AFM Technical Report

Number: CCT-AFM-2014-01; *Date*: 17 Oct. 2014; *Version*: 1.0

On the Composability of Design Patterns

Hong Zhu

Email: hzhu@brookes.ac.uk

and

Ian Bayley

Email: ibayley@brookes.ac.uk

Applied Formal Methods Research Group

Department of Computing and Communication Technologies

Oxford Brookes University

Oxford OX33 1HX, UK

On the Composability of Design Patterns

Hong Zhu 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, Composibility, 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] (referred to as the *GoF book* in the sequel), 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 GoF patterns Composite and Adapter, with the latter indicated by shading. Are these valid and is there a way to prove that they are?

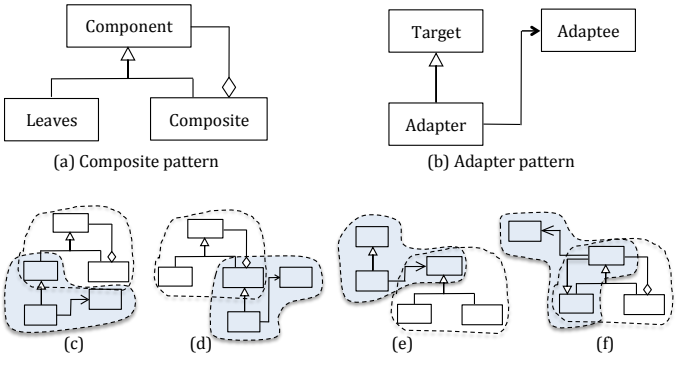

Fig. 1. Motivative examples of pattern compositions

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 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 work of many others on formalisation of design patterns [33]–[41].

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

II. Related Work

Although each pattern is specified separately, they are usually to be found composed with each other [43]. Thus, pattern composition plays a crucial role in the effective use of design knowledge. The past few years have seen a rapid growth in research on this topic. Existing 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.

A. Representation of Pattern Compositions and Instantiations

Compositions can be represented either visually or formally.

1) Visual representation: This is usually informal [42], [44]. Visual notations such as the Venn diagram with *Pattern:Role* annotation proposed by Vlissides [45] have been widely used in practice to show the component parts of the composition. Dong et al. [46] 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 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 [47] proposed the PIN notation (Pattern Instance Notation) to represent the same information in a hierarchical manner.

2) 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 [48] and [49], 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. 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 [50]. In [51], Dong et al. define instantiation again as a mapping, but from components in the pattern (e.g. concrete classes, attributes, methods) to those in the actual system.

Taibi 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.

Both of these approaches effectively specify how components from the composed patterns overlap in the composite pattern. In our previous work [52], 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 links 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], [53], [54], 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 [53], these operators are expressive enough to capture all pattern compositions suggested by the GoF book, 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.

B. 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 [55], Prechelt et al. [56], 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 [57], which uses quality metrics to measure the improvement effectiveness, and Ampatzoglou et al.'s methodology [58] 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. [48] propose the following *faithfulness conditions* to ensure that composition makes sense.

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

They showed out to verify these conditions for an example where Composite is composed with Iterator. Taibi also requires these conditions but argued in [59] that condition (2) is not always necessary.

III. The Proposed Approach

In this section, we outline our own approach to the open problem of verifying that a composition is valid. We refine the problem into defining and 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 meanings 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 are expressions formed from the application of six pattern operators [52] 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 present a formal method to enable software designers to prove or disprove the validity of pattern composition, by considering soundness preserving, 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 semantics 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 [52], [32]
  - a case study of a real pattern-oriented design.

IV. Preliminaries

This section lays the foundation for the paper by defining the preliminaries. We first recall the logic underlying the formal specification of design patterns, then the pattern composition and instantiation operators [32].

A. Logic Systems 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 the behavioural features, too, 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 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], [60]–[62] 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 [63], [64]. 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 $Vars = \{v_1 : T_1, \cdots, v_n : T_n\}$ 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), to represent a set of elements of the type Z, or $\mathbb{P}(\mathbb{P}(Z))$ (i.e. the power set of the power set of Z), etc. Note that, for the sake of convenience, we do not allow the empty set $\emptyset$ to be an instance of a power set type $\mathbb{P}(T)$.

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

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

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

Often predicate $Pred$ is split into static and dynamic conditions as in [38] and [41]. It can also be specialised to particular representations of software systems such as program code, UML diagrams etc, though in this paper, we will just consider the latter for our concrete examples for simplicity. The operators we use from [32], [53], [54] are also independent of the particular formalism, although the examples come from the previous work [41] and [65]. The theory developed in this paper is valid as far as the following notion of conformance is valid and the logic is consistent.

