An Algebra of Design Patterns

Hong Zhu and Ian Bayley
Oxford Brookes University

In a pattern-oriented software design process, design decisions are made by selecting and instantiating appropriate patterns, and composing them together. In our previous work, we enabled these decisions to be formalised by defining a set of operators on patterns with which instantiations and compositions can be represented. In this paper, we investigate the algebraic properties of these operators. We provide and prove a complete set of algebraic laws so that equivalence between pattern expressions can be proven. Furthermore, we define an always-terminating normalisation of pattern expressions to a canonical form, which is unique modulo equivalence in first-order logic.

By a case study, the pattern-oriented design of an extensible request-handling framework, we demonstrate two practical applications of the algebraic framework. Firstly, we can prove the correctness of a finished design with respect to the design decisions made and the formal specification of the patterns. Secondly, we can even derive the design from these components.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.10 [**Software**]: Design—*Methodologies*

General Terms: Design

Additional Key Words and Phrases: Design Patterns, Formal method, Software design methodology, Pattern composition, Algebra, Equational reasoning

1. INTRODUCTION

Design patterns are codified reusable solutions to recurring design problems [Gamma et al. 1995; Alur et al. 2003]. In the past two decades, much research on software design patterns have been reported in the literature. Many such patterns have been identified, documented, catalogued [Gamma et al. 1995; Alur et al. 2003; Grand 2002b; 1999; 2002a; Fowler 2003; Hohpe and Woolf 2004; Buschmann et al. 2007b; Voelter et al. 2004; Schumacher et al. 2005; Steel 2005; DiPippo and Gill 2005; Douglass 2002; Hanmer 2007], and formally specified [Alencar et al. 1996; Mikkonen 1998; Taibi et al. 2003; Gasparis et al. 2008; Bayley and Zhu 2010b]. Numerous software tools have been developed for detecting patterns in reverse engineering and instantiating patterns for software design [Niere et al. 2002; Hou and Hoover 2006; Nija Shi and Olsson 2006; Blewitt et al. 2005; Mapelsden et al. 2002; Dong et al. 2007; Kim and Lu 2006; Kim and Shen 2007; 2008; Zhu et al.

Note: This paper is an extended and revised version of the conference paper [Zhu and Bayley 2010] presented at ICFEM'2010.
Authors' address: Prof. Hong Zhu and Dr. Ian Bayley, Department of Computing and Communication Technologies, Faculty of Technology, Design and Environment, Oxford Brookes University, Wheatley Campus, Oxford OX33 1HX, UK, Tel: +44 (1865) 484580, Fax: +44 (1865) 484545, email: hzhu@brookes.ac.uk, ibayley@brookes.ac.uk.

2009; Zhu et al. 2009]. Although each pattern is documented and specified separately, they are usually to be found composed with each other with overlaps except in trivial cases [Riehle 1997]. Thus, pattern composition plays a crucial role in the effective use of design knowledge.

The composition of design patterns have been studied by many authors, for example, in [Buschmann et al. 2007a; Riehle 1997]. Visual notations such as the *Pattern:Role* annotation, and a forebear based on Venn diagrams, have been proposed by Vlissides [Vlissides 1998] and widely used in practice. They indicate where, in a design, patterns have been applied so that their compositions are comprehensible. These notations are implemented by Dong *et al.* [Dong et al. 2007] for computer-aided visualisation by defining appropriate UML profiles. Their tool, deployed as a web service, identifies pattern applications, and does so by displaying stereotypes, tagged values, and constraints. Such information is delivered dynamically with the movement of the user's mouse cursor on the screen. Their experiments show that this delivery on demand helps to reduce the information overload faced by designers. More recently, Smith [Smith 2011] proposed the *Pattern Instance Notation* (PIN), to visually represent the composition of patterns in a hierarchical manner. Most importantly, he also recognised that multiple instances of roles needed to be better expressed and he devised a suitable graphic notation for this.

However, the existing research on pattern compositions is mostly informal, though much has been done by others to formalise the patterns themselves [Alencar et al. 1996; Mikkonen 1998; Lauder and Kent 1998; Taibi et al. 2003; Eden 2001; Gasparis et al. 2008; Bayley and Zhu 2010b]. These approaches use many different formalisms but the basic ideas underlying them are similar. In particular, a specification of a pattern usually consists of statements about the common structural features and, sometimes, the behavioural features of its instances. The structural features are typically specified by assertions of the existence of certain types of components in the pattern, with the configuration of the elements described in terms of the static relationship between them. The behavioural features are normally defined by assertions on the temporal order of the messages exchanged between these components.

Although such formalisations make possible a systematic investigation of design pattern compositions in a formal setting, few authors have done so. Two that have are Dong *et al.*, who appear to have been the first, and Taibi and Ngo.

Dong *et al.* define a composition of two patterns as a pair of *name mappings*. Each mapping "associates the names of the classes and objects declared in a pattern with the classes and objects declared in the composition of this pattern and other patterns" [Dong et al. 2000; 1999; Dong et al. 2004]. This approach can be regarded as formalisation of the "Pattern:Role" graphic notation. They also demonstrate that structural and behavioural properties of patterns instances can be inferred even after composition. Recently, in [Dong et al. 2011], they studied the commutability of pattern instantiation with pattern integration, which is their term for composition. A pattern instantiation was defined as a mapping from names of various kinds of elements in the pattern to classes, attributes, methods *etc.* in the instance. An integration of two patterns was defined as a mapping from the set union of the names of elements in the two patterns into the names of the elements in the resulting pattern. This formal definition of integration is mathematically equivalent to the multiple name mapping approach.

Taibi and Ngo [Taibi and Ngo 2003; Taibi 2006] took an approach very similar to this, but instead of defining mappings for pattern compositions and instantiations, they use substitution to directly rename the variables that represent pattern elements. For instantiation, the variables are renamed to constants, whereas for composition, they are renamed to new variables. The composition of two patterns is then the logical conjunction of the predicates that specify the structural and behavioural properties of the patterns after substitution.

In [Bayley and Zhu 2008a], we formally defined a pattern composition operator based on the notion of overlaps between the elements in the composed patterns. We distinguished three different kinds of overlaps: one-to-one, one-to-many and many-to-many. Both Dong *et al.* and Taibi's approaches can only compose patterns with one-to-one overlaps. However, the other two kinds of overlaps are often required. For example, if the Composite pattern is composed with the Adapter pattern in such a way that one or more of the leaves are adapted then that is a one-to-many overlap. This cannot be represented as a mapping between names, nor by a substitution or instantiation of variables. However, although this operator is universally applicable, we found in our case study that it is not very flexible for practical uses and its properties are complex to analyse.

In [Bayley and Zhu 2010a], therefore, we revised this previous work of ours and took a radically different approach. Instead of defining a single universal composition operator, we proposed a set of six more primitive operators, with which each sort of composition can then be accurately and precisely expressed. We preserve the advantage of being able to deal with more advanced overlaps. A case study was also reported there to demonstrate the expressiveness of the operators.

In this paper, we now investigate how to reason about compositions, especially how to prove that two pattern expressions are equivalent. As pointed out in [Dong et al. 2011], this is of particular importance in pattern-oriented software design, where design decisions are made by selecting and applying design patterns to address various design problems. If is often desirable to determine whether two alternative decisions result in the same design, especially if one is more abstract and general, and the other one more concrete and easier to understand. We demonstrate that such design decisions can be formally represented using our pattern operators. The subsequent focus on proving the equivalence between pattern expressions leads us to a set of algebraic laws and an always-terminating normalisation process that leads to a canonical form, which is unique subject to logical equivalence. As we demonstrate with a case study of a real-world example, our algebra supports two typical practical scenarios:

—*validation and verification scenario*: Recall that the current practice of pattern-oriented software design is to instantiate and compose design patterns informally by hand, and then present the result in the form of a class diagram annotated with *pattern:role* information. In the *validation and verification* scenario, this design must be checked for correct use of the design patterns. Our algebra can be used to formally prove it to be equivalent to a pattern expression denoting the component design patterns and the decisions made. This means that the result is consistent with the structural and dynamic features of the component.

—*formal derivation scenario*: On the other hand, given a pattern expression, we can, in the *formal derivation* scenario, obtain the design by normalising the expressions to the canonical form. This is directly readable as a concrete design.

This paper has three main contributions. It

—proves a set of algebraic laws that pattern operators obey,

—proves the completeness of the laws, and presents a pattern expression normalisation process that always terminates with unique canonical forms subject to logic equivalence,

—demonstrates with a real-world example the applicability of the algebra to pattern-oriented software design in both the validation/verification and formal derivation scenarios.

These results advance the pattern-oriented software design methodology by improving the rigour in three ways: (a) design decisions are formally represented, (b) a new method is presented for formally proving the correctness of the finished design with respect to these decisions (c) a new method is presented for deriving this design. It also offers the possibility of automated tool support for both of these methods.

