

$$P\langle v_{\uparrow} \prec_C us \rangle Q \triangleq (P \uparrow (v \setminus vs) * Q)[vs \subset us]$$

$$P\langle v_{\uparrow} \prec_{=} us \rangle Q \triangleq (P \uparrow (v \setminus vs) * Q)[vs = us]$$

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

$$P\langle v_{\uparrow} \prec_R us \rangle Q \preceq P\langle v_{\uparrow} \prec_{R'} us \rangle Q,$$

$$P\langle v_{\uparrow} \prec_{\subseteq} us \rangle Q \preceq P\langle v \prec us \rangle Q.$$

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 $P\langle v_{\uparrow} \prec_R us \rangle Q$ for R to be \subseteq , \subset and $=$, and among $P\langle v_{\uparrow} \prec_R us \rangle Q$.

The above \preceq 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 $A \preceq B$. On the right-hand side of Fig. 3 are the ordering relations given in Theorem 5. On the left-hand side are the \preceq relationships between the one-to-many overlap composition operators.

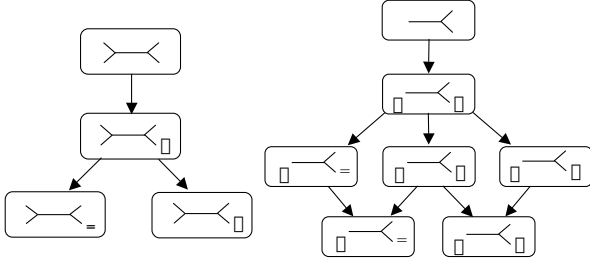


Fig. 3. Relationships Between Compositions with Overlaps

B. 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 $P\langle v \text{---} u \rangle Q$, we have that

- 1) it preserves features;
- 2) its semantics $\llbracket P\langle v \text{---} u \rangle Q \rrbracket$ is:

$$\{m \mid m \in \llbracket P \rrbracket \cap \llbracket Q \rrbracket \wedge m \models (v = u)\};$$

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

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

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

$$\{m \mid m \in \llbracket P \rrbracket \cap \llbracket Q \rrbracket \wedge m \models (vs R us)\};$$

- 3) it loses soundness, if

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

where $(vs R us)$ 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 $P\langle v \dagger \text{---}_R us \rangle Q$, 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

$$\{m \mid m \in \llbracket P \rrbracket \cap \llbracket Q \rrbracket \wedge m \models (vs R us)\};$$

- 3) it loses soundness, if

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

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 subsection, a one-to-one overlap composition $P\langle v \text{---} u \rangle Q$ can be expressed with the pattern operators as follows.

$$P\langle v \text{---} u \rangle Q = (P * Q)[v = u] \quad (\text{Def. 12})$$

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

For 2), we have that

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

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

VII. 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 I summarises the five design decisions that result in the design depicted in Fig. 4.² 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.

A. 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.

²Note 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].

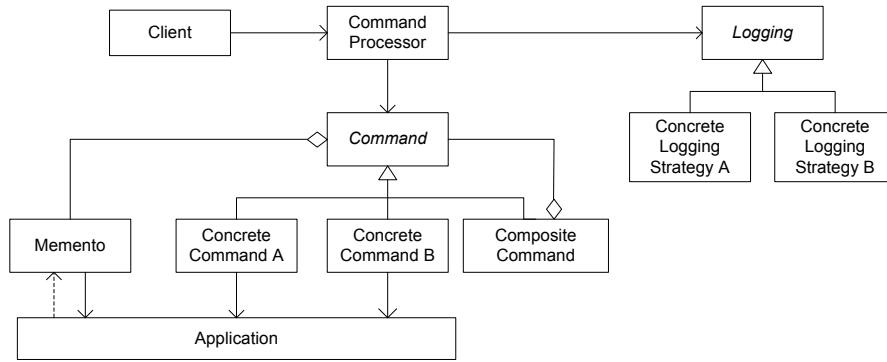


Fig. 4. Design of Request Handling Framework as Derived from the Formal Definition of RHF

TABLE I
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 Command Processor pattern to provide such coordination. The clients pass concrete commands to a CommandProcessor component for further handling and execution. It is inserted inbetween 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. <i>The Caretaker component creates a memento</i> , 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.

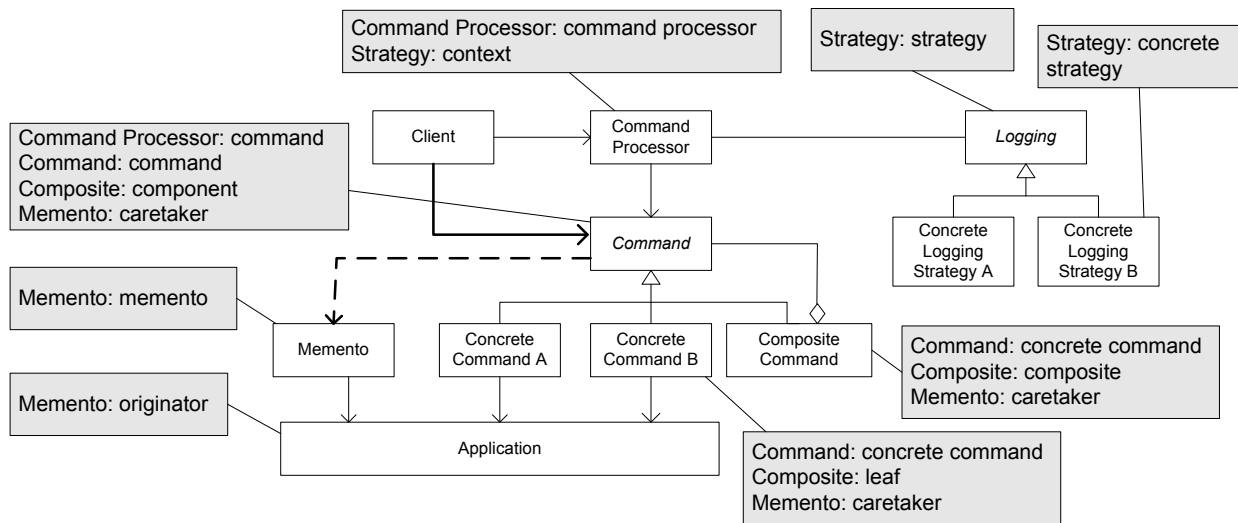


Fig. 5. Original Design of Request Handling Framework as in [42]

B. Semantics Preservation

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

$$m \in \llbracket RHF_O \rrbracket \implies m \models (Client \rightarrow Command).$$

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 I can be formally expressed using pattern operators as follows, where RHF is the final result.

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

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

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

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

C. Soundness Preservation

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

$$RHF \approx (Command * CommandProcessor * Memento * Strategy * Composite)[Connection]$$

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.

$$Pred(Command * CommandProcessor * Memento * Strategy * Composite) \implies \neg Connection.$$

Using the theorem prover SPASS, we can show this is not true; see online Appendix 3 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.

VIII. 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.

A. 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.

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, \dots, P_n , Dong et al. define a composition mapping $C : Vars(P_1) \times \dots \times Vars(P_n) \rightarrow Vars(P)$ that associates names of component pattern P_i to those of P , which is mathematically equivalent to a set of *name mappings* $C_i : Vars(P_i) \rightarrow Vars(P)$, $i = 1, \dots, n$.