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

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

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

Proposition 1: (Consistency of Specification Logic)

For all models m and predicates p and q on models, we have that $\vdash (p \Rightarrow q)$ and $m \models p$ imply that $m \models q$. □

Note that the logic system also has axioms about the atomic predicates of software systems. One such predicate is $\longrightarrow\!\!\triangleright$, where $X \longrightarrow\!\!\triangleright Y$ means that X is a subclass of Y. Two of its

axioms are the transitivity and asymmetry properties below. $\forall X, Y, Z \in Class$,

$$(X \longrightarrow\!\!\triangleright\, Y) \wedge (Y \longrightarrow\!\!\triangleright\, Z) \implies X \longrightarrow\!\!\triangleright\, Z. \quad (2)$$

$$\neg(X \longrightarrow\!\!\triangleright\, Y \wedge Y \longrightarrow\!\!\triangleright\, X) \quad (3)$$

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.

Definition 3: (Specialisation Relation between Patterns) Let P and Q be design patterns. Pattern P is a *specialisation* of Q, written $P \preccurlyeq Q$, if for all models m, whenever m conforms to P, m also conforms to Q. Formally, $P \preccurlyeq Q \triangleq \forall m \cdot (m \models P \ \Rightarrow m \models Q)$.

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

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

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

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

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

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

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

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

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

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

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

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

□

Informal explanations of the operators are as follows. Restriction operator $P[c]$ imposes an additional condition c on an existing pattern P. A common use of restriction as shown in our case studies [53] is 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' is written as $P[x\backslash x']$; this 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 $Target$s of adapted classes. Each of these has a dependent $Adapter$ and $Adaptee$ class configured as in the original $Adapter$ pattern. In other words, the component $Target$ in the lifted pattern plays a role similar to the primary key in a relational database. The difference between lifting and generalisation is illustrated in Example 1.

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

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

Example 1: Consider patterns P and Q defined as below.

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

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

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

Consider the pattern compositions R_1 to R_4 defined as follows:

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

where for any classes X and Y, $X \diamond\!\longrightarrow Y$ means X has part Y, i.e. there is composite/aggregate relation from X to Y.

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

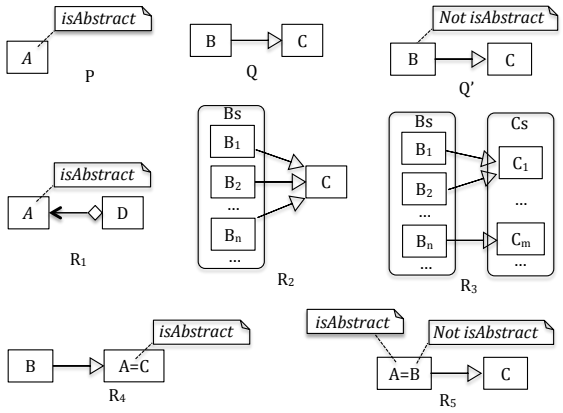

Fig. 2. Illustration of the Patterns in Examples 1 to 4

From the definitions of the operators we immediately have:

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

□

In [32], we proved a complete set of the equational laws that the operators obey. Some of theme 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. 3, where the following notations are used.

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

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

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

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

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

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

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

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

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, $Spec(\oplus P) \Rightarrow Spec(P)$.

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

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

(b) Assume that for a property p, we have that $Spec(P) \vdash p$. Because $Spec(P \oplus Q) \Rightarrow Spec(P)$, we have that $Spec(P \oplus Q) \Rightarrow p$. If we have that $Spec(Q) \vdash p$, then, because $Spec(P \oplus Q) \Rightarrow Spec(Q)$, we have that $Spec(P \oplus Q) \Rightarrow p$. Thus, we have that

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

That is, $\oplus$ preserves features. ■

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

1. Laws of $\preccurlyeq$:

$$P \preccurlyeq P \tag{4}$$
$$(P \preccurlyeq Q) \wedge (Q \preccurlyeq R) \Rightarrow (P \preccurlyeq R) \tag{5}$$
$$P \preccurlyeq Q \wedge Q \preccurlyeq P \Rightarrow P \approx Q \tag{6}$$
$$FALSE \ \preccurlyeq P \preccurlyeq \ TRUE \tag{7}$$

2. Laws of $[c]$:

$$(c_1 \Rightarrow c_2) \Rightarrow P[c_1] \preccurlyeq P[c_2] \tag{8}$$
$$P[c][c] \approx P[c] \tag{9}$$
$$P[c_1][c_2] \approx P[c_2][c_1] \tag{10}$$
$$P[c_1][c_2] \approx P[c_1 \wedge c_2] \tag{11}$$
$$P[true] \approx P \tag{12}$$
$$P[false] \approx FALSE \tag{13}$$

3. Laws of $*$:

$$(P * Q) \preccurlyeq P \tag{14}$$
$$Q \preccurlyeq P \Rightarrow P * Q \approx Q \tag{15}$$
$$P * P \approx P \tag{16}$$
$$P * TRUE \approx TRUE * P \approx P \tag{17}$$
$$P * FALSE \approx FALSE * P \ \approx \ FALSE \tag{18}$$
$$P * Q \approx Q * P \tag{19}$$
$$(P * Q) * R \approx P * (Q * R) \tag{20}$$