The formalism we use to achieve these goals is the same as that in our previous work. In particular, we use the first-order logic induced from the abstract syntax of UML defined in GEBNF [Zhu and Shan 2006; Zhu 2010; 2012] to define both the structural and behavioural features of design patterns. In the same formalism, we have already formally specified the 23 patterns in the classic Gang of Four book [Gamma et al. 1995], hereafter referred to as the *GoF book*. And, we have specified variants too [Bayley and Zhu 2007; 2008b; 2010b]. We have also constructed a prototype software tool to check whether a design represented in UML conforms to a pattern [Zhu et al. 2009; Zhu et al. 2009].

It is worth noting that the definitions of the operations and the algebraic laws proved in this paper are independent of the formalism used to define patterns. Thus, the results can be applied equally well to other formalisms such as OCL [France et al. 2004], temporal logic [Taibi 2006], process algebra [Dong et al. 2010], and so on, but the results may be less readable and the proofs may be more complicated and lengthy. In particular, OCL would need to be applied at the meta-level to assert the existence of the required classes and methods.

The remainder of the paper is organised as follows. Section 2 reviews our approach to formalisation of patterns and lays the theoretical foundation for our proofs. Section 3 outlines the set of operations on design patterns. Section 4 presents the algebraic laws that they obey. Section 5 uses the laws to reason about the equivalence of pattern compositions. Section 6 proves the completeness of the algebraic laws. Section 7 reports a case study with the applications of the theory to a real-world example: the pattern-oriented design of an extensible request-handling framework through pattern composition. Section 8 concludes the paper with a discussion of related works and future work.

2. BACKGROUND

This section briefly reviews our approach to the formal specification of design patterns, to present the background for our formal development of the algebra of design patterns. Our approach is based on meta-modelling in the sense that each pattern is a subset of the design models having certain structural and behavioural features. Readers are referred to [Bayley and Zhu 2007; 2008b; Zhu et al. 2009; Bayley and Zhu 2010b] for details.

2.1 Meta-Modelling in GEBNF

We start by defining the domain of all models with an abstract syntax written in the meta-notation Graphic Extension of BNF (GEBNF) [Zhu and Shan 2006]. GEBNF extends the traditional BNF notation with a 'reference' facility to define the graphical structure of diagrams. In addition, each syntactic element in the definition of a language construct is

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

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

assigned an identifier (called a *field name*) so that a first-order language can be induced from the abstract syntax definition [Zhu 2010; 2012].

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

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

The first line defines a class diagram as consisting of a non-empty set of classes and a collection of three relations on the set. Here $classes$, $assocs$, $inherits$ and $compag$ are field names. Each field name is a function. For example, $classes$ is a function from a $ClassDiag$ to the set of class nodes in the model. Functions $assocs$, $inherits$ and $compag$ are mappings from a class diagram to the sets of association, inheritance and composite/aggregate relations in the model. The non-terminal $Class$ in the definition of End is a reference occurrence. This means that the node at the end of a relation must be an existing class node in the diagram, not a newly-introduced class node. The definitions of the class diagrams and sequence diagrams of UML in GEBNF can be found in [Bayley and Zhu 2010b]. Table I gives the functions used in this paper that are induced from these definitions as well as those that are based on them. A formal more detailed treatment of this can be found in [Bayley and Zhu 2010b].

2.2 Formal Specification of Patterns

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

DEFINITION 1. *(Formal specification of DPs)*
A formal specification of a design pattern is a triple $P = \langle V, Pr_s, Pr_d \rangle$, where Pr_s

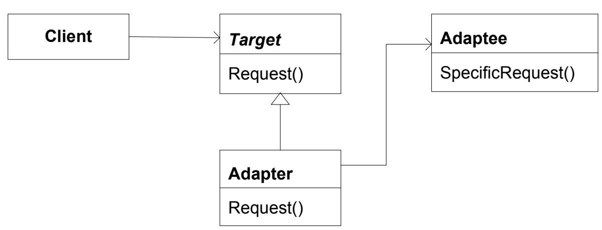
is a predicate on the domain of UML class diagrams that expresses the static structural properties of the pattern and Pr_d *is, similarly, a predicate on the domain of UML sequence diagrams that expresses the dynamic behavioural properties of the pattern;* $V = \{v_1 : T_1, \cdots, v_n : T_n\}$ *is the set of free variables in the predicates* Pr_s *and* Pr_d*. For each* $i \in \{1, \cdots, n\}$, v_i *represents a component of type* T_i *in the pattern. A type can be a basic type* T *of elements,* [1] *such as class, method, attribute, message, lifeline, etc. in the design model, or* $\mathbb{P}(T)$ *(i.e. a power set of* T*), to represent a set of elements of the type* T*, or* $\mathbb{P}(\mathbb{P}(T))$*, etc.*

The semantics of the specification is a closed formula in the following form.

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

Given a pattern specification P*, we write* $Spec(P)$ *to denote the predicate (1) above,* $Vars(P)$ *for the set of variables declared in* V*, and* $Pred(P)$ *for the predicate* $Pr_s \wedge Pr_d$*.* □

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



Specification 1. *(Object Adapter Pattern)*

Components

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

Static Conditions

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

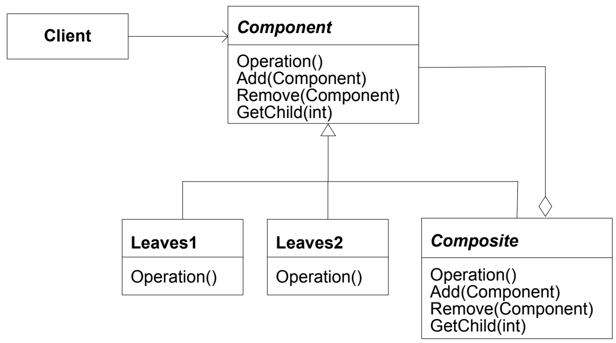
Dynamic Conditions

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

Fig. 1. Specification of Object Adapter Pattern

Figure 2 gives the specification of the $Composite$ pattern, where the class diagram from GoF book [Gamma et al. 1995] only shows one Leaf class while in general there may be many leaves. Both patterns will be used throughout the paper.

[1] Formally speaking, a basic type corresponds to a non-terminal symbol in the GEBNF definition of the modelling language.



SPECIFICATION 2. *(Composite)*

Components

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

Static Conditions

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

Dynamic Conditions

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

Fig. 2. Specification of the Composite Pattern

It is worth noting that the word *model* in classic formal logics has a meaning subtly different from that in software engineering. In mathematical logic, a mathematical theory is represented in the form of set of formulas called axioms, while a model of the theory is a mathematical structure on which the set of formulas are all true. In software engineering, on the other hand, a model is widely regarded as a diagram or a set of diagrams that characterizes the structural and/or dynamic features of a software system, as a means of presenting the design. By defining a pattern as a predicate on software models, the gap between these two notions of models can be bridged. In particular, a software model (such as a UML diagram) is an instance of a pattern when the software model is a structure on which the formal specification (i.e. a logic formula) defining the pattern is true. So these two notions are consistent in our framework and we do not distinguish them in this paper. Readers are referred to [Zhu 2010; 2012] for a formal treatment of software models as mathematical structures.

2.3 Reasoning About Patterns

We often want to show that a concrete design really conforms to a design pattern. This is a far from trivial task for some other formalisation approaches. For us though, the use of predicate logic makes it easy and we formally define the conformance relation as follows.

Let m be a model and pr be a predicate. We write $m \models pr$ to denote that predicate pr is true in model m.

DEFINITION 2. *(Conformance of a design to a pattern)*
Let m be a model and $P = \langle V, Pr_s, Pr_d \rangle$ be a formal specification of a design pattern. The model m conforms *to the design pattern as specified by P if and only if $m \models Spec(P)$.* $\square$

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

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

It is worth noting that the evaluation of $Eva_\alpha(m, pr)$ is independent of the assignment α if pr contains no free variables; thus the subscript α can be omitted. In such cases, the evaluation always terminates with a result being either $True$ or $False$. In fact, $m \models pr$ can be formally defined as $Eva(m, pr) = True$, where, when $Pr = Spec(P)$ is a formal specification of a design pattern P, it contains no free variables. Readers are referred to [Zhu 2010; 2012] for more details of the definition of $Eva_\alpha(m, pr)$.

A software tool has been developed that employs the first order logic theorem prover *SPASS*. With it, proofs of conformance can be performed automatically [Zhu et al. 2009; Zhu et al. 2009].

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

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

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

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

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

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

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

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

(1) $P \preccurlyeq Q$, *if $Spec(P) \Rightarrow Spec(Q)$, and*

(2) $P \approx Q$, *if $Spec(P) \Leftrightarrow Spec(Q)$.* □

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

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

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

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

Therefore, letting P, Q and R be any given patterns, we have the following.

$$P \preccurlyeq P \tag{2}$$
$$(P \preccurlyeq Q) \wedge (Q \preccurlyeq R) \Rightarrow (P \preccurlyeq R) \tag{3}$$
$$FALSE \preccurlyeq P \preccurlyeq TRUE \tag{4}$$

3. OPERATORS ON DESIGN PATTERNS

In this section, we review the set of operators on patterns defined in [Bayley and Zhu 2010a]. The restriction operator was first introduced in [Bayley and Zhu 2008a], where it was called the *specialisation* operator.