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

$$Subst\{v_1 \setminus t_1, \dots, v_n \setminus t_n\}(\varphi_1 \wedge \varphi_2)$$

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.

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 $C = \langle C_1, \dots, C_n \rangle$ of mappings from patterns P_1, \dots, 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 com-

position, 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 [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.

B. 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.

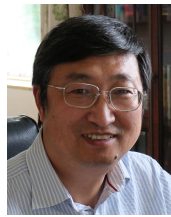
The composition of OO design patterns has also been studied in the context of aspect-oriented programming (AOP) [70], in which overlap based compositions can be implemented by employing a crosscutting mechanism. Cacho et al. demonstrated in an empirical study that under certain conditions such blending 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," *Communications of the ACM*, vol. 35, no. 9, pp. 152–159, Sept. 1992.
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Boston, USA: Addison-Wesley, 1995.
- [3] M. Grand, *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1*. New York, USA: John Wiley & Sons, 2002.
- [4] —, *Patterns in Java, volume 2*. New York, USA: John Wiley & Sons, 1999.
- [5] —, *Java Enterprise Design Patterns*. New York, USA: John Wiley & Sons, 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, June 2003.
- [7] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, USA: Addison Wesley, 2003.
- [8] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, 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: John Wiley & Sons, 2007.
- [10] M. Voelter, M. Kircher, and U. Zdun, *Remoting Patterns*. West Sussex, England: John Wiley & Sons, 2004.
- [11] M. Schumacher, E. Fernandez, D. Hybertson, and F. Buschmann, *Security Patterns: Integrating Security and Systems Engineering*. West Sussex, England: John Wiley & Sons, 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, 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 *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2002)*. Orlando, Florida, USA: IEEE CS, May 2002, pp. 338–348.
- [17] D. Hou and H. J. Hoover, "Using SCL to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, Jun. 2006.
- [18] N. Nija Shi and R. Olsson, "Reverse engineering of design patterns from Java source code," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. Tokyo, Japan: IEEE Computer Society, Sept. 2006, pp. 123–134.
- [19] A. Blewitt, A. Bundy, and I. Stark, "Automatic verification of design patterns in Java," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*. Long Beach, California, USA: ACM Press, Nov. 2005, pp. 224–232.
- [20] D. Maplesden, J. Hosking, and J. Grundy, "Design pattern modelling and instantiation using DPML," in *Proceedings of the 4th International Conference on Tools Pacific (TOOLS Pacific 2002)*. Darlinghurst, Australia: Australian Computer Society, 2002, pp. 3–11.
- [21] J. Dong, Y. Zhao, and T. Peng, "Architecture and design pattern discovery techniques - a review," in *Proceedings of the 2007 International Conference on Software Engineering Research and Practice (SERP 2007)*, H. R. Arabia and H. Reza, Eds., vol. II. Las Vegas Nevada, USA: CSREA Press, Jun. 2007, pp. 621–627.
- [22] D.-K. Kim and L. Lu, "Inference of design pattern instances in UML models via logic programming," in *Proceedings of the 11th International Conference on Engineering of Complex Computer Systems (ICECCS 2006)*. Stanford, California, USA: IEEE Computer Society, Aug. 2006, pp. 47–56.
- [23] D.-K. Kim and W. Shen, "An approach to evaluating structural pattern conformance of UML models," in *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07)*. Seoul, Korea: ACM Press, Mar. 2007, pp. 1404–1408.
- [24] —, "Evaluating pattern conformance of UML models: a divide-and-conquer approach and case studies," *Software Quality Journal*, 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. of COMPAC'09*. Seattle, Washington, USA: IEEE Computer Society, 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. John Wiley & Sons, Inc., Nov. 2009.
- [27] F. Khomh and Y.-G. Guéhéneuc, "Do design patterns impact software quality positively?" in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*. Athens, Greece: IEEE, Apr. 1–4 2008, pp. 274–278.
- [28] B. Venners, "How to use design patterns: A conversation with Erich Gamma, Part I," [Online] <http://www.artima.com/lejava/articles/gammadp.html>, May 2005.
- [29] S. M. Yacoub and H. H. Ammar, "UML support for designing software systems as a composition of design patterns," in *Proceedings of the 4th International Conference on The Unified Modeling Language – Modeling Languages, Concepts, and Tools (UML 2001), Lecture Notes in Computer Science, Vol. 2185*, 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 *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001)*, 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," *Journal of Theoretical and Applied Electronic Commerce Research*, vol. 5, no. 1, pp. 25–38, Apr. 2010.
- [32] H. Zhu and I. Bayley, "An algebra of design pattern composition," *ACM Transactions on Software Engineering and Methodology (ACM TOSEM)*, vol. 22, no. 3, p. Article 23, Jul. 2013.
- [33] K. Lano, J. C. Bicarregui, and S. Goldsack, "Formalising design patterns," in *BCS-FACS Northern Formal Methods Workshop*, Ilkley, UK, Sept. 1996.
- [34] T. Mikkonen, "Formalizing design patterns," in *Proc. of ICSE'98, Kyoto, Japan*. IEEE CS, Apr. 1998, pp. 115–124.
- [35] J. Dong, P. S. C. Alencar, and D. D. Cowan, "Correct composition of design components," in *Proceedings of the 4th International Workshop on Component-Oriented Programming in conjunction with ECOOP'99*, 1999.
- [36] A. Lauder and S. Kent, "Precise visual specification of design patterns," in *Proc. of ECOOP'98*, Lecture Notes in Computer Science Vol. 1445. Springer, 1998, pp. 114–134.
- [37] A. H. Eden, "Formal specification of object-oriented design," in *International Conference on Multidisciplinary Design in Engineering, Montreal, Canada*, Nov. 2001.
- [38] T. Taibi, D. Check, and L. Ngo, "Formal specification of design patterns—a balanced approach," *Journal of Object Technology*, vol. 2, no. 4, Jul.-Aug. 2003.
- [39] I. Bayley and H. Zhu, "Formalising design patterns in predicate logic," in *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*. London, UK: IEEE Computer Society, Sept. 2007, pp. 25–36.
- [40] E. Gasparis, A. H. Eden, J. Nicholson, and R. Kazman, "The design navigator: charting Java programs," in *Proc. of ICSE'08*, Companion Volume, 2008, pp. 945–946.
- [41] I. Bayley and H. Zhu, "Formal specification of the variants and behavioural features of design patterns," *Journal of Systems and Software*, 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: John Wiley & Sons, 2007.
- [43] I. Bayley and H. Zhu, "A formal language of pattern composition," in *Proceedings of The 2nd International Conference on Pervasive Patterns (PATTERNS 2010)*. Lisbon, Portugal: Xpert Publishing Services, Nov. 2010, pp. 1–6.
- [44] —, "A formal language for the expression of pattern compositions," *International Journal on Advances in Software*, vol. 4, no. 3&4, pp. 354–366, 2011.
- [45] —, "On the composition of design patterns," in *Proceedings of the 8th International Conference on Quality Software (QSIC 2008)*. Oxford, UK: IEEE Computer Society, Aug. 2008, pp. 27–36.
- [46] A. H. Eden, *Codecharts: Roadmaps and Blueprints for Object-Oriented Programs*. Hoboken, New Jersey: Wiley-Blackwell, 2011.
- [47] A. H. Eden, E. Gasparis, J. Nicholson, and R. Kazman, "Modeling and visualizing object-oriented programs with codecharts," *Formal Methods in System Design*, 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," *Science of Computer Programming*, 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 *Proceedings of the 4th IEEE Symposium on Theoretical Aspects of Software Engineering (TASE 2010)*. Taipei, Taiwan: IEEE CS, August 2010, pp. 95–104.
- [50] —, "An institution theory of formal meta-modelling in graphically extended BNF," *Frontiers of Computer Science*, vol. 6, no. 1, pp. 40–56, 2012.
- [51] H. Zhu and I. Bayley, "Laws of pattern composition," in *Proceedings of 12th International Conference on Formal Engineering Methods (ICFEM 2010)*, LNCS, vol. 6447. Shanghai, China: Springer, Nov. 17-19 2010, pp. 630–645.
- [52] I. Bayley and H. Zhu, "Specifying behavioural features of design patterns," Department of Computing, 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 *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2010)*, Leganes, Spain, Sept. 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: John Wiley & Sons, 1996.
- [55] W. B. McNatt and J. M. Bieman, "Coupling of design patterns: Common practices and their benefits," in *Proceedings of the 25th Computer Software and Applications Conference (COMPSAC 2001)*. IEEE Computer Society Press, Oct. 2001, pp. 574 – 579.
- [56] D. Riehle, "Composite design patterns," in *Proceedings of the 1997 ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. Atlanta, Georgia: ACM Press, Oct. 5-9 1997, pp. 218–228.
- [57] J. Vlissides, "Notation, notation, notation," *C++ Report*, April 1998.
- [58] J. Dong, S. Yang, and K. Zhang, "Visualizing design patterns in their applications and compositions," *IEEE Transactions on Software Engineering*, vol. 33, no. 7, pp. 433–453, July 2007.
- [59] J. M. Smith, "The pattern instance notation: A simple hierarchical visual notation for the dynamic visualization and comprehension of software patterns," *Journal of Visual Languages and Computing*, 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 *Proceedings of the IEEE 7th Annual International Conference and Workshop on Engineering Computer Based Systems (ECBS 2000)*. Edinburgh, Scotland: IEEE CS Press, Apr. 2000, pp. 279–287.
- [61] T. Taïbi, "Formalising design patterns composition," *Software, IEE Proceedings*, vol. 153, no. 3, pp. 126–153, Jun. 2006.
- [62] J. Dong, T. Peng, and Y. Zhao, "Automated verification of security pattern compositions," *Information and Software Technology*, vol. 52, no. 3, p. 274–295, Mar. 2010.
- [63] —, "On instantiation and integration commutability of design pattern," *The Computer Journal*, vol. 54, no. 1, pp. 164–184, Jan. 2011.
- [64] T. Taïbi and D. C. L. Ngo, "Formal specification of design pattern combination using BPSL," *Information and Software Technology*, vol. 45, no. 3, pp. 157–170, Mar. 2003.
- [65] B. Huston, "The effects of design pattern application on metric scores," *Journal of Systems and Software*, vol. 58, no. 3, pp. 261–269, Sept. 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 Transactions on Software Engineering*, vol. 27, no. 12, pp. 1134 – 1144, 2001.
- [67] N.-L. Hsueh, P.-H. Chu, and W. Chu, "A quantitative approach for evaluating the quality of design patterns," *The Journal of Systems and Software*, 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," *Information and Software Technology*, 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," *Information and Software Technology*, 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," *Journal of Systems and Software*, vol. 98, pp. 117 – 139, 2014.