4. Laws of $\#$:

$$P\#(X \bullet c_1) \preccurlyeq P \tag{21}$$
$$P\#(X \bullet c_1) \preccurlyeq P\#(X \bullet c_2), \ if \ (c_1 \Rightarrow c_2) \tag{22}$$
$$P\#(X \bullet c_1) \preccurlyeq Q\#(X \bullet c_1), \ if \ P \preccurlyeq Q \tag{23}$$
$$P\#(X \bullet c_1) \preccurlyeq P\#(X \bullet c_2), \ if \ c_1 \Rightarrow c_2 \tag{24}$$
$$P \approx TRUE\#(Vars(P) \bullet Pred(P)) \tag{25}$$
$$P\#(\emptyset \bullet True) \approx P \tag{26}$$
$$P\#(X \bullet False) \approx FALSE \tag{27}$$
$$P\#(X \bullet c_1)\#(Y \bullet c_2) \ \approx P\#(X \cup Y \bullet c_1 \wedge c_2) \tag{28}$$
$$P\#(X \bullet c_1)\#(Y \bullet c_2) \ \approx P\#(Y \bullet c_2)\#(X \bullet c_1) \tag{29}$$

5. Laws of $\Downarrow$ and $\Uparrow$:

$$(P \Downarrow \emptyset) \approx P \tag{30}$$
$$(P \Downarrow X) \Downarrow Y \approx (P \Downarrow Y) \Downarrow X \tag{31}$$
$$(P \Downarrow X) \Downarrow Y \approx P \Downarrow (X \cup Y) \tag{32}$$
$$(P \Uparrow \emptyset) \approx P \tag{33}$$
$$(P \Uparrow X) \Uparrow Y \approx (P \Uparrow Y) \Uparrow X \tag{34}$$
$$(P \Uparrow X) \Uparrow Y \approx P \Uparrow (X \cup Y) \tag{35}$$

6. Laws connecting $*$ with others:

$$P[c] * Q \approx (P * Q)[c] \tag{36}$$
$$(P \Uparrow X) * Q \approx (P * Q) \Uparrow X \tag{37}$$
$$(P \Downarrow X) * Q \approx (P * Q) \Downarrow X \tag{38}$$
$$(P * Q) \Uparrow X \ \approx (P \Uparrow X_P) * (Q \Uparrow X_Q) \tag{39}$$
$$(P * Q) \Downarrow X \ \approx (P \Downarrow X_P) * (Q \Downarrow X_Q) \tag{40}$$
$$(P \uparrow X) * Q \ \approx ((P * Q) \uparrow X) \Downarrow Vars(Q) \tag{41}$$

7. Laws connecting $\Uparrow$, $\Downarrow$ and $\uparrow$:

$$(P \Uparrow X \backslash U) \Downarrow (U \backslash X) \approx P \tag{42}$$
$$(P \Downarrow X \backslash U) \Uparrow U \backslash X) \approx P \tag{43}$$
$$(P \uparrow x) \Downarrow (Vars(P \uparrow x)) \approx P \tag{44}$$
$$P \Uparrow X \approx (P \uparrow X) \Downarrow ((Vars(P \uparrow X)) - X^{\uparrow}) \tag{45}$$

8. Laws connecting $\#$ with others:

$$P\#(X \bullet c) \approx P[\exists X \cdot c] \tag{46}$$
$$P \Downarrow (xs \backslash x) \approx P\#(\{x : T\} \bullet (xs \approx \{x\}) \tag{47}$$
$$P \Uparrow x \backslash xs \approx P\#(\{xs : \mathbb{P}(T)\} \bullet (\forall x \in xs \cdot Pred(P)) \tag{48}$$
$$P[c] \approx P\#(\emptyset \bullet c) \tag{49}$$
$$P \approx TRUE\#(Vars(P) \bullet Pred(P)) \tag{50}$$
$$P * Q \approx P\#(Vars(Q) \bullet Pred(Q)) \tag{51}$$
$$P \uparrow X \approx P\#(Vars(P \uparrow X) \bullet Pred(P \uparrow X)) \tag{52}$$

9. Laws connecting $[c]$ with $\Uparrow$, $\Downarrow$ and $\uparrow$:

$$P[c] \Uparrow X \approx (P \Uparrow X)[c_{\Uparrow}] \tag{53}$$
$$P[c] \uparrow X \approx (P \uparrow X)[c_{\uparrow}] \tag{54}$$
$$P[c] \Downarrow X \approx (P \Downarrow X)[c_{\Downarrow}] \tag{55}$$

Fig. 3. Laws of Pattern Operators

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

$$\forall p \in Thm_P \cdot (Spec(P) \Rightarrow p), \quad (56)$$
$$\forall q \in Thm_Q \cdot (Spec(Q) \Rightarrow q). \quad (57)$$

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 (56) and statement (b) of Lemma 1, we have that $Spec(P \oplus Q) \Rightarrow s$. Similarly, if $s \in Thm_Q$ then by (57) and statement (b) again, $Spec(P \oplus Q) \Rightarrow s$. So for all statements $s \in Thm_P \cup Thm_Q$ implies that $Spec(P \oplus Q) \Rightarrow s$. This means that $\oplus$ preserves completeness. ■

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

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

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

□

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 $[\![P]\!]$, is the set of models m that satisfy the specification. Formally,

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

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

$$P \approx Q \Leftrightarrow [\![P]\!] = [\![Q]\!], \quad (58)$$
$$P \preccurlyeq Q \Leftrightarrow [\![P]\!] \subseteq [\![Q]\!]. \quad (59)$$

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

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

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

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

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

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

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

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

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

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

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

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

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 similar.