DEFINITION 6. *(Restriction operator)*
Let P be a given pattern and c be a predicate such that the set $vars(c)$ of free variables in c is included in $Vars(P)$; i.e. formally $vars(c) \subseteq Vars(P)$. A restriction of P with constraint c, written $P[c]$, is the pattern obtained from P by imposing the predicate c as an additional condition of the pattern. Formally,

(1) $Vars(P[c]) = Vars(P)$,

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

Informally, the predicate c is defined on the components of P; thus it gives an additional constraint on the components and/or on how the components relate to each other. For example, let $||X||$ denote the cardinality of set X. The pattern $Composite_1$ is the variant of the $Composite$ pattern that has only one leaf:

$$Composite_1 \triangleq Composite[\ ||Leaves|| = 1].$$

Many more examples are given in the case studies reported in [Bayley and Zhu 2010a]. A frequently occurring use is in expressions of the form $P[u = v]$ for pattern P and variables u and v of the same type. This is the pattern obtained from P by unifying components u and v and making them the same element.

Note that the instantiation of a variable u in pattern P with a constant a of the same type of variable u can also be expressed by using restriction operator $P[u = a]$. Some researchers also regard restricting the number of elements in a specific component variable of power set type as instantiation of the pattern. This can also be represented by applying the restriction operator as shown in the above example.

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

DEFINITION 7. *(Superposition operator)*
Let P and Q be two patterns. Assume that the names of the components in P and Q are all different, i.e. $Vars(P) \cap Vars(Q) = \emptyset$. The superposition *of P and Q, written $P * Q$, is defined as follows.*

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

Informally, $P * Q$ is the minimal pattern (i.e. that with the fewest components and weakest conditions) containing both P and Q. Note that, although the names of components in $P * Q$ are required to be different, their instances may have overlap. The requirement that components are named differently can always be achieved, for example, by systematically renaming the component variables to make them different and the notation for renaming is as follows.

Let $x \in Vars(P)$ be a component of pattern P and $x' \notin Vars(P)$. The systematic renaming of x to x' is written as $P[x \backslash x']$. Obviously, for all models m, we have that $m \models P \Leftrightarrow m \models P[x \backslash x']$ because $Spec(P)$ is a closed formula. In the sequel, we assume that renaming is made implicitly before two patterns are superposed when there is a naming conflict between them.

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

(1) $Vars(P\#(V \bullet c)) = Vars(P) \cup V$;
(2) $Pred(P\#(V \bullet c)) = Pred(P) \wedge c$. $\Box$

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

Now we define the flatten operator as follows.

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

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

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

$$Composite \Downarrow Leaves\backslash Leaf$$

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

$$(P \Downarrow x_1\backslash x_1') \Downarrow x_2\backslash x_2' \approx (P \Downarrow x_2\backslash x_2') \Downarrow x_1\backslash x_1'. \tag{5}$$

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

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

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

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

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

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

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

A formal proof of the above equation can be found in Section 5.1.

The lift operator was first introduced in our previous work [Bayley and Zhu 2008a].

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

(1) $Vars(P \uparrow X) = \{xs_1 : \mathbb{P}T_1, \cdots, xs_n : \mathbb{P}T_n\}$,

(2) $Pred(P \uparrow X) = \forall x_1 \in xs_1 \cdots \forall x_k \in xs_k \cdot \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot Pred(P)$.

□

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

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

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

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

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

SPECIFICATION 3. *(Lifted Object Adapters Pattern)*

Components

(1) $Targets, Adapters, Adaptees \subseteq classes$,

Conditions

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

Fig. 3. Specification of Lifted Object Adapter Pattern

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

4. ALGEBRAIC LAWS OF THE OPERATIONS

This section studies the algebraic laws that the operators obey.

4.1 Laws of Restriction

The following are the basic algebraic laws that the restriction operator obeys.

THEOREM 1. *For all patterns P, predicates c, c_1 and c_2 such that $vars(c), vars(c_1)$, and $vars(c_2) \subseteq Vars(P)$, the following equalities hold.*

$$P[c_1] \preccurlyeq P[c_2],\ if\ c_1 \Rightarrow c_2 \tag{6}$$

$$P[c_1][c_2] \approx P[c_1 \wedge c_2] \tag{7}$$

$$P[true] \approx P \tag{8}$$
$$P[false] \approx FALSE \tag{9}$$

PROOF.
Let P be any given pattern, and c_1, c_2 be any predicates such that $vars(c_i) \subseteq Vars(P)$, $i = 1, 2$.

For Law (6), by Definition 6, we have $Vars(P[c_i]) = Vars(P)$, and $Pred(P[c_i]) = Pred(P) \wedge c_i$, for $i = 1, 2$. Assume that $c_1 \Rightarrow c_2$. Then, we have that

$$\begin{aligned} Pred(P[c_1]) &= Pred(P) \wedge c_1 \\ &\Rightarrow Pred(P) \wedge c_2 \\ &= Pred(P[c_2]). \end{aligned}$$

So by Lemma 4, we have that $P[c_1] \preccurlyeq P[c_2]$.

Similarly, we can prove that

$$Pred(P[c_1][c_2]) \Leftrightarrow Pred(P[c_1 \wedge c_2],$$

and

$$Pred(P[true]) \Leftrightarrow Pred(P).$$

Thus, Law (7) and (8) are true by Lemma 4.

Law (9) holds because $Pred(P[false]) = Pred(P) \wedge false$, which cannot be satisfied by any models. □

From the above laws, we can prove that the following laws also hold.

COROLLARY 1. *For all patterns P, predicates c, c_1 and c_2, we have that*

$$P[c] \preccurlyeq P \tag{10}$$
$$P[c][c] \approx P[c] \tag{11}$$
$$P[c_1][c_2] \approx P[c_2][c_1] \tag{12}$$

PROOF.
Law (10) is the special case of (6) where c_2 is $true$. That is, we have that

$$\begin{aligned} P[c] &\preccurlyeq P[true] && \langle by\ Law\ (6)\rangle \\ &\approx P && \langle by\ Law\ (8)\rangle \end{aligned}$$

For Law (11), we have that $c \wedge c \Leftrightarrow c$. Thus, it follows from (7).

For Law (12), we have that

$$\begin{aligned} P[c_1][c_2] &\approx P[c_1 \wedge c_2] && \langle by\ Law(7)\rangle \\ &\approx P[c_2 \wedge c_1] && \langle c_1 \wedge c_2 \Leftrightarrow c_2 \wedge c_1\rangle \\ &\approx P[c_2][c_1] && \langle by\ Law(7)\rangle \end{aligned}$$

□

4.2 Laws of Superposition

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

THEOREM 2.
For all patterns P and Q, we have that

$$(P * Q) \preccurlyeq P \quad (13)$$

$$Q \preccurlyeq P \Rightarrow P * Q \approx Q \quad (14)$$

PROOF.
Let P and Q be patterns with

$$Vars(P) \cap Vars(Q) = \emptyset. \quad (15)$$

Assume that

$$\begin{aligned} Vars(P) &= \{x_1 : T_1, \ldots, x_m : T_m\}, \\ Vars(Q) &= \{y_1 : T'_1, \ldots, y_n : T'_n\}. \end{aligned}$$

Then, we have that

$$Vars(P * Q) = \{x_1 : T_1, \ldots, x_m : T_m, y_1 : T'_1, \ldots, y_n : T'_n\}.$$

For Law (13), we have that

$$\begin{array}{lll} & Spec(P * Q) & \\ = & \exists x_1 : T_1 \cdots x_m : T_m, y_1 : T'_1 \cdots y_n : T'_n \cdot (Pred(P) \wedge Pred(Q)) & \langle by\ Def.\ 1\rangle \\ \Leftrightarrow & \exists x_1 : T_1 \cdots x_m : T_m \cdot Pred(P) \wedge \exists y_1 : T'_1 \cdots y_n : T'_n \cdot Pred(Q) & \langle by\ (15)\rangle \\ \Rightarrow & \exists x_1, \ldots, x_m \cdot Pred(P) & \langle by\ logic\rangle \\ = & Spec(P) & \langle by\ Def.\ 1\rangle \end{array}$$

Thus, by Lemma 3, we have that $(P * Q) \preccurlyeq P$.
For Law (14), assume that $Q \preccurlyeq P$. By Lemma 3(1), we have that

$$Spec(Q) \Rightarrow Spec(P). \quad (16)$$

Therefore, we have that

$$\begin{array}{lll} & Spec(P * Q) & \\ = & \exists x_1 : T_1 \cdots x_m : T_m, y_1 : T'_1 \cdots y_n : T'_n \cdot (Pred(P) \wedge Pred(Q)) & \langle by\ Def.\ 1\rangle \\ \Leftrightarrow & \exists x_1 : T_1 \cdots x_m : T_m \cdot Pred(P) \wedge \exists y_1 : T'_1 \cdots y_n : T'_n \cdot Pred(Q) & \langle by\ (15)\rangle \\ = & Spec(P) \wedge Spec(Q) & \langle by\ Def.\ 1\rangle \\ = & Spec(Q) & \langle by\ (16)\rangle \end{array}$$

That is, $P * Q \approx Q$. □

From Theorem 2, we can prove that *TRUE* and *FALSE* patterns are the identity and zero element of superposition operator, which is also idempotent.