Dr. Hong Zhu obtained his 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 Full Professor from August 1987 to November 1998. From October 1990 to December 1994 while on leave from Nanjing University, Dr. Zhu was a research fellow at Brunel University and the Open University, UK. He joined Oxford Brookes University, UK, in November 1998 as a Senior Lecturer in Computing and became a Professor of Computer Science in October 2004. Prof. Zhu chairs the Applied Formal Methods Research Group of the Department of Computing and Communication Technologies. He is a senior member of IEEE Computer Society, a member of British Computer Society, ACM, and China Computer Federation. 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. Prof. Zhu has published 2 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.



Dr. Ian Bayley studied for an MEng Computing (Mathematical Foundations and Formal Methods) at Imperial College London and earned a DPhil Computation at 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.

APPENDIX 1: LAWS OF PATTERN OPERATIONS

In [32], we proved a complete set of the equational laws that the operators obey. Some of them are used in the proof of theorems and in the case study. For the sake of self-containedness, these laws are listed here in Fig. 6, where the following notations are used.

Let P be any given pattern, $X = \{x_1 : T_1, \dots, x_n : T_n\}$ be any set of variables, and p be any predicate.

- $X_P = X \cap \text{Vars}(P)$;
- $X^\uparrow = \{xs_1 : \mathbb{P}(T_1), \dots, xs_n : \mathbb{P}(T_n)\}$,
- $\exists X \cdot p$ denotes $\exists x_1 : T_1 \dots \exists x_n : T_n \cdot p$,
- $\forall X \cdot p$ denotes $\forall x_1 : T_1 \dots \forall x_n : T_n \cdot p$,
- $\exists X \in Xs \cdot p$ denotes $\exists x_1 \in xs_1 \dots \exists x_n \in xs_n \cdot p$,
- $\forall X \in Xs \cdot p$ denotes $\forall x_1 \in xs_1 \dots \forall x_n \in xs_n \cdot p$,
- $\text{vars}(p)$ denotes the set of free variables in predicate p ,
- Let $X' = (X \cap \text{vars}(p)) \cup (X - \text{vars}(p))$, and $Y = (\text{Vars}(P) \cup \text{vars}(p)) - X'$. Then,

$$\begin{aligned} p_{\uparrow X} &= \forall X \in X^\uparrow \cdot p \\ p_{\downarrow X} &= p(\{x'_1\}, \dots, \{x'_n\}, x_{n+1}, \dots, x_k), \\ p_{(P \uparrow X)} &= \forall X' \in X'^\uparrow \cdot \exists Y \in Y^\uparrow \cdot (\text{Pred}(P) \wedge p). \end{aligned}$$

APPENDIX 2: PROOFS OF THEOREMS

This Appendix gives the proofs of the theorems.

Proof: of Theorem 1

In each case, we prove $\text{Spec}(\oplus(P)) \Rightarrow \text{Spec}(P)$ by using Definition 5, which defines the operators, and then use Lemma 1 to conclude that the operator is feature preserving. We give proofs for just three of the operators. The other three proofs are similar. Let $X = \{x_1 : T_1, \dots, x_n : T_n\}$ in each case.