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

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

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

Then, we have that

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

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

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

because the following is not satisfiable.

$$Spec(R_5) \quad = \quad \exists A, C : Class \cdot ((A \longrightarrow\!\!\!\!\triangleright\, C) \wedge \neg A.isAbstract \wedge A.isAbstract),$$

□

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.

VI. Analysis of Pattern Operators

Now we analyse the preservation properties of the operators, proving a set of general theorems. The lengthier proofs are in the appendix.

A. Feature Preservation Properties

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

Proof: (Theorem 1)

In each case, we prove $Spec(\oplus(P)) \Rightarrow Spec(P)$ 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 $X = \{x_1 : T_1, \cdots, x_n : T_n\}$ in each case.

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

$$Spec(P\,[c]) = \exists X \cdot (Pred(P) \wedge c)$$
$$\Rightarrow \exists X \cdot (Pred(P)) = Spec(P).$$

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

$$Spec(P\#(V \bullet c)) = \exists X\ \exists V \cdot (Pred(P) \wedge c)$$
$$\Rightarrow \exists X\ \exists V \cdot (Pred(P)) \Rightarrow \exists X \cdot (Pred(P)) = Spec(P)$$

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

$$Spec(P \Downarrow x \backslash x') = \exists x' : T \cdot \exists X \cdot (p(\{x'\}, x_1, \cdots, x_n))$$
$$\Rightarrow \exists x : \mathbb{P}(T) \cdot \exists X \cdot (p(x, x_1, \cdots, x_n)) = Spec(P)$$

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 $k > 0$.

$$Spec(P \Downarrow X) \Rightarrow Spec(P). \tag{60}$$

Then, for all variable sets X' of size $k+1$, $X' = \{x\} \cup X$, where X is of size k. By algebraic law (32) in Fig. 3, the following equality is true.

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

Therefore, we have that

$$Spec(P \Downarrow X') = Spec(P \Downarrow (\{x\} \cup X))$$
$$= Spec((P \Downarrow x) \Downarrow X)$$
$$\Rightarrow Spec(P \Downarrow x) \Rightarrow Spec(P)$$

Therefore, for all variable sets of size $k+1$, we also have the property (60).

So by mathematical induction, 60 is true for finite sets of variables X and by Lemma 1, the operator $\Downarrow$ is feature-preserving. ■

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

As shown in the case studies and examples given in [53], pattern compositions are expressions formed from patterns and the six operators. Using Theorem 1, by induction on the structure of expression, we can prove all such pattern expressions are feature preserving. Thus, we have the following theorem.

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

Proof: (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 $Spec(E) \Rightarrow Spec(P_i)$.

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

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

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 $X \subseteq Vars(P)$, we have that for all models m,

$$(m \models P * Q) \Leftrightarrow ((m \models P) \wedge (m \models Q)), \tag{61}$$
$$(m \models (P \uparrow X)) \Leftrightarrow (m \models P), \tag{62}$$
$$(m \models (P \Uparrow X)) \Leftrightarrow (m \models P). \tag{63}$$

□

Proof: (Theorem 3)

(1) *Superposition.*

By definition, $m \models (P * Q)$ if and only if there is an assignment α from $Vars(P * Q)$ to elements in m such that $[\![Pred(P * Q)]\!]^m_\alpha$ is true.

By Definition 5, $Pred(P * Q) = Pred(P) \wedge Pred(Q)$. In predicate logic, we have that $[\![Pred(P) \wedge Pred(Q)]\!]^m_\alpha = true$ if and only if $[\![Pred(P)]\!]^m_\alpha = true$ and $[\![Pred(Q)]\!]^m_\alpha = true$.

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

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

Therefore, the theorem holds for the superposition operator.

(2) *Lifting.*

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

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

$$[\![Pred(P)]\!]^m_\alpha = true. \tag{64}$$

We have that Equ (64) means that $p(a_1, a_2, \cdots, a_n)$ is true in model m.

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

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

In other words, $[\![Pred(P \uparrow V)]\!]^m_{\alpha'} = true$. Therefore, we have that $m \models P \uparrow V$.

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

$$[\![Pred(P \uparrow V)]\!]^m_\alpha = true. \tag{65}$$

Let $\alpha(vs_i) = A_i$. Equ (65) means that the following predicate is true in model m for all $a_1 \in A_1$, $\cdots$, $a_k \in A_k$.