COROLLARY 2. *For all patterns P and Q, we have that*

$$P * TRUE \approx P \quad (17)$$

$$P * FALSE \approx FALSE \quad (18)$$

$$P * P \approx P \quad (19)$$

PROOF.
Law (17) follows from Law (14), since $TRUE$ is top in $\preccurlyeq$ according to (4).
Law (18) also follows from Law (14), since $FALSE$ is bottom in $\preccurlyeq$ according to (4).
Law (19) follows Law (14), since $\preccurlyeq$ is reflexive according to (3). □

The following theorem proves that the superposition operator is commutative and associative.

THEOREM 3. *For all patterns P, Q and R, we have that*

$$P * Q \approx Q * P \tag{20}$$
$$(P * Q) * R \approx P * (Q * R) \tag{21}$$

PROOF. The proofs of Law (20) and (21) are very similar to the proof of Theorem 2. Details are omitted for the sake of space. □

4.3 Laws of Extension

From now on, the proofs of algebraic laws will be omitted unless it is not so obvious.

THEOREM 4.
Let P and Q be any given patterns, X and Y be any sets of component variables that are disjoint to $Vars(P)$ and to each other, c_1 and c_2 be any given predicates such that $vars(c_1) \subseteq Vars(P) \cup X$ and $vars(c_2) \subseteq Vars(P) \cup Y$. The extension operation has the following properties.

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

□

From Law (25) and (26), we have the following.

COROLLARY 3. *For all patterns P, we have the equality*

$$P\#(\emptyset \bullet True) \approx P, \tag{28}$$

and for all sets X of variables,

$$P\#(X \bullet False) \approx FALSE. \tag{29}$$

□

4.4 Laws of Flattening and Generalisation

We first generalise the definitions of flattening and generalisation operators such that for all patterns P,

$$P \Uparrow \emptyset \approx P, \tag{30}$$
$$P \Downarrow \emptyset \approx P. \tag{31}$$

We have the following laws for the flattening and generalisation operators.

THEOREM 5.

Let P be any given pattern, $X, Y \subseteq Vars(P)$ and $X \cap Y = \emptyset$. We have that

$$(P \Downarrow X) \Downarrow Y \approx (P \Downarrow Y) \Downarrow X \quad (32)$$
$$(P \Downarrow X) \Downarrow Y \approx P \Downarrow (X \cup Y) \quad (33)$$
$$(P \Uparrow X) \Uparrow Y \approx (P \Uparrow Y) \Uparrow X \quad (34)$$
$$(P \Uparrow X) \Uparrow Y \approx P \Uparrow (X \cup Y) \quad (35)$$

□

We now study the algebraic laws that involve more than one operator.

4.5 Laws Connecting Superposition with Other Operators

The following theorem gives a law about restriction and superposition.

THEOREM 6. *For all predicates c such that $vars(c) \subseteq Vars(P)$, we have that*

$$P[c] * Q \approx (P * Q)[c]. \quad (36)$$

□

Note that an instantiation of a pattern can be represented as an expression that uses only the restriction operator. Furthermore, when a pattern composition only has one-to-one and many-to-many overlaps, the composition can be represented as an expression that only involves restriction and superposition operators. Such a composition is called as *pattern integration* [Dong et al. 2011]. From Theorem 6, we can prove the following law, which is equivalent to the commutability of instantiation and integration of patterns [Dong et al. 2011].

COROLLARY 4. *(Commutability of Pattern Instantiation and Integration)*
For all patterns P and Q, and all predicates C_I such that $vars(C_I) \subseteq Vars(P)$ and predicate C_C such that $vars(C_C) \subseteq Vars(P) \cup Vars(Q)$, we have that

$$(P[C_I] * Q)[C_C] \approx (P * Q)[C_C][C_I]$$

PROOF.

$$\begin{aligned}(P[C_I] * Q)[C_C] &\approx ((P * Q)[C_I])[C_C] && \langle by\ Law\ (36)\rangle \\ &\approx (P * Q)[C_C][C_I] && \langle by\ Law\ (12)\rangle\end{aligned}$$

□

Since one interpretation of $P[C_I]$ is as the instantiation of pattern P with restriction C_I, and integration is superposition followed by restriction, the corollary states that if we first instantiate a pattern, and then integrate it with another pattern, then that is equal to integrating the patterns first and then instantiating them. In other words, the instantiation and integration are commutable if the restriction and superposition operators are applied properly.

In the same way, the following theorems state the commutability of generalisation/ flattening with superposition. They can be used to prove the commutabilities of various pattern compositions that involve one-to-many overlaps.

THEOREM 7. *For all $X \subseteq Vars(P)$, we have that*

$$(P \Uparrow X) * Q \approx (P * Q) \Uparrow X, \quad (37)$$
$$(P \Downarrow X) * Q \approx (P * Q) \Downarrow X. \quad (38)$$

PROOF. For the sake of simplicity, we give the proof for the case when X is a singleton; i.e. $X = \{x\}$. The general case can be proved by induction on the size of X.

For equation (37), assume that $Vars(P) \cap Vars(Q) = \emptyset$ and $xs \notin Vars(P) \cup Vars(Q)$. By the definitions of the $*$ and $\Uparrow$ operators, we have that

$$\begin{array}{lll} & Vars((P \Uparrow x\backslash xs) * Q) & \\ = & Vars(P \Uparrow x\backslash xs) \cup Vars(Q) & \langle by\ Def.\ 7\rangle \\ = & ((Vars(P) - \{x : T\}) \cup \{xs : \mathbb{P}(T)\}) \cup Vars(Q) & \langle by\ Def.\ 10\rangle \\ = & (Vars(P) \cup Vars(Q)) - \{x : T\} \cup \{xs : \mathbb{P}(T)\} & \langle by\ set\ theory\rangle \\ = & Vars((P * Q) \Uparrow x\backslash xs) & \langle by\ Def.\ 7\ and\ 10\rangle \end{array}$$

And,

$$\begin{array}{lll} & Pred((P \Uparrow X) * Q) & \\ = & Pred(P \Uparrow X) \wedge Pred(Q) & \langle by\ Def.\ 7\rangle \\ = & (\forall x \in xs \cdot Pred(P)) \wedge Pred(Q) & \langle by\ Def.\ 10\rangle \\ \Leftrightarrow & \forall x \in xs \cdot (Pred(P) \wedge Pred(Q)) & \langle by\ first\ order\ logic\rangle \\ = & Pred((P * Q) \Uparrow x\backslash xs) & \langle by\ Def.\ 7\ and\ 10\rangle \end{array}$$

Therefore, by Lemma 4, equation (37) holds.

The proof of equation (38) is very similar to the above. It is omitted for the sake of space. □

Combining the above laws with the laws about generalisation and flattening, we have the following corollary.

COROLLARY 5. *Let P and Q be any patterns, and $X \subseteq Vars(P) \cup Vars(Q)$. The following equations hold.*

$$(P * Q) \Uparrow X \approx (P \Uparrow X_P) * (Q \Uparrow X_Q), \tag{39}$$

$$(P * Q) \Downarrow X \approx (P \Downarrow X_P) * (Q \Downarrow X_Q), \tag{40}$$

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

PROOF.

By the definition of operator $*$, we have that $Vars(P) \cap Vars(Q) = \emptyset$. Thus, $X_P \cap X_Q = \emptyset$. Note that $X = X_P \cup X_Q$. Therefore, for Law (39) we have that

$$\begin{array}{rcll} (P * Q) \Uparrow X & \approx & (P * Q) \Uparrow (X_P \cup X_Q) & \\ & \approx & (P * Q) \Uparrow X_P \Uparrow X_Q & \langle by\ Law\ (35)\rangle \\ & \approx & ((P \Uparrow X_P) * Q) \Uparrow X_Q & \langle by\ Law\ (37)\rangle \\ & \approx & (P \Uparrow X_P) * (Q \Uparrow X_Q) & \langle by\ Law\ (37)\rangle \end{array}$$

For Law (40), the proof is similar to the proof of Law (39), but using Law (38) rather than (37). □

To prove the commutability between lifting and superposition, we first introduce a new notation.

Let $X = \{x_1 : T_1, \cdots, x_n : T_n\}$. We write $X^{\uparrow}$ to denote the set $\{xs_1 : \mathbb{P}(T_1), \cdots, xs_n : \mathbb{P}(T_n)\}$.