(1) *Restriction.* Let $\text{Vars}(P) = X$. By the definition of the operator,

$$\begin{aligned} \text{Spec}(P[c]) &= \exists X \cdot (\text{Pred}(P) \wedge c) \\ &\Rightarrow \exists X \cdot (\text{Pred}(P)) = \text{Spec}(P). \end{aligned}$$

(2) *Extension.* Let $V = \{y_1 : T'_1, \dots, y_k : T'_k\}$. By the definition of the operator, and since $V \cap X = \emptyset$, we have that

$$\begin{aligned} \text{Spec}(P \# (V \bullet c)) &= \exists X \exists V \cdot (\text{Pred}(P) \wedge c) \\ &\Rightarrow \exists X \exists V \cdot (\text{Pred}(P)) \Rightarrow \exists X \cdot (\text{Pred}(P)) = \text{Spec}(P) \end{aligned}$$

(3) *Flattening.* We first consider $P \downarrow x \setminus x'$ where $x : \mathbb{P}(T)$. Let $\text{Vars}(P) = \{x : \mathbb{P}(T)\} \cup X$ and $\text{Pred}(P) = p(x, x_1, \dots, x_n)$. We have that

$$\begin{aligned} \text{Spec}(P \downarrow x \setminus x') &= \exists x' : T \cdot \exists X \cdot (p(\{x'\}, x_1, \dots, x_n)) \\ &\Rightarrow \exists x : \mathbb{P}(T) \cdot \exists X \cdot (p(x, x_1, \dots, x_n)) = \text{Spec}(P) \end{aligned}$$

We now extend this by mathematical induction to all finite sets of variables.

Assume that the flatten operator has the following property for all variable sets X of size $k > 0$.

$$\text{Spec}(P \downarrow X) \Rightarrow \text{Spec}(P). \quad (69)$$

Then, for all variable sets X' of size $k + 1$, $X' = \{x\} \cup X$, where X is of size k . By algebraic law (45) in Fig. 6, the following equality is true.

$$P \downarrow (\{x\} \cup X) \approx (P \downarrow x) \downarrow X.$$

Therefore, we have that

$$\begin{aligned} \text{Spec}(P \downarrow X') &= \text{Spec}(P \downarrow (\{x\} \cup X)) \\ &= \text{Spec}((P \downarrow x) \downarrow X) \\ &\Rightarrow \text{Spec}(P \downarrow x) \Rightarrow \text{Spec}(P) \end{aligned}$$

Therefore, for all variable sets of size $k + 1$, we also have the property (69).

So by mathematical induction, 69 is true for finite sets of variables X and by Lemma 1, the operator \downarrow is feature-preserving. ■

Proof: of Theorem 2

The theorem follows by induction on the structure of expression E .

For the base case, $E = P_i$ for some i such that P_i is the only constituent pattern so $\text{Spec}(E) \Rightarrow \text{Spec}(P_i)$.

Suppose, as the induction hypothesis, that for some expression E' we have for each i that $\text{Spec}(E') \Rightarrow \text{Spec}(P_i)$.

Then, for any of the five unary operators \oplus , we have that $\text{Spec}(\oplus E') \Rightarrow \text{Spec}(E')$ because of their feature preservation properties. And, by transitivity of \Rightarrow , for each i , the statement $\text{Spec}(\oplus E') \Rightarrow \text{Spec}(P_i)$ follows from the induction hypothesis. The argument for a binary operator \oplus is similar. ■

Proof: of Theorem 3

(1) *Superposition.*

By definition, $m \models (P * Q)$ if and only if there is an assignment α from $\text{Vars}(P * Q)$ to elements in m such that $\llbracket \text{Pred}(P * Q) \rrbracket_\alpha^m$ is true.

By Definition 5, $\text{Pred}(P * Q) = \text{Pred}(P) \wedge \text{Pred}(Q)$. In predicate logic, we have that $\llbracket \text{Pred}(P) \wedge \text{Pred}(Q) \rrbracket_\alpha^m = \text{true}$ if and only if $\llbracket \text{Pred}(P) \rrbracket_\alpha^m = \text{true}$ and $\llbracket \text{Pred}(Q) \rrbracket_\alpha^m = \text{true}$.

Because $\text{Vars}(P * Q) = \text{Vars}(P) \cup \text{Vars}(Q)$, let α_P and α_Q be the assignments obtained by restricting α on $\text{Vars}(P)$ and $\text{Vars}(Q)$, respectively. Then, we have that $\llbracket \text{Pred}(P) \rrbracket_\alpha^m = \text{true}$ is true if and only if $\llbracket \text{Pred}(P) \rrbracket_{\alpha_P}^m = \text{true}$.

By definition, we have that $\llbracket \text{Pred}(P * Q) \rrbracket_{\alpha_P}^m = \text{true}$ if and only if $m \models P$. Similarly, we have that $\llbracket \text{Pred}(Q) \rrbracket_\alpha^m = \text{true}$ if and only if $m \models Q$.

Therefore, the theorem holds for the superposition operator.

(2) *Lifting.*

Without loss of generality, we assume that $\text{Vars}(P) = \{x_1 : T_1, \dots, x_n : T_n\}$ and $X = \{x_1 : T_1, \dots, x_k : T_k\}$ with $n > k > 0$, and $\alpha(x_i) = a_i$ for each i . Let $\text{Pred}(P) = p(x_1, \dots, x_n)$, and $Y = \text{Vars}(P) - X$.

(\Rightarrow) Let $m \models P$. By Definition 2, there is an assignment α such that

$$\llbracket \text{Pred}(P) \rrbracket_\alpha^m = \text{true}. \quad (70)$$

We have that Equ (70) means that $p(a_1, a_2, \dots, a_n)$ is true in model m .

Define $\alpha'(xs_i) = \{a_i\}$, for $i \in 1 \dots n$. We also have that $xs_i \neq \emptyset$ for all $i \in 1 \dots k$. Then the following predicate must be true under assignment α' .

$$\forall X \in X^\uparrow \cdot \exists Y \in Y^\uparrow \cdot p(x_1, x_2, \dots, x_n)$$

In other words, $\llbracket \text{Pred}(P \uparrow V) \rrbracket_{\alpha'}^m = \text{true}$. Therefore, we have that $m \models P \uparrow V$.