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

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

(3) *Generalisation.*

($\Rightarrow$) Algebraic law (45) in Fig. 3 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 $\preccurlyeq$ on this and proof (2), we get for all valid Y that $(P \uparrow X) \Downarrow Y \preccurlyeq (P \uparrow X) \preccurlyeq P$. Since $\preccurlyeq$ is transitive, and taking $Y = \emptyset$, we have $P \Uparrow X \preccurlyeq P$. By the definition of the $\preccurlyeq$ 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 $[\![Pred(P)]\!]^m_\alpha = true$. 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

$$\begin{aligned}[\![Pred(P \Uparrow v)]\!]^m_{\alpha'} &= [\![\forall v \in vs \cdot Pred(P)]\!]^m_{\alpha'} \\ &= [\![\forall v \in \{a\} \cdot Pred(P)]\!]^m_{\alpha'} = [\![Pred(P)]\!]^m_\alpha = true.\end{aligned}$$

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

An immediate corollary is the following.

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

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

These operators change the structure of the pattern without affecting conformance. They are usually applied, as seen in [53], in preparation for restriction and extension, which do affect conformance since they add constraints.

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

$$[\![P[c]]\!] = \{m | m \in [\![P]\!] \wedge m \models c\}, \tag{66}$$

$$[\![P\#(V \bullet c)]\!] = \{m | m \in [\![P]\!] \wedge m \models \exists V \cdot c\}, \tag{67}$$

$$[\![P \Downarrow x]\!] = \{m | m \in [\![P]\!] \wedge m \models (||x|| = 1).\} \tag{68}$$

□

Proof: (Theorem 4)

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

(1) *Restriction.*

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

(2) *Extension.*

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

(3) *Flattening.*

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

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

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

C. Soundness Preservation Properties

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

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

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

Example 5: (Counterexamples of Soundness Preservation)

(1) *Restriction.* Suppose $[\![P]\!] \neq \emptyset$. But $[\![P[false]]\!] = \emptyset$ because $P[false] \approx FALSE$.

(2) *Extension.* Suppose $[\![P]\!] \neq \emptyset$ again. But $[\![P\#(V \bullet false)]\!] = \emptyset$ because $P\#(V \bullet false) \approx FALSE$.

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

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

Corollary 4: (Conditions of losing soundness)

Let P be any given pattern. We have that

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

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

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.

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

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

- *Semantics Preservation.*

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

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

As $[\![Composite]\!] \neq [\![Adapter]\!]$, we have $[\![(c)]\!] \subset [\![Adapter]\!]$ and $[\![(c)]\!] \subset [\![Composite]\!]$.

So (c) does not preserve semantics but instead restricts the semantics with a further condition. Compositions (d) and (f) are similar.

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

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

- *Soundness Preservation.*

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

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

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

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

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

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

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

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

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 proofs of the conditions of lost soundness can be performed by employing a theorem prover such as SPASS[1]. Appendix 3 gives the details of using SPASS in the proof of above example.

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

[1] http://www.spass-prover.org

VII. Analysis of Overlap-based Compositions

In both Dong et al's and Taibi's approaches, the composition of patterns A and B is formally expressed as mapping of components of patterns A and B to the components in the result pattern. This describes how components in the composed patterns overlap. This approach is further developed in our previous work [52], where pattern composition is formally defined in terms of overlaps between components in the patterns composed. Three types of overlaps were identified and pattern compositions were defined for each kind of overlaps. In this section, we re-express them in terms of 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 $[\![Pred(P)]\!]^m_{\alpha_1} = true$ and $[\![Pred(Q)]\!]^m_{\alpha_2} = true$. There is an *overlap* between two assignments if there is an element of the model m assigned to two variables, one in $Vars(P)$ and the other in $Vars(Q)$. There are three types of overlaps, distinguished by whether the variables are elements (one-to-one), sets of elements (many-to-many) or one of each (many-to-one or one-to-many). The following defines composition with various types of overlaps using the pattern operators.

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

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

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

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

For example, in Definition 12, T could be the type $Class$, 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 $=$.

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

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

$$(P\langle vs \gtrless_\subset us\rangle Q) \preccurlyeq (P\langle vs \gtrless_\subseteq us\rangle Q) \tag{69}$$
$$(P\langle vs \gtrless_= us\rangle Q) \preccurlyeq (P\langle vs \gtrless_\subseteq us\rangle Q) \tag{70}$$
$$(P\langle vs \gtrless_\subseteq us\rangle Q) \preccurlyeq (P\langle vs \gtrless us\rangle Q) \tag{71}$$

Proof: The ordering relations follow Law (8) in Fig. 3, and the fact that $(vs \subset us) \Rightarrow (vs \subseteq us)$, $(vs = us) \Rightarrow (vs \subseteq us)$ and $(vs \subseteq us) \Rightarrow (vs \cap us \neq \emptyset)$. ■

The third sort of composition is defined as follows.

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

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

Naturally, a composition with many-to-one overlap can also be defined by symmetry. The version in [52] however is slightly more complex in that P is first lifted to duplicate its class components. It is defined as follows.

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

$$P\langle v_\uparrow —\!<_\subseteq us\rangle Q \triangleq (P \uparrow (v \backslash vs) * Q)[vs \subseteq us] \quad \square$$

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.