THEOREM 8.
Let $X \subseteq Vars(P)$, we have that

$$(P \uparrow X) * Q \approx ((P * Q) \uparrow X) \Downarrow Vars(Q)^{\uparrow}. \tag{41}$$

PROOF.
Let $V_P = Vars(P) = \{x_1 : T_1, \cdots, x_n : T_n\}$, $X = \{x_1 : T_1, \cdots, x_k : T_k\}, 1 \leq k < n$, $V_Q = Vars(Q) = \{y_1 : R_1, \cdots, y_m : R_m\}$.
By the definitions of $*$ and $\uparrow$, we have that

$$Vars((P \uparrow X) * Q) = Vars(P \uparrow X) \cup Vars(Q) = V_P^{\uparrow} \cup V_Q. \tag{42}$$

$$Vars((P * Q) \uparrow X) = (V_P \cup V_Q)^{\uparrow}) = V_P^{\uparrow} \cup V_Q^{\uparrow} \tag{43}$$

Therefore, we have that

$$\begin{array}{lll} & Vars((P * Q) \uparrow X) \Downarrow Vars(Q)^{\uparrow}) & \\ = & (Vars((P * Q) \uparrow X) - Vars(Q)^{\uparrow}) \cup Vars(Q), & \langle by\ Def.\ 9\rangle \\ = & (V_P^{\uparrow} \cup V_Q^{\uparrow} - V_Q^{\uparrow}) \cup Vars(Q), & \langle by\ (43)\rangle \\ = & V_P^{\uparrow} \cup Vars(Q), & \langle by\ V_P \cap V_Q = \emptyset\rangle \\ = & Vars((P \uparrow X) * Q). & \langle by (42)\rangle \end{array}$$

By the definitions of $*$ and $\uparrow$, we also have that

$$\begin{array}{l} Pred((P \uparrow X) * Q) = Pred(P \uparrow X) \wedge Pred(Q) \\ = (\forall x_1 \in xs_1 \cdots \forall x_k \in xs_k \cdot \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot Pred(P)) \wedge Pred(Q) \\ \Leftrightarrow \forall x_1 \in xs_1 \cdots x_k \in xs_k \cdot \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot (Pred(P) \wedge Pred(Q)) \end{array}$$

Similarly, we have

$$\begin{array}{l} Pred((P * Q) \uparrow X) = \forall x_1 \in xs_1 \cdots \forall x_k \in xs_k \cdot \\ \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot \exists y_1 \in ys_1 \cdots \exists y_m \in ys_m \cdot (Pred(P) \wedge Pred(Q)) \end{array}$$

By the definition of $\Downarrow$, we have that

$$\begin{array}{l} Pred((P * Q) \uparrow X) \Downarrow Vars(Q)^{\uparrow}) = Pred(P * Q) \uparrow X)[ys_1 \backslash \{y_1'\}, \cdots ys_m \backslash \{y_m'\}] \\ \quad \Leftrightarrow \forall x_1 \in xs_1 \cdots x_k \in xs_k \cdot \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot \\ \qquad \exists y_1 \in \{y_1'\} \cdots \exists y_m \in \{y_m'\} \cdot (Pred(P) \wedge Pred(Q)) \\ \quad \Leftrightarrow \forall x_1 \in xs_1 \cdots x_k \in xs_k \cdot \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot \\ \qquad (Pred(P) \wedge \exists y_1 \in \{y_1'\} \cdots \exists y_m \in \{y_m'\} \cdot Pred(Q)) \\ \quad \Leftrightarrow \forall x_1 \in xs_1 \cdots x_k \in xs_k \cdot \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot (Pred(P) \wedge Pred(Q)) \\ \quad = Pred((P \uparrow X) * Q). \end{array}$$

Therefore, by Lemma 4, the theorem is true. □

4.6 Laws Connecting Generalisation, Flattening and Lifting

Generalisation, flattening and lifting are the three operators that change the structure of the pattern. They are connected to each other by the following algebraic laws.

THEOREM 9.
For all patterns P, all sets of variables $X, Y \subseteq Vars(P)$ and $X' \cap Vars(P) = \emptyset$, we have that

$$P \Uparrow X \approx (P \uparrow X) \Downarrow (V - X^{\uparrow}) \tag{44}$$

$$(P \Uparrow X \backslash X') \Downarrow (X' \backslash X) \approx P \tag{45}$$

$$(P \Downarrow X \backslash X') \Uparrow X' \backslash X) \approx P \tag{46}$$

$$(P \uparrow x) \Downarrow V \approx P, \tag{47}$$

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

4.7 Laws Connecting Restriction to Generalisation, Flattening and Lifting

THEOREM 10.
Let P *be any given pattern,* $c(x_1, \cdots, x_k)$ *be any given predicate such that* $vars(c) = \{x_1 : T_1, \cdots, x_k : T_k\} \subseteq Vars(P)$. *Let* $X \subseteq Vars(P)$ *be a set of variables such that*

(1) $vars(c) \cap X = \{x_1, \cdots, x_m\}$, $m \leq k$;

(2) $X - vars(c) = \{y_1, \cdots, y_u\}$; *and*

(3) $Vars(P) - (X \cup vars(c)) = \{z_1, \cdots, z_v\}$.

We have that

$$P[c] \Uparrow X \ \approx\ (P \Uparrow X)[c_{\Uparrow}] \tag{48}$$

$$P[c] \uparrow X \ \approx\ (P \uparrow X)[c_{\uparrow}] \tag{49}$$

$$P[c] \Downarrow X \ \approx\ (P \Downarrow X)[c_{\Downarrow}] \tag{50}$$

where
$c_{\Uparrow} = \forall x_1 \in xs_1, \cdots, \forall x_m \in xs_m \cdot c$, *and* $\{xs_1 : \mathbb{P}(T_1), \cdots, xs : \mathbb{P}(T_m)\}$;
$c_{\Downarrow} = c(\{x_1'\}, \cdots, \{x_m'\}, x_{m+1}, \cdots, x_k)$, $x_i' : T_i'$ *and* $T_i = \mathbb{P}(T_i')$ *for* $i = 1, \cdots, m$;

$$\begin{aligned} c_{\uparrow} \ =\ & \forall x_1 \in xs_1 \cdots \forall x_n \in xs_m \cdot \forall y_1 \in ys_1 \cdots \forall y_u \in ys_u \cdot \\ & \exists x_{m+1} \in xs_{n+1} \cdots \exists x_n \in xs_n \cdot \exists z_1 \in zs \cdots z_v \in zs_v \cdot (Pred(P) \wedge c). \quad \square \end{aligned}$$

The proof of the theorem is very similar to the proof of Theorem 8, but lengthy and tedious. Thus, it is omitted for the sake of space.

COROLLARY 6.
Let P *be any given pattern,* $X, Y \subseteq Vars(P), X \cap Y = \emptyset$, *and* c *be any predicate such that* $vars(c) \subseteq X \cup Y$. *We have that*

$$((P[c] \uparrow X) \Downarrow Y^{\uparrow}) \approx ((P \uparrow X) \Downarrow Y^{\uparrow})[c'], \tag{51}$$

where
$c' = \forall x_1 \in xs_1, \cdots, \forall x_m \in xs_m \cdot c$,
$vars(c) \cap X = \{x_1 : T_1, \cdots, x_m : T_m\}$, *and* $xs_i : \mathbb{P}(T_i)$ *for* $i = 1 \cdots m$. $\square$

PROOF.
By Law (49), we have that

$$(P[c] \uparrow X) \approx (P \uparrow X)[c_{\uparrow}]$$

where
$c_{\uparrow} = \forall x_1 \in xs_1 \cdots \forall x_n \in xs_n \forall y_1 \in ys_1 \cdots \forall y_u \in ys_u \cdot$
$\exists x_{m+1} \in xs_{n+1} \cdots \exists x_n \in xs_n \exists z_1 \in zs \cdots z_v \in zs_v \cdot (Pred(P) \wedge c)$.
Because $vars(c) \subseteq X \cup Y$ and $X \cap Y = \emptyset$, and by Law (50), we have that

$$(P \uparrow X)[c_{\uparrow}] \Downarrow Y \approx ((P \uparrow X) \Downarrow Y)[(c_{\uparrow})_{\Downarrow})]$$

where, assuming that $Y = \{y_{u'+1}, \cdots, y_u\} \cup \{x_{m+1}, \cdots, x_n\} \cup \{z_{v'+1}, \cdots, z_v\}$, we have

$$\begin{aligned}
(c_{\uparrow})_{\Downarrow} \;=\;& \forall x_1 \in xs_1 \cdots \forall x_n \in xs_m \cdot \\
& \forall y_1 \in ys_1 \cdots \forall y_{u'} \in ys_{u'} \cdot \\
& \forall y_{u'+1} \in \{y_{u'+1}\} \cdots \forall y_u \in \{y_u\} \cdot \\
& \exists x_{m+1} \in \{x_{m+1}\} \cdots \exists x_n \in \{x_n\} \cdot \\
& \exists z_1 \in zs_1 \cdots z_{v'} \in zs_{v'} \cdot \\
& \exists z_{v'+1} \in \{z_{v'+1}\} \cdots \exists z_v \in \{z_v\} \cdot (Pred(P) \wedge c) \\
=\;& \forall x_1 \in xs_1 \cdots \forall x_n \in xs_m \cdot \\
& \forall y_1 \in ys_1 \cdots \forall y_{u'} \in ys_{u'} \cdot \\
& \exists z_1 \in zs_1 \cdots z_{v'} \in zs_{v'} \cdot (Pred(P) \wedge c)
\end{aligned}$$

Because $vars(c) \cap \{y_1, \cdots, y_{u'}, z_1, \cdots z_{v'}\} = \emptyset$, the above predicate is equivalent to

$$\begin{aligned}
& \forall x_1 \in xs_1 \cdots \forall x_n \in xs_m \cdot c \wedge \\
& \forall x_1 \in xs_1 \cdots \forall x_n \in xs_m \cdot \\
& \forall y_1 \in ys_1 \cdots \forall y_{u'} \in ys_{u'} \cdot \\
& \exists z_1 \in zs_1 \cdots z_{v'} \in zs_{v'} \cdot Pred(P)
\end{aligned}$$

By definition of lifting and flattening, we have that

$$\begin{aligned}
Pred(P \uparrow X \Downarrow Y) \;=\;& \forall x_1 \in xs_1 \cdots \forall x_n \in xs_m \cdot \\
& \forall y_1 \in ys_1 \cdots \forall y_{u'} \in ys_{u'} \cdot \\
& \exists z_1 \in zs_1 \cdots \exists z_{v'} \in zs_{v'} \cdot (Pred(P))
\end{aligned}$$

Therefore, $(P \uparrow X \Downarrow Y)[(c_{\uparrow})_{\Downarrow}] \approx (P \uparrow X \Downarrow Y)[c']$. The theorem is true. □

4.8 Laws Connecting Extension to the Other Operators

The following theorem gives the algebraic laws that relate the extension operators to the others.