1. Laws of \preceq :		5. Laws of \Downarrow and \Uparrow :	
	$P \preceq P$ (17)	$(P \Downarrow \emptyset) \approx P$ (43)	
	$(P \preceq Q) \wedge (Q \preceq R) \Rightarrow (P \preceq R)$ (18)	$(P \Downarrow X) \Downarrow Y \approx (P \Downarrow Y) \Downarrow X$ (44)	
	$P \preceq Q \wedge Q \preceq P \Rightarrow P \approx Q$ (19)	$(P \Downarrow X) \Downarrow Y \approx P \Downarrow (X \cup Y)$ (45)	
	$FALSE \preceq P \preceq TRUE$ (20)	$(P \Uparrow \emptyset) \approx P$ (46)	
2. Laws of $[c]$:		$(P \Uparrow X) \Uparrow Y \approx (P \Uparrow Y) \Uparrow X$ (47)	
	$(c_1 \Rightarrow c_2) \Rightarrow P[c_1] \preceq P[c_2]$ (21)	$(P \Uparrow X) \Uparrow Y \approx P \Uparrow (X \cup Y)$ (48)	
	$P[c][c] \approx P[c]$ (22)	6. Laws connecting $*$ with others:	
	$P[c_1][c_2] \approx P[c_2][c_1]$ (23)	$P[c] * Q \approx (P * Q)[c]$ (49)	
	$P[c_1][c_2] \approx P[c_1 \wedge c_2]$ (24)	$(P \Uparrow X) * Q \approx (P * Q) \Uparrow X$ (50)	
	$P[true] \approx P$ (25)	$(P \Downarrow X) * Q \approx (P * Q) \Downarrow X$ (51)	
	$P[false] \approx FALSE$ (26)	$(P * Q) \Uparrow X \approx (P \Uparrow X_P) * (Q \Uparrow X_Q)$ (52)	
3. Laws of $*$:		$(P * Q) \Downarrow X \approx (P \Downarrow X_P) * (Q \Downarrow X_Q)$ (53)	
	$(P * Q) \preceq P$ (27)	$(P \Uparrow X) * Q \approx ((P * Q) \Uparrow X) \Downarrow Vars(Q)$ (54)	
	$Q \preceq P \Rightarrow P * Q \approx Q$ (28)	7. Laws connecting \Uparrow , \Downarrow and \uparrow :	
	$P * P \approx P$ (29)	$(P \Uparrow X \setminus U) \Downarrow (U \setminus X) \approx P$ (55)	
	$P * TRUE \approx TRUE * P \approx P$ (30)	$(P \Downarrow X \setminus U) \Uparrow U \setminus X \approx P$ (56)	
	$P * FALSE \approx FALSE * P \approx FALSE$ (31)	$(P \Uparrow x) \Downarrow (Vars(P \Uparrow x)) \approx P$ (57)	
	$P * Q \approx Q * P$ (32)	$P \Uparrow X \approx (P \Uparrow X) \Downarrow ((Vars(P \Uparrow X)) - X^\uparrow)$ (58)	
	$(P * Q) * R \approx P * (Q * R)$ (33)	8. Laws connecting $\#$ with others:	
4. Laws of $\#$:		$P\#(X \bullet c) \approx P[\exists X \cdot c]$ (59)	
	$P\#(X \bullet c_1) \preceq P$ (34)	$P \Downarrow (xs \setminus x) \approx P\#(\{x : T\} \bullet (xs \approx \{x\}))$ (60)	
	$P\#(X \bullet c_1) \preceq P\#(X \bullet c_2)$, if $(c_1 \Rightarrow c_2)$ (35)	$P \Uparrow x \setminus xs \approx P\#(\{xs : \mathbb{P}(T)\} \bullet (\forall x \in xs \cdot Pred(P)))$ (61)	
	$P\#(X \bullet c_1) \preceq Q\#(X \bullet c_1)$, if $P \preceq Q$ (36)	$P[c] \approx P\#(\emptyset \bullet c)$ (62)	
	$P\#(X \bullet c_1) \preceq P\#(X \bullet c_2)$, if $c_1 \Rightarrow c_2$ (37)	$P \approx TRUE\#(Vars(P) \bullet Pred(P))$ (63)	
	$P \approx TRUE\#(Vars(P) \bullet Pred(P))$ (38)	$P * Q \approx P\#(Vars(Q) \bullet Pred(Q))$ (64)	
	$P\#(\emptyset \bullet True) \approx P$ (39)	$P \Uparrow X \approx P\#(Vars(P \Uparrow X) \bullet Pred(P \Uparrow X))$ (65)	
	$P\#(X \bullet False) \approx FALSE$ (40)	9. Laws connecting $[c]$ with \Uparrow , \Downarrow and \uparrow :	
	$P\#(X \bullet c_1)\#(Y \bullet c_2) \approx P\#(X \cup Y \bullet c_1 \wedge c_2)$ (41)	$P[c] \Uparrow X \approx (P \Uparrow X)[c_\uparrow]$ (66)	
	$P\#(X \bullet c_1)\#(Y \bullet c_2) \approx P\#(Y \bullet c_2)\#(X \bullet c_1)$ (42)	$P[c] \Uparrow X \approx (P \Uparrow X)[c_\uparrow]$ (67)	
		$P[c] \Downarrow X \approx (P \Downarrow X)[c_\downarrow]$ (68)	

Fig. 6. Laws of Pattern Operators

(\Leftarrow) Let $m \models P \uparrow V$. By Definition 2, there is an assignment α such that

$$\llbracket Pred(P \uparrow V) \rrbracket_{\alpha}^m = true. \quad (71)$$

Let $\alpha(vs_i) = A_i$. Equ (71) means that the following predicate is true in model m for all $a_1 \in A_1, \dots, a_k \in A_k$.

$$\exists v_{k+1} \dots \exists v_n \cdot p(a_1, \dots, a_k, v_{k+1}, \dots, v_n)$$

Let a_{k+1}, \dots, a_n be the witnesses for v_{k+1}, \dots, v_n in the above. Then, we have that $p(a_1, \dots, a_n)$ is true in model m . Define assignment $\alpha'(v_i) = a_i$ for each $i \in 1 \dots n$. We have that $\llbracket p(v_1, v_2, \dots, v_n) \rrbracket_{\alpha'}^m = true$. That is, $m \models P$.

(3) *Generalisation.*

(\Rightarrow) Algebraic law (58) in Fig. 6 states that

$$P \uparrow X \approx (P \uparrow X) \Downarrow (V - X^\uparrow),$$

where $V = Vars(P \uparrow X)$ and X^\uparrow is X with all the variables lifted.

Using reflexivity of \preceq on this and proof (2), we get for all valid Y that $(P \uparrow X) \Downarrow Y \preceq (P \uparrow X) \preceq P$. Since \preceq is transitive, and taking $Y = \emptyset$, we have $P \uparrow X \preceq P$. By the definition of the \preceq relation, we have that for all models m that $m \models P \uparrow X \Rightarrow m \models P$.

(\Leftarrow) We prove the statement for a single variable v . The general statement can then easily be proved by induction on the number of variables in the set X .

Let $v \in Vars(P)$. Assume that $m \models P$. By definition, there is an assignment α such that $\llbracket Pred(P) \rrbracket_{\alpha}^m = true$. Let $\alpha' = \alpha[vs := \{a\}]$, where $a = \alpha(v)$. Then, α' is an assignment for $P \uparrow v$. By the definition of $Pred(P \uparrow V)$, we have

$$\begin{aligned} \llbracket Pred(P \uparrow v) \rrbracket_{\alpha'}^m &= \llbracket \forall v \in vs \cdot Pred(P) \rrbracket_{\alpha'}^m \\ &= \llbracket \forall v \in \{a\} \cdot Pred(P) \rrbracket_{\alpha'}^m = \llbracket Pred(P) \rrbracket_{\alpha}^m = true. \end{aligned}$$

By the definition of \models , we then have that $m \models P \uparrow v$. \blacksquare

Proof: of Theorem 4

In each case, the (\Leftarrow) case is similar to the (\Rightarrow) case.