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

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

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

$$P\langle v_\uparrow —\!<_R us\rangle Q \quad \preccurlyeq \quad P\langle v_\Uparrow —\!<_R us\rangle Q,$$
$$P\langle v_\uparrow —\!<_\subseteq us\rangle Q \quad \preccurlyeq \quad P\langle v —\!< us\rangle Q.$$

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

The proof of the theorem is omitted for the sake of space.

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

The above $\preccurlyeq$ relationships between these composition operators are summarised in Fig. 4, where nodes represent various composition operators and an arrow from node A to node B means $A \preccurlyeq B$. On the right-hand side of Fig. 4 are the ordering relations given in Theorem 5. On the left-hand side are the $\preccurlyeq$ relationships between the one-to-many overlap composition operators.

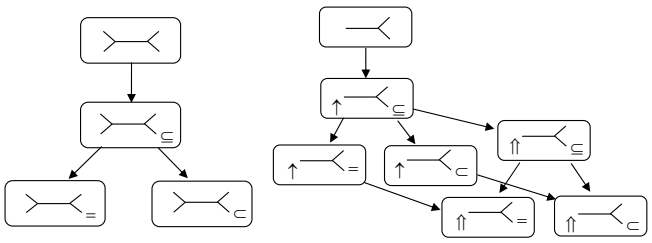

Fig. 4. Relationships Between Compositions with Overlaps

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) By the theorems of feature preservation, semantics preservation and soundness preservation for the operators we can draw conclusions about their validity.

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

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

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

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

(II) *Many-to-Many Overlaps.* For a many-to-many overlap composition $P\langle vs \gtrdot\!\lessdot_R us\rangle Q$, we have that

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

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

3) it loses soundness, if

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

where $(vs\ R\ us)$ 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 $P\langle v_\dagger \,—\!\lessdot_R us\rangle Q$, 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

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

3) it loses soundness, if

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

Proof: To save space, we only give the proof of (I). The proofs for (II) and (III) are similar.

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

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

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

(2) For statement 2), we have that

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

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

VIII. 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 [29]. Table I summarises the five design decisions that result in the design depicted in Figure 5. [2] A case study with this example on how design decisions can be formally expressed using pattern operators can be found in [32].

We now turn to the validity problem, applying the same three-step process as before. As we shall see, it will not only identify two differences between the original manual design presented in [42] and the design formally derived from the algebra of design patterns as detected in our previous case study [32]. It will also enable us to prove that the differences are indeed errors in the manual design and to prove that the formal design is valid.

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, the feature in design pattern is not preserved in the design.

B. Semantics Preservation

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

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

where RHF_O denote 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

[2]Note that, there are two different versions of Command Processor pattern in the literature by the same group of authors [9], [66]. The one used in [42] is the one given in [9].

TABLE I
DESIGN DECISIONS MADE IN THE DESIGN OF RHF

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 in between client and the Command class.
The system need to support undoing the actions performed in response to requests.	Use Memento pattern. The Memento component maintains copies of the states of the Originator, which is the Application class. *The Caretaker component creates a memento*, holds it over time, and if needed, passes it back to the Originator.
Requests from client must be logged. Requests from different users may be logged differently.	Apply Strategy pattern. The CommandProcessor passes the requests it received to a logging context, i.e. the context role in Strategy, which implements the invariant parts of the logging service and delegates the customer-specific logging aspects to the ConcreteStrategy component in Strategy.
The system should support compound commands, which are aggregates of other commands executed in a particular order.	Use the Composite pattern with atomic commands as the Leaves and compound commands as the Composite. Thus, add a new class CompoundCommand and an whole-part relation from this new class to the Command class.