THEOREM 11. *(Laws of Extension Operator)*
Let P and Q be any given patterns, $X = \{x_1 : T_1, \cdots, x_k : T_k\}$ be any given set of variables such that $X \cap Vars(P) = \emptyset$, and c be any given predicate with free variables in $(Vars(P) \cup X)$. The following equations hold. [2]

$$P\#(X \bullet c) \approx P[\exists X \cdot c] \tag{52}$$

$$P \Downarrow (xs \backslash x) \approx P\#(\{x : T\} \bullet (xs = \{x\}),\ where\ xs : \mathbb{P}(T) \in Vars(P) \tag{53}$$

$$P \Uparrow x \backslash xs \approx P\#(\{xs : \mathbb{P}(T)\} \bullet (\forall x \in xs \cdot Pred(P)) \wedge x \in xs),\ where\ x : T \in Vars(P) \tag{54}$$

$$P[c] \approx P\#(\emptyset \bullet c) \tag{55}$$

$$P \approx TRUE\#(Vars(P) \bullet Pred(P)) \tag{56}$$

$$P * Q \approx P\#(Vars(Q) \bullet Pred(Q)) \tag{57}$$

$$P \uparrow X \approx P\#(Vars(P \uparrow X) \bullet Pred(P \uparrow X)) \tag{58}$$

[2]Notation: For the sake of space, here we write $\exists X \cdot c$ to denote $\exists x_1 : T_1 \cdots \exists x_k : T_k \cdot c$.

PROOF.
For the sake of space, we prove only the first three equations. The proofs for the other equations are very similar, thus omitted.

For Law (52), by the definitions of the extension operator and the restriction operator, we have that

$$\begin{aligned} Spec(P\#(X \bullet c)) &= \exists(Vars(P) \cup X) \cdot (Pred(P) \wedge c) \\ &\Leftrightarrow \exists Vars(P) \cdot (Pred(P) \wedge (\exists X \cdot c)) \\ &\Leftrightarrow Spec(P[\exists X \cdot c]) \end{aligned}$$

For Law (53), let $Vars(P) = \{x : \mathbb{P}(T), x_1 : T_1, \cdots, x_n : T_n\}$ and $Pred(P) = p(x, x_1, \cdots, x_n)$. By the definitions of the extension operator and the flatten operator, we have that

$$\begin{aligned} & Spec(P \Downarrow (xs \backslash x)) \\ & = \exists x : T \cdot \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot p(\{x\}, x_1, \cdots, x_n) \\ & \Leftrightarrow \exists xs : \mathbb{P}(T) \cdot \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot (p(xs, x_1, \cdots, x_n) \wedge \exists x : T \cdot (xs = \{x\})) \\ & \Leftrightarrow Spec(P\#(\{x : T\} \bullet (xs = \{x\}))) \end{aligned}$$

For Law (54), let $Vars(P) = \{x : T, x_1 : T_1, \cdots, x_n : T_n\}$. By the definitions of extension operator and the generalisation operator, we have that

$$\begin{aligned} & Spec(P \Uparrow x \backslash xs) \\ = \; & \exists xs : \mathbb{P}(T) \cdot \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot (\forall x \in xs \cdot Pred(P)) \\ \Leftrightarrow \; & \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot \exists xs : \mathbb{P}(T) \cdot (\forall x \in xs \cdot Pred(P)) \\ \Leftrightarrow \; & \exists x : T \cdot \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot (Pred(P) \wedge \exists xs : \mathbb{P}(T) \cdot (\forall x \in xs \cdot Pred(P))) \\ \Leftrightarrow \; & Spec(P\#\{xs : \mathbb{P}(T)\} \bullet \forall x \in xs \cdot Pred(P)) \end{aligned}$$

□

For example, from the equations given above, we can prove that the following equations hold.

COROLLARY 7.

$$P\#(X \bullet c_1 \wedge c_2) \approx P\#(X \bullet c_1)[c_2] \tag{59}$$

$$P\#(X \bullet c) \approx P\#(X \bullet true)[c] \tag{60}$$

PROOF.
For Law (59), we have that

$$\begin{aligned} P\#(X \bullet c_1)[c_2] &\approx P\#(X \bullet c_1)\#(\emptyset \bullet c_2), \;\; \langle by (55) \rangle \\ &\approx P\#(X \cup \emptyset \bullet c_1 \wedge c_2), \;\; \langle by (27) \rangle \\ &\approx P\#(X \bullet (c_1 \wedge c_2) \end{aligned}$$

For Law (60), it is a special case of Law (59) with $c_1 = true$. □

5. PROVING THE EQUALITY OF PATTERN COMPOSITIONS: EXAMPLES

In the previous section, we have already used algebraic laws to prove many equations of pattern composition expressions. In this section, we further demonstrate the use of the laws to prove the equivalence of pattern compositions with real examples of patterns.

5.1 Different Definitions of the Composite Pattern

In Section 3, we have seen a number of definitions of the $Composite$ and $Composite_1$ patterns. They are as follows.

$$Composite_1 \triangleq Composite[||Leaves|| = 1] \tag{61}$$
$$Composite_1 \triangleq Composite \Downarrow Leaves\backslash Leaf \tag{62}$$
$$Composite \approx Composite_1 \Uparrow Leaf\backslash Leaves \tag{63}$$

We now first prove that these two definitions of the $Composite_1$ pattern are equivalent. That is, the following equation is true.

$$Composite[||Leaves|| = 1] \approx Composite \Downarrow Leaves\backslash Leaf.$$

PROOF.

$$\begin{array}{ll} Composite \Downarrow Leaves\backslash Leaf & \\ \approx Composite\#(\{Leaf : class\}) \bullet Leaves = \{Leaf\}), & \langle by\ Law(53)\rangle \\ \approx Composite[\exists Leaf : class \cdot (Leaves = \{Leaf\})], & \langle by\ Law(52)\rangle \\ \approx Composite[||Leaves|| = 1], & \langle by\ set\ theory\rangle \end{array}$$

□

We can also prove that the equation (63) holds when the definition of $Composite_1$ in equation (62) is substituted into the right-hand-side of (63). That is,

$$Composite \approx (Composite \Downarrow Leaves\backslash Leaf) \Uparrow Leaf\backslash Leaves.$$

This is quite trivial because it follows from Law (46) immediately.

Similarly, by substituting the definition of $Composite$ into equation (62), we can see that the following is also true.

$$Composite_1 \approx (Composite_1 \Uparrow Leaf\backslash Leaves) \Downarrow Leaves\backslash Leaf.$$

This follows Law (45) immediately.

5.2 Composition of Composite and Adapter

In this subsection, we consider two different ways in which the Composite and Adapter patterns can be composed and then prove that the two compositions are equivalent.

A. First composition

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

$$\begin{array}{l} OneAdaptedLeaf \triangleq \\ \quad (Adapter * Composite)[Target \in Leaves][Target\backslash AdaptedLeaf] \end{array}$$

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

$$(OneAdaptedLeaf \uparrow (AdaptedLeaf \backslash AdaptedLeaves)) \Downarrow \{Composites, Components, Leaveses\} \quad (64)$$

By the definitions of the operators, we derive the predicates of the pattern in the specification given in Figure 4, after some simplification of the first-order logic.

SPECIFICATION 4. *(ManyAdaptedLeaves)*

Components

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

Static Conditions

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

Dynamic Conditions

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

Fig. 4. Specification of Composition of Composite and Adapter Patterns

B. Second composition

An alternative way of expressing the composition is first to lift the $Adapter$ with $Target$ as the key and then to superposition it to the $Composite$ patterns so that many leaves can be adapted. This approach is illustrated in Figure 5. Formally,

$$ManyAdaptedLeaves \triangleq$$

$$(((Adapter \uparrow (Target\backslash Targets)) * Composite)[Targets \subseteq Leaves]$$
$$[Targets\backslash AdaptedLeaves]$$

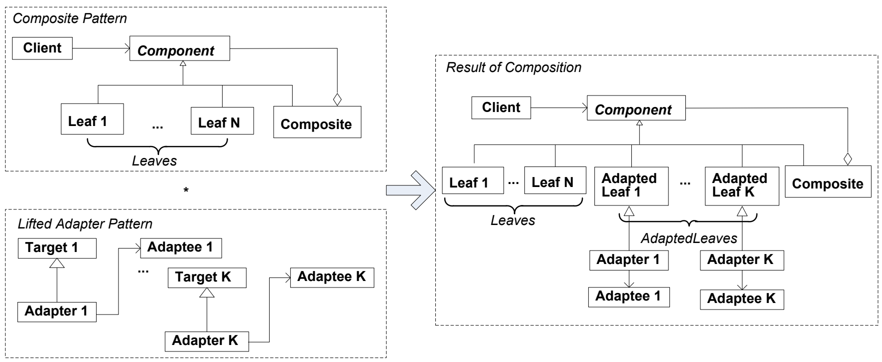


Fig. 5. Composition of Adapter and Composite

C. Proof of equivalence

We now apply the algebraic laws to prove that expression (64) is equivalent to the definition of $ManyAdaptedLeaves$.