(1) *Restriction.*

(\Rightarrow): Let $m \in \llbracket P[c] \rrbracket$. By definition of the denotational semantics of a pattern, we have that $m \models \text{Spec}(P[c])$. That is, there is an assignment α such that $\llbracket \text{Pred}(P) \wedge c \rrbracket_{\alpha}^m = \text{true}$. This means $\llbracket \text{Pred}(P) \rrbracket_{\alpha}^m = \text{true}$ and $\llbracket c \rrbracket_{\alpha}^m = \text{true}$. The former means that $m \models \text{Spec}(P)$, thus $m \in \llbracket P \rrbracket$. And, the latter means $m \models c$. Therefore, we have that $m \in \{m \mid m \in \llbracket P \rrbracket \wedge m \models c\}$.

(2) *Extension.*

(\Rightarrow): Let $m \in \llbracket P\#(V \bullet c) \rrbracket$. By definition, we have that $m \models \text{Spec}(P\#(V \bullet c))$. Thus, there is an assignment α such that $\llbracket \text{Pred}(P) \wedge \exists V \cdot c \rrbracket_{\alpha}^m = \text{true}$. This means $\llbracket \text{Pred}(P) \rrbracket_{\alpha}^m = \text{true}$ and $\llbracket \exists V \cdot c \rrbracket_{\alpha}^m = \text{true}$. The former means that $m \models \text{Spec}(P)$, so $m \in \llbracket P \rrbracket$. The latter means that $m \models (\exists V \cdot c)$. Therefore, we have that $m \in \{m \mid m \in \llbracket P \rrbracket \wedge m \models \exists V \cdot c\}$.

(3) *Flattening.*

(\Rightarrow): Let $m \in \llbracket P \Downarrow x \rrbracket$. By definition, we have that $m \models \text{Spec}(P \Downarrow x)$. This means there is an assignment α such that $\llbracket \text{Pred}(P) \wedge x = \{x'\} \rrbracket_{\alpha}^m = \text{true}$. This means $\llbracket \text{Pred}(P) \rrbracket_{\alpha}^m = \text{true}$ and $\llbracket x = \{x'\} \rrbracket_{\alpha}^m = \text{true}$. The former means that $m \in \llbracket P \rrbracket$. The latter means that $m \models x = \{x'\}$ for some x' , which is equivalent to $m \models \|x\| = 1$. Therefore, $m \in \{m \mid m \in \llbracket P \rrbracket \wedge m \models \|x\| = 1\}$. ■

APPENDIX 3: THE USE OF SPASS THEOREM PROVER

We have employed SPASS Theorem Prover in the analysis of the validity of pattern compositions. In this section, we give the details about how SPASS is used, i.e. the input to SPASS Theorem Prover and the corresponding output generated by SPASS, for the examples and case study reported in the paper.

A. *Proof of Lost Soundness*

The input to SPASS for proving the lost of soundness of the pattern composition (e) of the motivative example is listed below. It consists of three parts:

- 1) *A list of symbol declarations.* It contains the declarations of components in the patterns.
- 2) *A list of axioms.* It contains the consistency and completeness constraints on UML models, and the specification of the patterns.
- 3) *A list of conjectures to be proved.* It is the condition of lost of soundness.

```
begin_problem(CA).

list_of_descriptions.
name({*Composite Adapter*}).
author({*Ian Bayley*}).
status(unsatisfiable).
description({* Attempt to show that the composition (f) of
Composite and Adapter in IEEE paper is invalid *}).
end_of_list.

list_of_symbols.
functions[
  (component, 0),
  (composite, 0),
  (leaf, 0),
  (target, 0),
  (adapter, 0),
  (adaptee, 0)
].

predicates[
```

```
(Class, 1),
(Associated, 2),
(Inherit, 2),
(AggComp, 2),
(IsAbstract, 1),
(IsInterface, 1)
].
end_of_list.

list_of_formulae(axioms).
formula(forall([x,y], implies(Inherit(x,y),
  not(Inherit(y,x))))).
formula(forall([x,y,z], implies(and(Inherit(x,y),
  Inherit(y,z)), Inherit(x,z)))).

formula(and(
  Class(leaf),
  Class(component),
  Class(composite),
  Inherit(leaf,component),
  Inherit(composite,component),
  AggComp(composite,component),
  not(AggComp(leaf,component)),
  IsAbstract(component)
)).

formula(and(
  Class(target),
  Class(adapter),
  Class(adaptee),
  Inherit(adapter,target),
  AggComp(adapter,adaptee)
)).

end_of_list.

list_of_formulae(conjectures).

formula(not(
  and(
    equal(target, leaf),
    equal(component, adapter)
  )))
end_of_list.

end_problem.
```

The theorem prover was invoked via its web interface by submitting the above input. The prover executed for 0.01 seconds and generated the following output.

WebSPASS - Interactive SPASS

Input Form Submission

You are running 'Mozilla/5.0
(Macintosh; Intel Mac OS X 10_9_4)
AppleWebKit/537.78.2 (KHTML, like Gecko) Version/7.0.6
Safari/537.78.2' from '81.109.126.224'