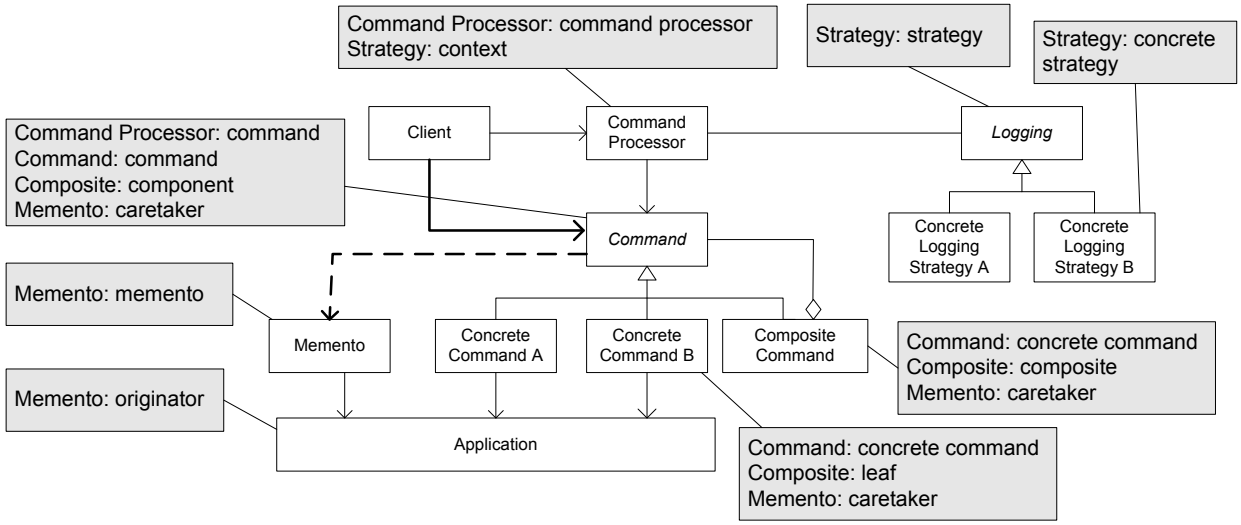

Fig. 5. Original Design of Request Handling Framework as in [42]

the designer intended to do, so it is a semantics error and is removed from the our revised version of the design.

Fixing the above two problems led to the revised design depicted in Fig. 6.

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.

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

By applying algebraic laws, we can rewrite this to the following, which exactly matches the diagram in Fig 6.

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

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

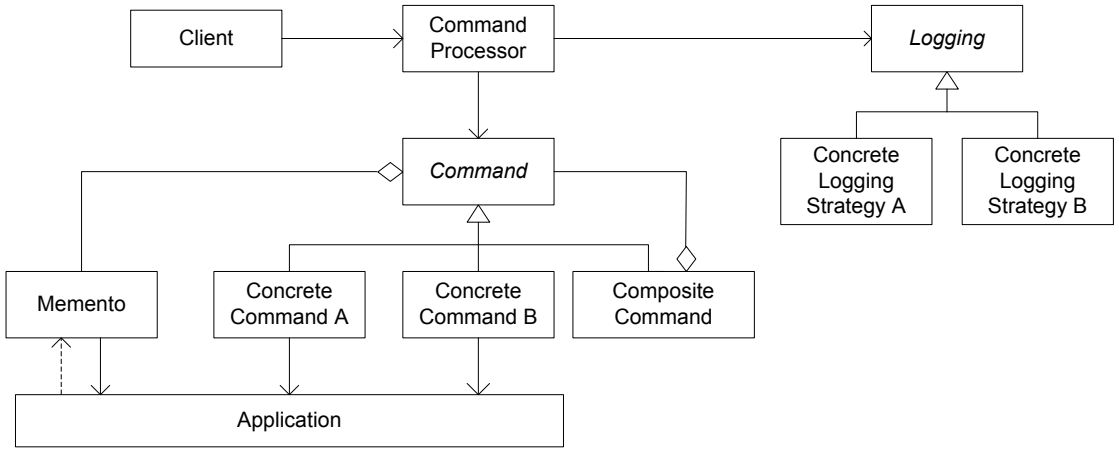

Fig. 6. Revised Design of Request Handling Framework

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.

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

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 Appendix 3.B. 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.

IX. Conclusion

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], [53]. The theory is applied this the theoretical analysis of pattern composition 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

Dong et al.'s [48] were perhaps the first to study compositions of design patterns and their 'correctness'. For a composition P of $P_1, \cdots, P_n$, Dong *et al.* define a composition mapping $C : P_1 \times \cdots \times P_n \to P$ that associates names of component pattern P_i to those of P.[3] Let each P_i have a set θ_i of properties and the composition have the set θ' of properties. A mapping M translates the sentences in each θ_i to θ', preserving the types of each variable. Their faithfulness conditions are as follows:

1) for every sentence S, if $S \in \theta_i$ then $M(S) \in \theta'$;
2) if $S \notin \theta_i$ then $M(S) \notin \theta'$.

Condition (1), that composition should not lose properties, is equivalent to our feature preservation condition, but limited to one-to-one overlaps. Condition (2), means that composition should not gain properties, but it is very difficult if not impossible to prove, and not necessary, as argued by Taibi and Ngo [59]: "*while the second condition of faithfulness is relevant to component, it is not always necessary in the case of patterns*".

Taibi and Ngo's approach [49], [59] is similar, but they represent compositions as combinations of predicates. These predicates are first substituted with the variables of the composite pattern or with constants to represent instantiation. More formally, for patterns P_1 and P_2 with properties φ_1 and φ_2 their composition is given by

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

where the terms t_i are either variables or constants.

Mathematically speaking, these substitutions are equivalent to the name mappings of Dong et al. Both must preserve types of variables for the resulting formulae to be well-typed and for that reason both approaches can only express one-to-one

[3] In fact, they use a set of *name mappings* $C_i : P_i \to P$.

and many-to-many overlaps, but not one-to-many overlaps. Taibi observed the faithfulness conditions 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 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 can apply laws for the operators and logic reasoning, even 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 [67]. 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 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 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 automated theorem prover SPASS. It indicates that it is is feasible to design and implement such a tool.

REFERENCES

[1] P. Coad, "Object-oriented patterns," *Communications of the ACM*, vol. 35, no. 9, pp. 152– 159, September 1992, special issue on analysis and modeling in software development.