First, by (41), we can rewrite $ManyAdaptedLeaves$ to the following expression, where $V_C = \{Composites, Components, Leaveses\}$.

$$((Adapter * Composite) \uparrow (Target\backslash Targets) \Downarrow V_C$$
$$[Targets \subseteq Leaves]\,[Targets\backslash AdaptedLeaves] \quad (65)$$

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

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

by (51), we have that

$$((Adapter * Composite) \uparrow (Target\backslash Targets) \Downarrow V_C)\,[Targets \subseteq Leaves] \quad (66)$$
$$\approx ((Adapter * Composite)[Target \in Leaves]) \uparrow (Target\backslash Targets) \Downarrow V_C$$

Now, renaming $Target$ to $AdaptedLeaf$ and $Targets$ to $AdaptedLeaves$ in expression on the right-hand-side of (66), we have the following.

$$((Adapter * Composite)[Target \in Leaves][Target\backslash AdaptedLeaf])$$
$$\uparrow (AdaptedLeaf\backslash AdaptedLeaves) \Downarrow V_C \quad (67)$$

By substituting the definition of $OneAdaptedLeaf$ into (67), we obtain (64).

6. THE COMPLETENESS OF THE ALGEBRAIC LAWS

This section addresses the completeness question about the set of laws given in Section 4. In particular, given two equivalent pattern expressions, is it always possible to transform one pattern expression to another by applying the algebraic laws?

In general, a set of algebraic laws is complete if they satisfy the following four conditions:

(1) every expression can be transformed into a canonical form by applying the algebraic laws as rewriting rules;

(2) the process of transformation always terminates within a finite number of steps;

(3) the canonical form of an expression is unique subject to certain equivalence relation;

(4) any two expressions that are equivalent if and only if their canonical forms are equivalent and the equivalence between the canonical forms can be determined by certain mechanism.

If a set of algebraic laws satisfies these conditions, one can always transform two expressions into their canonical forms by applying the algebraic laws as rewriting rules and then determine the equivalence between them by checking if their canonical forms are equivalent.

Here, a pattern expression is constructed by applying the operators to specific design patterns. Formally, let $E(\Gamma_1, \cdots, \Gamma_k)$, $k \geq 0$, be a pattern expression that contains variables $\Gamma_1, \cdots, \Gamma_k$ that range over patterns, and $P_1, \cdots, P_k$ be specific design patterns. We write $E(P_1, \cdots, P_k)$ to represent the pattern obtained by replacing Γ_n with P_n in $E(\Gamma_1, \cdots, \Gamma_k)$ for $n \in \{1, \cdots, k\}$, if it is syntactically valid.

DEFINITION 12. *(Equivalence of Pattern Expressions)*
Let $E_1(\Gamma_1, \cdots, \Gamma_k)$ and $E_2(\Gamma_1, \cdots, \Gamma_k)$, $k \geq 0$, be two pattern expressions that contain variables $\Gamma_1, \cdots, \Gamma_k$ that range over patterns. E_1 is equivalent *to E_2, written $E_1 \approx E_2$, if for all specific patterns $P_1, \cdots, P_k$, we have that for all valid models m,*

$$m \models E_1(P_1, \cdots, P_k) \Leftrightarrow m \models E_2(P_1, \cdots, P_k). \quad \square$$

6.1 Canonical Form

To prove the completeness of the algebraic laws, we first prove the following lemma, stating that pattern expressions have canonical forms.

LEMMA 5. *(Canonical Form Lemma)*
For all pattern expressions E, we can always transform it, by applying the algebraic laws for a finite number of times, into the form

$$TRUE\#(V \bullet c),$$

where V is a set of variables and c is a predicate on those variables.

Informally, the canonical form of a pattern expression can be obtained by repeated applications of the laws of extension given in Section 4 and the laws that connect extension with the other operators, i.e. Laws (53)-(60). Each left-to-right application of laws (53)-(58) will reduce the number of non-extension operators in the expression by one, eventually reaching zero. An expression that contains multiple uses of the extension operator can then always be reduced to one by applying the laws of extensions and equations (59) and (60). Eventually, it will reduce to the canonical form. A formal inductive proof of the lemma follows.

PROOF. Let E be any given pattern expression. We now prove by induction on the number n of applications of operators that E contains.

(a) *Base*: When the number n of operators in E equals 0, i.e. E contains no pattern operator, E is either a variable that ranges over patterns, or a constant (i.e., a given pattern), such as $Composite$, $Adapter$, etc. In both cases, by Law (56), we have that

$$E = TRUE\#(Vars(E) \bullet Pred(E)).$$

Thus, the Lemma is true for the base case $n = 0$.

(b) *Induction Hypothesis*: Assume that for all $n \leq N$ the lemma is true, where $N \geq 0$.

(c) *Induction*: We now prove that for all pattern expressions E that contains $N + 1$ applications of the operators, the lemma is also true. We have six cases, according to which operator is applied at the top level:

Case $*$: Suppose $E = E_1 * E_2$ for some pattern expressions E_1 and E_2, where the numbers of applications of the operators in E_1 and E_2 must be less than the number of applications of the operators in E. By the induction hypothesis, we have that both E_1 and E_2 can be transformed into the form $TRUE\#(V \bullet c)$ by applying the algebraic laws. Let E_i be transformed into $TRUE\#(V_i \bullet c_i)$, $i = 1, 2$. Then, we have that

$$\begin{array}{lll} E & \approx TRUE\#(V_1 \bullet c_1) * TRUE\#(V_2 \bullet c_2) & \langle by\ induction\ hypothesis \rangle \\ & \approx TRUE[\exists V_1 \cdot c_1] * TRUE[\exists V_2 \cdot c_2] & \langle by\ Law\ (52) \rangle \\ & \approx (TRUE * TRUE)[\exists V_1 \cdot c_1][\exists V_2 \cdot c_2] & \langle by\ Theorem\ 6 \rangle \\ & \approx TRUE[\exists V_1 \cdot c_1][\exists V_2 \cdot c_2] & \langle by\ Law (17) \rangle \\ & \approx TRUE[(\exists V_1 \cdot c_1) \wedge (\exists V_2 \cdot c_2)] & \langle by\ Law\ (7) \rangle \\ & \approx TRUE[\exists V_1 \cdot \exists V_2 \cdot (c_1 \wedge c_2)] & \langle V_1 \cap V_2 = \emptyset, by\ Def.\ 7 \rangle \\ & \approx TRUE[\exists (V_1 \cup V_2) \cdot (c_1 \wedge c_2)] & \langle by\ logic \rangle \\ & \approx TRUE\#((V_1 \cup V_2) \bullet (c_1 \wedge c_2)) & \langle by\ Law\ (52) \rangle \end{array}$$

Therefore, the lemma is true in this case.

Case $[-]$: Suppose that $E = E'[c]$ for some pattern expression E' and predicate c, where the number of operator applications contained in E' must be N. Thus, by the induction hypothesis, we have that E' can be transformed into the canonical form by applying the algebraic laws, i.e. $E' \approx TRUE\#(V' \bullet c')$. Then, we have that

$$\begin{array}{lll} E & = E'[c] & \\ & \approx TRUE\#(V' \bullet c')[c] & \langle by\ induction\ hypothesis \rangle \\ & \approx TRUE\#(V' \bullet c' \wedge c) & \langle by\ Law\ (59) \rangle \end{array}$$

Therefore, the lemma is true in this case.

Case $\Uparrow$: Suppose that $E = E' \Uparrow X \backslash XS$ for some pattern expression E' and $X \subseteq Vars(E')$, where the number of operator applications contained in E' must be N. Thus, by the induction hypothesis, we have that E' can be transformed into the canonical form by applying the algebraic laws, i.e. $E' \approx TRUE\#(V' \bullet c')$. Let $X = \{x_1 : T_1, \cdots x_k : T_k\}$ and $XS = \{xs_1 : \mathbb{P}(T_1), \cdots, xs_k : \mathbb{P}(T_k)\}$. Then, we have that

$$\begin{array}{lll} E & = E' \Uparrow X \backslash XS & \\ & \approx TRUE\#(V' \bullet c') \Uparrow X \backslash XS & \langle hypothesis \rangle \\ & \approx TRUE\#(V' \bullet c')\#(XS \bullet \forall x_1 \in xs_1 \cdots x_k \in xs_k \cdot c') & \langle by\ Law\ (54) \rangle \\ & \approx TRUE\#(V' \cup XS \bullet c' \wedge \forall x_1 \in xs_1 \cdots x_k \in xs_k \cdot c') & \langle by\ Law\ (26) \rangle \end{array}$$

Therefore, the lemma is true in this case.