Your WebSPASS form submission is now being processed...

```
-----SPASS-START-----
Input Problem:
1[0:Inp] || -> Class(target)*.
2[0:Inp] || -> Class(adapter)*.
3[0:Inp] || -> Class(adaptee)*.
4[0:Inp] || -> Class(leaf)*.
5[0:Inp] || -> Class(component)*.
6[0:Inp] || -> Class(composite)*.
7[0:Inp] || -> IsAbstract(component)*.
8[0:Inp] || -> Inherit(adapter,target)*.
9[0:Inp] || -> AggComp(adapter,adaptee)*.
10[0:Inp] || -> Inherit(leaf,component)*.
11[0:Inp] || -> Inherit(composite,component)*.
12[0:Inp] || -> AggComp(composite,component)*.
13[0:Inp] || -> equal(target,leaf)**.
14[0:Inp] || -> equal(adapter,component)**.
15[0:Inp] || AggComp(leaf,component)* -> .
16[0:Inp] || Inherit(U,V)* Inherit(V,U)* -> .
17[0:Inp] || Inherit(U,V)* Inherit(W,U)* -> Inherit(W,V)*.
This is a first-order Horn problem containing equality.
This is a problem that has, if any, a finite domain model.
There are no function symbols.
```

```

The conjecture is ground.
The following monadic predicates have finite extensions:
  IsAbstract, Class.
Axiom clauses: 15 Conjecture clauses: 2
Inferences: ISpR=1 ISpL=1 IORe=1
Reductions: RFRew=1 RBRew=1 RFMR=1 RBMR=1 RObv=1
  RUnC=1 RTaut=1 RFSub=1 RSub=1 RCon=1
Extras : Input Saturation, Always Selection, No Splitting,
  Full Reduction, Ratio: 5, FuncWeight: 1, VarWeight: 1
Precedence: div > id > Class > Associated > Inherit > AggComp
  > IsAbstract > IsInterface > adaptee > adapter > target
  > leaf > composite > component
Ordering : KBO
Processed Problem:

Worked Off Clauses:

Usable Clauses:
7[0:Inp] || -> IsAbstract(component)*.
6[0:Inp] || -> Class(composite)*.
5[0:Inp] || -> Class(component)*.
4[0:Inp] || -> Class(leaf)*.
3[0:Inp] || -> Class(adaptee)*.
14[0:Inp] || -> equal(adapter,component)**.
13[0:Inp] || -> equal(target,leaf)**.
12[0:Inp] || -> AggComp(composite,component)*.
20[0:Rew:14.0,9.0] || -> AggComp(component,adaptee)*.
11[0:Inp] || -> Inherit(composite,component)*.
10[0:Inp] || -> Inherit(leaf,component)*.
21[0:Rew:14.0,8.0,13.0,8.0] ||
-> Inherit(component,leaf)*.
15[0:Inp] || AggComp(leaf,component)* -> .
24[0:Res:21.0,16.1] || Inherit(leaf,component)* -> .
25[0:Res:21.0,17.1] || Inherit(leaf,U)*
-> Inherit(component,U).
22[0:Res:21.0,17.0] || Inherit(U,component)
-> Inherit(U,leaf)*.
16[0:Inp] || Inherit(U,V)* Inherit(V,U)* -> .
17[0:Inp] || Inherit(U,V)* Inherit(V,W)*
-> Inherit(U,W)*.
SPASS V 3.0
SPASS beiseite: Proof found.
Problem: /tmp/webspass-webform_2014-09-15_
23:42:42_59701.txt
SPASS derived 6 clauses, backtracked 0 clauses and
kept 18 clauses.
SPASS allocated 559 KBytes.
SPASS spent 0:00:00.01 on the problem.
0:00:00.00 for the input.
0:00:00.00 for the FLOTTER CNF translation.
0:00:00.00 for inferences.
0:00:00.00 for the backtracking.
0:00:00.00 for the reduction.

Here is a proof with depth 1, length 8 :
8[0:Inp] || -> Inherit(adapter,target)*.
10[0:Inp] || -> Inherit(leaf,component)*.
13[0:Inp] || -> equal(target,leaf)**.
14[0:Inp] || -> equal(adapter,component)**.
16[0:Inp] || Inherit(U,V)* Inherit(V,U)* -> .
21[0:Rew:14.0,8.0,13.0,8.0] || -> Inherit(component,leaf)*.
23[0:Res:21.0,16.0] || Inherit(leaf,component)* -> .
26[0:MRR:23.0,10.0] || -> .
Formulae used in the proof: axiom3 axiom2 conjecture0 axiom0
-----SPASS-STOP-----

SPASS proved that the condition of lost soundness is true,
thus the composition is not sound. It is consistent with the
manual proof given in Section V-D.

B. Proof of RHF Soundness

The input for analysing the soundness of RHF design is of
the same structure as the previous section shown below.

begin_problem(RHF) .

list_of_descriptions.
name({*Request Handling Framework*}).
author({*Ian Bayley*}).
status(unsatisfiable) .

description({* Attempt to show that the RHF composition
is valid *}).
end_of_list.

list_of_symbols.
functions[
(client,0),
(command,0),
(concreteCommand,0),
(invoker,0),
(receiver,0),
(commandProcessor,0),
(componentCP,0),
(originator,0),
(memento,0),
(caretaker,0),
(context,0),
(strategy,0),
(concreteStrategy,0),
(component,0),
(composite,0),
(leaf,0)
].

predicates[
(Class,1),
(Associated,2),
(Inherit,2),
(AggComp,2),
(Creates,2),
(IsAbstract,1),
(IsInterface,1)
].
end_of_list.

list_of_formulae(axioms) .
formula(forall([x,y], implies(Inherit(x,y),
not(Inherit(y,x))))).
formula(forall([x,y,z], implies(and(Inherit(x,y),
Inherit(y,z)), Inherit(x,z)))).

formula(and(
AggComp(invoker,command),
Inherit(concreteCommand,command),
Associated(concreteCommand,receiver),
Associated(client,receiver),
Creates(client,concreteCommand)
)).

formula(
Associated(commandProcessor,componentCP)
).

formula(and(
Creates(originator,memento),
AggComp(caretaker,memento)
)).

formula(and(
AggComp(context,strategy),
Inherit(concreteStrategy,strategy)
)).

formula(and(
Inherit(leaf,component),
Inherit(composite,component),
AggComp(composite,component),
not(AggComp(leaf,component)),
IsAbstract(component)
)).

end_of_list.

list_of_formulae(conjectures) .

formula(not(exists([Application],and(
equal(command,caretaker),
equal(command,componentCP),
equal(originator,Application),
equal(client,commandProcessor),
equal(originator,Application),
equal(leaf,concreteCommand),
equal(component,command),
equal(commandProcessor,context),
equal(caretaker,command)
))))).

```

```

))) .
end_of_list.

end_problem.

```

The execution of the the SPASS theorem prover on the above input takes 0.01 seconds to conclude that "Completion found", which means no proof of the conjecture. In other words, the condition of lost soundness cannot be proved from the pattern specification and consistency and completeness constraints.

WebSPASS - Interactive SPASS

Input Form Submission

```

You are running 'Mozilla/5.0
(Macintosh; Intel Mac OS X 10_9_4)
AppleWebKit/537.78.2 (KHTML, like Gecko) Version/7.0.6
Safari/537.78.2' from '81.109.126.224'
Your WebSPASS form submission is now being processed...

```

-----SPASS-START-----

```

Input Problem:
1[0:Inp] || -> IsAbstract (component) *.
2[0:Inp] || -> Inherit (leaf, component) *.
3[0:Inp] || -> Inherit (composite, component) *.
4[0:Inp] || -> AggComp (composite, component) *.
5[0:Inp] || -> AggComp (context, strategy) *.
6[0:Inp] || -> Inherit (concreteStrategy, strategy) *.
7[0:Inp] || -> Creates (originator, memento) *.
8[0:Inp] || -> AggComp (caretaker, memento) *.
9[0:Inp] || -> Associated (commandProcessor, componentCP) *.
10[0:Inp] || -> AggComp (invoker, command) *.
11[0:Inp] || -> Inherit (concreteCommand, command) *.
12[0:Inp] || -> Associated (concreteCommand, receiver) *.
13[0:Inp] || -> Associated (client, receiver) *.
14[0:Inp] || -> Creates (client, concreteCommand) *.
15[0:Inp] || -> equal (command, componentCP) **.
16[0:Inp] || -> equal (commandProcessor, client) **.
17[0:Inp] || -> equal (originator, originator) *.
18[0:Inp] || -> equal (leaf, concreteCommand) **.
19[0:Inp] || -> equal (command, component) **.
20[0:Inp] || -> equal (commandProcessor, context) **.
21[0:Inp] || -> equal (command, caretaker) **.
22[0:Inp] || AggComp (leaf, component) * -> .
23[0:Inp] || Inherit (U,V) * Inherit (V,U) * -> .
24[0:Inp] || Inherit (U,V) * Inherit (W,U) * -> Inherit (W,V) *.
This is a first-order Horn problem containing equality.
This is a problem that has, if any, a finite domain model.
There are no function symbols.
The conjecture is ground.
The following monadic predicates have finite extensions:
  IsAbstract.
Axiom clauses: 17 Conjecture clauses: 7
Inferences: ISpR=1 ISpL=1 IORe=1
Reductions: RFReW=1 RBReW=1 RFMR=1 RBMR=1 RObv=1
             RUnC=1 RTaut=1 RFSub=1 RBSub=1 RCon=1
Extras      : Input Saturation, Always Selection,
             No Splitting, Full Reduction, Ratio: 5,
             FuncWeight: 1, VarWeight: 1
Precedence: div > id > Class > Associated > Inherit
             > AggComp > Creates > IsAbstract > IsInterface
             > leaf > composite > command > component
             > concreteStrategy > strategy > commandProcessor
             > context > caretaker > memento > originator
             > componentCP > receiver > invoker > concreteCommand
             > client
Ordering   : KBO
Processed Problem:
Worked Off Clauses:

```

```

Usable Clauses:
32[0:Rew:29.0,26.0] ||
-> IsAbstract (componentCP) *.
18[0:Inp] ||
-> equal (leaf, concreteCommand) **.
27[0:Rew:20.0,16.0] ||
-> equal (context, client) **.
29[0:Rew:21.0,15.0] ||
-> equal (caretaker, componentCP) **.