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

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

[4] ——, *Patterns in Java, volume 2*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[5] ——, *Java Enterprise Design Patterns*. New York, NY, USA: John Wiley & Sons, Inc., 2002.

[6] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd ed. 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: A Pattern Language for Distributed Computing*. West Sussex, England: John Wiley & Sons Ltd., 2007, vol. 4.

[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 PTR, 2005.

[13] L. DiPippo and C. D. Gill, *Design Patterns for Distributed Real-Time Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 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, June 2006.

[18] N. Nija Shi and R. Olsson, "Reverse engineering of design patterns from Java source code," in *Proc. of ASE'06, Tokyo, Japan*. IEEE Computer Society, September 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. [Online]. Available: http://www.inf.ed.ac.uk/~stark/autvdp.html

[20] D. Maplesden, J. Hosking, and J. Grundy, "Design pattern modelling and instantiation using DPML," in *Proceedings of the Fortieth International Conference on Tools Pacific (TOOLS Pacific 2002)*. Darlinghurst, Australia: Australian Computer Society, Inc., 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. Arabnia and H. Reza, Eds., vol. II. Las Vegas Nevada, USA: CSREA Press, June 25-28 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, August 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, March 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 COMPSAC'09*. Seattle, Washington, USA: IEEE Computer Society, July 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, iSBN-13: 978-0470409084.

[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, April 1-4 2008, pp. 274–278.

[28] B. Venners, "How to use design patterns: A conversation with Erich Gamma, part i," 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 *?UML? 2001 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, March 2001, pp. 77C–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, April 2010.

[32] H. Zhu and I. Bayley, "An algebra of design pattern composition," *ACM Transactions on Software Engineering and Methdology*, In press.

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

[34] T. Mikkonen, "Formalizing design patterns," in *Proc. of ICSE'98, Kyoto, Japan*. IEEE CS, April 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 ECOOP99*, 1999.

[36] A. Lauder and S. Kent, "Precise visual specification of design patterns," in *Lecture Notes in Computer Science Vol. 1445*, ECOOP'98. 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*, November 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, July-August 2003.

[39] I. Bayley and H. Zhu, "Formalising design patterns in predicate logic," in *5th IEEE International Conference on Software Engineering and Formal Methods*. 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*, vol. 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] 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, October 2001, pp. 574 – 579.

[44] 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, October 5-9 1997, pp. 218–228.

[45] J. Vlissides, "Notation, notation, notation," *C++ Report*, April 1998.

[46] 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.

[47] 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, doi:10.1016/j.jvlc.2011.03.003.

[48] 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, April 2000, pp. 279–287.

[49] T. Taibi, "Formalising design patterns composition," *Software, IEE Proceedings*, vol. 153, no. 3, pp. 126–153, June 2006.

[50] J. Dong, T. Peng, and Y. Zhao, "Automated verification of security pattern compositions," *Information and Software Technology*, vol. 52, no. 3, p. 274C295, March 2010.

[51] ——, "On instantiation and integration commutability of design pattern," *The Computer Journal*, vol. 54, no. 1, pp. 164–184, January 2011.

[52] I. Bayley and H. Zhu, "On the composition of design patterns," in *Proceedings of the Eighth International Conference on Quality Software (QSIC 2008)*. Oxford, UK: IEEE Computer Society, Aug. 2008, pp. 27–36.

[53] ——, "A formal language of pattern composition," in *Proceedings of The 2nd International Conference on Pervasive Patterns (PATTERNS 2010)*. Lisbon, Portugal: XPS (Xpert Publishing Services), Nov. 2010, pp. 1–6.

[54] H. Zhu and I. Bayley, "Laws of pattern composition," in *Proceedings of 12th International Conference on Formal Engineering Methods (ICFEM 2010)*, ser. LNCS, vol. 6447. Shanghai, China: Springer, Nov. 17-19 2010, pp. 630–645.

[55] B. Huston, "The effects of design pattern application on metric scores," *Journal of Systems and Software*, vol. 58, no. 3, pp. 261–269, September 2001.

[56] 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.

[57] 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.

[58] 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, April 2012.

[59] T. Taibi and D. C. L. Ngo, "Formal specification of design pattern combination using BPSL," *Information and Software Technology*, vol. 45, no. 3, pp. 157–170, March 2003.

[60] A. H. Eden, *Codecharts: Roadmaps and Blueprints for Object-Oriented Programs*. Hoboken, New Jersey: Wiley-Blackwell, 2011.

[61] 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. 128, 2013.

[62] 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. 211222, Feb. 2014.

[63] 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.

[64] ——, "An institution theory of formal meta-modelling in graphically extended BNF," *Frontiers of Computer Science*, vol. 6, no. 1, pp. 40–56, 2012.

[65] 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.

[66] 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.

[67] 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, p. 821844, 2010.

APPENDIX: 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 RFMRR=1 RBMRR=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
      > 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 VI-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 RFMRR=1 RBMRR=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:Rew: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_14946l.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-----------------------
```