```

```

28[0:Rew:27.0,20.0] ||
-> equal (commandProcessor, client) **.
30[0:Rew:29.0,21.0] ||
-> equal (command, componentCP) **.
31[0:Rew:29.0,25.0] ||
-> equal (component, componentCP) **.
14[0:Inp] ||
-> Creates (client, concreteCommand) *.
7[0:Inp] ||
-> Creates (originator, memento) *.
13[0:Inp] ||
-> Associated (client, receiver) *.
12[0:Inp] ||
-> Associated (concreteCommand, receiver) *.
36[0:Rew:28.0,9.0] ||
-> Associated (client, componentCP) *.
33[0:Rew:29.0,8.0] ||
-> AggComp (componentCP, memento) *.
34[0:Rew:27.0,5.0] ||
-> AggComp (client, strategy) *.
38[0:Rew:30.0,10.0] ||
-> AggComp (invoker, componentCP) *.
39[0:Rew:31.0,4.0] ||
-> AggComp (composite, componentCP) *.
6[0:Inp] ||
-> Inherit (concreteStrategy, strategy) *.
37[0:Rew:30.0,11.0] ||
-> Inherit (concreteCommand, componentCP) *.
40[0:Rew:31.0,3.0] ||
-> Inherit (composite, componentCP) *.
42[0:Rew:18.0,22.0,31.0,22.0] ||
AggComp (concreteCommand, componentCP) * -> .
44[0:Res:37.0,23.0] ||
Inherit (componentCP, concreteCommand) * -> .
48[0:Res:40.0,23.0] ||
Inherit (componentCP, composite) * -> .
46[0:Res:37.0,24.1] ||
Inherit (componentCP, U) * -> Inherit (concreteCommand, U) .
50[0:Res:40.0,24.1] ||
Inherit (componentCP, U) -> Inherit (composite, U) *.
43[0:Res:37.0,24.0] ||
Inherit (U, concreteCommand) -> Inherit (U, componentCP) *.
47[0:Res:40.0,24.0] ||
Inherit (U, composite) * -> Inherit (U, componentCP) .
23[0:Inp] ||
Inherit (U,V) * Inherit (V,U) * -> .
24[0:Inp] ||
Inherit (U,V) * Inherit (V,W) * -> Inherit (U,W) *.
Given clause: 32[0:Rew:29.0,26.0] ||
-> IsAbstract (componentCP) *.
Given clause: 18[0:Inp] ||
-> equal (leaf, concreteCommand) **.
Given clause: 27[0:Rew:20.0,16.0] ||
-> equal (context, client) **.
Given clause: 29[0:Rew:21.0,15.0] ||
-> equal (caretaker, componentCP) **.
Given clause: 28[0:Rew:27.0,20.0] ||
-> equal (commandProcessor, client) **.
Given clause: 30[0:Rew:29.0,21.0] ||
-> equal (command, componentCP) **.
Given clause: 31[0:Rew:29.0,25.0] ||
-> equal (component, componentCP) **.
Given clause: 14[0:Inp] ||
-> Creates (client, concreteCommand) *.
Given clause: 7[0:Inp] ||
-> Creates (originator, memento) *.
Given clause: 13[0:Inp] ||
-> Associated (client, receiver) *.
Given clause: 12[0:Inp] ||
-> Associated (concreteCommand, receiver) *.
Given clause: 36[0:Rew:28.0,9.0] ||
-> Associated (client, componentCP) *.
Given clause: 33[0:Rew:29.0,8.0] ||
-> AggComp (componentCP, memento) *.
Given clause: 34[0:Rew:27.0,5.0] ||
-> AggComp (client, strategy) *.
Given clause: 38[0:Rew:30.0,10.0] ||
-> AggComp (invoker, componentCP) *.
Given clause: 39[0:Rew:31.0,4.0] ||
-> AggComp (composite, componentCP) *.
Given clause: 6[0:Inp] ||
-> Inherit (concreteStrategy, strategy) *.
Given clause: 37[0:Rew:30.0,11.0] ||
-> Inherit (concreteCommand, componentCP) *.
Given clause: 40[0:Rew:31.0,3.0] ||
-> Inherit (composite, componentCP) *.

```

```
Given clause: 42[0:Res:18.0,22.0,31.0,22.0] ||
  AggComp(concreteCommand,componentCP)** -> .
Given clause: 44[0:Res:37.0,23.0] ||
  Inherit(componentCP,concreteCommand)** -> .
Given clause: 48[0:Res:40.0,23.0] ||
  Inherit(componentCP,composite)** -> .
Given clause: 46[0:Res:37.0,24.1] ||
  Inherit(componentCP,U)** -> Inherit(concreteCommand,U).
Given clause: 50[0:Res:40.0,24.1] ||
  Inherit(componentCP,U)+ -> Inherit(composite,U)*.
Given clause: 23[0:Inp] ||
  Inherit(U,V)** Inherit(V,U)* -> .
Given clause: 57[0:Res:6.0,23.0] ||
  Inherit(strategy,concreteStrategy)** -> .
Given clause: 43[0:Res:37.0,24.0] ||
  Inherit(U,concreteCommand)+ -> Inherit(U,componentCP)*.
Given clause: 47[0:Res:40.0,24.0] ||
  Inherit(U,composite)** -> Inherit(U,componentCP).
Given clause: 24[0:Inp] ||
  Inherit(U,V)** Inherit(V,W)* -> Inherit(U,W)*.
Given clause: 60[0:Res:6.0,24.0] ||
  Inherit(strategy,U)+ -> Inherit(concreteStrategy,U)*.
SPASS V 3.0
SPASS beiseite: Completion found.
Problem: /tmp/webspass-webform_2014-09-16_
00:02:27_149461.txt
SPASS derived 31 clauses, backtracked 0 clauses and
kept 41 clauses.
SPASS allocated 568 KBytes.
SPASS spent 0:00:00.01 on the problem.
0:00:00.00 for the input.
0:00:00.00 for the FLOTTER CNF translation.
0:00:00.00 for inferences.
0:00:00.00 for the backtracking.
0:00:00.00 for the reduction.
```

-----SPASS-STOP-----