Case $\Downarrow$: Suppose that $E = E' \Downarrow XS \backslash X$ for some pattern expression E' and $XS \subseteq Vars(E')$, where the number of applications of the operators contained in E' must be N. Thus, by the induction hypothesis, we have that E' can be transformed into the canonical form by applying the algebraic laws, i.e. $E' \approx TRUE\#(V' \bullet c')$. Let $XS = \{XS_1 :$

$\mathbb{P}(T_1), \cdots, xs_k : \mathbb{P}(T_k)\}$ and $X = \{x_1 : T_1, \cdots x_k : T_k\}$. Then, we have that

$$\begin{aligned} E &= E' \Downarrow XS \backslash X \\ &\approx TRUE\#(V' \bullet c') \Downarrow XS \backslash X & \langle hypothesis \rangle \\ &\approx TRUE\#(V' \bullet c')\#(X \bullet xs_1 = \{x_1\} \wedge \cdots \wedge xs_k = \{x_k\}) & \langle by\ Law\ (53) \rangle \\ &\approx TRUE\#(V' \cup X \bullet c' \wedge (xs_1 = \{x_1\} \wedge \cdots xs_k = \{x_k\})) & \langle by\ Law\ (26) \rangle \end{aligned}$$

Therefore, the lemma is true in this case.

Case $\uparrow$: Suppose that $E = E' \uparrow X \backslash XS$ for some pattern expression E' and $X \subseteq Vars(E')$, where the number of applications of the operators contained in E' must be N. Thus, by the induction hypothesis, we have that E' can be transformed into the canonical form by applying the algebraic laws, i.e. $E' \approx TRUE\#(V' \bullet c')$. Let $V' = \{x_1 : T_1, \cdots, x_n : T_n\}$ and $X = \{x_1 : T_1, \cdots x_k : T_k\}$, where $0 < k \leq n$. Then, we have that

$$\begin{aligned} E &= E' \uparrow X \backslash XS \\ &\approx TRUE\#(V' \bullet c') \uparrow X \backslash XS & \langle hypothesis \rangle \\ &\approx TRUE\#(V' \bullet c')\#(V'^{\uparrow} \bullet c'') & \langle by\ Law\ (58) \rangle \\ &\approx TRUE\#(V' \cup V'^{\uparrow} \bullet c' \wedge c'') & \langle by\ Law\ (26) \rangle \end{aligned}$$

where $c'' = \forall x_1 \in xs_1 \cdots \forall x_k \in xs_k \cdot \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot c'$. Therefore, the lemma is true in this case.

Case $\#$: Suppose that $E = E'\#(V \bullet c)$ for some pattern expression E' and $V \not\subseteq Vars(E')$ and predicate c, where the number of applications of the operators contained in E' must be N. Thus, by the induction hypothesis, we have that E' can be transformed into the canonical form by applying the algebraic laws, i.e. $E' \approx TRUE\#(V' \bullet c')$. Then, we have that

$$\begin{aligned} E &= E'\#(V \bullet c) \\ &\approx TRUE\#(V' \bullet c')\#(V \bullet c) & \langle hypothesis \rangle \\ &\approx TRUE\#(V \cup V' \bullet (c \wedge c')) & \langle by\ Law\ (26) \rangle \end{aligned}$$

Therefore, the lemma is also true in this case.

Since the six cases cover all possible forms of pattern expressions, the Lemma is true for all expressions that contain $N + 1$ applications of the operators. Consequently, by the induction proof principle, the Lemma is true for every expression that contains a finite number of operator applications. $\square$

6.2 The Completeness Theorem

We can now prove the following uniqueness property of the canonical forms of pattern expressions.

THEOREM 12. *(Completeness of The Algebraic Laws)*
Let E_1 and E_2 be any two given pattern expressions, with canonical forms $TRUE\#(V_1 \bullet c_1)$ and $TRUE\#(V_2 \bullet c_2)$, respectively. Pattern expressions $E_1 \approx E_2$ if and only if $\exists V_1 \cdot c_1 \Leftrightarrow \exists V_2 \cdot c_2$.

PROOF.
Let $E_1 \approx E_2$. By Lemma 5, both E_1 and E_2 can always be transformed into canonical form, say,

$$E_1 \approx TRUE\#(V_1 \bullet Pr_1),$$
$$E_2 \approx TRUE\#(V_2 \bullet Pr_2).$$

Then, by Law (52), we have that

$$E_1 \approx TRUE[\exists V_1 \cdot Pr_1],$$
$$E_2 \approx TRUE[\exists V_2 \cdot Pr_2].$$

By Definition 2, we have that for all models m, $m \models E_i$ if and only if $m \models \exists V_i \cdot Pr_i$, for $i = 1, 2$. Therefore, $E_1 \approx E_2$ if and only if $\exists V_1 \cdot Pr_1 \Leftrightarrow \exists V_2 \cdot Pr_2$. $\square$

The above theorem and the lemma prove that the algebraic laws are complete in sense outlined at the start of this section.

(1) in Lemma 5, we have proved that for every pattern expression E, we can transform it into a canonical form $TRUE\#(V \bullet c)$.
(2) the proof of Lemma 5 shows that the canonical transformation process always terminates within a finite number of steps.
(3) given a canonical form $TRUE\#(V \bullet c)$, we call the logic formula $\exists V \cdot c$ the *logic representation* of the canonical form. Theorem 12 proves that the canonical form of a pattern expression is always unique subject to logic equivalence between the logic representation of the canonical form.
(4) Theorem 12 also shows that the mechanism to determine the equivalence of the canonical forms is logic inference in first order logic and set theory.

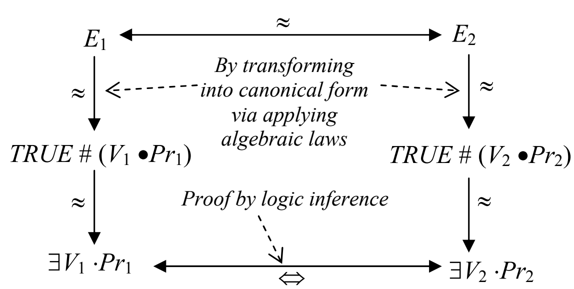


Fig. 6. Illustration of the Proof of Completeness Theorem

Theorem 12 and Lemma 5 also suggest a general process for proving the equivalence between two pattern expressions. As illustrated in Figure 6, pattern expressions E_1 and E_2 are transformed to their canonical forms by applying the algebraic laws separately. Then, the equivalence between them is proved by logic inference in first-order predicate logic and set theory to determine whether the logic representations of their canonical forms are logically equivalent.

The following example demonstrates how to prove the equivalence of two pattern expressions using this process. It also shows that it is some times impossible to avoid relying on set theory and first-order logic to determine the equivalence between two pattern expressions.

EXAMPLE 1. We prove that the following equation holds for all patterns P and all variables $X : \mathbb{P}(T)$ in $Vars(P)$.

$$P[\|X\| = 1] \approx P \Downarrow X$$

PROOF. For the left-hand-side of the equation, we have that

$$\begin{aligned} P[\|X\|=1] &\approx TRUE\#(Vars(P)\bullet Pred(P))[\|X\|=1] && \langle by\ Law\ (56)\rangle \\ &\approx TRUE\#(Vars(P)\bullet (Pred(P)\wedge(\|X\|=1))) && \langle by\ Law\ (59)\rangle \end{aligned}$$

For the right-hand-side, we have that

$$\begin{aligned} P\Downarrow X &\approx P\#(\{x:T\}\bullet X=\{x\}) && \langle by\ Law\ (53)\rangle \\ &\approx P[\exists x:T\cdot(X=\{x\})] && \langle by\ Law\ (52)\rangle \\ &\approx TRUE\#(Vars(P)\bullet Pred(P))[\exists x:T\cdot(X=\{x\})] && \langle by\ Law\ (56)\rangle \\ &\approx TRUE\#(Vars(P)\bullet (Pred(P)\wedge\exists x:T\cdot(X=\{x\}))) && \langle by\ Law\ (59)\rangle \end{aligned}$$

Because, in formal predicate logic and set theory, we can prove that

$$\|X\|=1\Leftrightarrow\exists x:T\cdot(X=\{x\}),$$

we have that the equation holds for all patterns P. □

7. CASE STUDY: DESIGN OF A REQUEST-HANDLING FRAMEWORK

In this section, we present an application of the formal algebra to the development of an extensible request-handling framework through pattern composition. The original experiment was reported in [Buschmann et al. 2007a]. Here, we demonstrate, first, how to represent design decisions as pattern expressions, then how to validate and verify the correct usage of patterns in a manually-prepared design, and finally, how to formally derive a design from the design decisions taken.

7.1 Formal Representation of Design Decisions

In [Buschmann et al. 2007a], the formulation of a request-handling framework was reported as a sequence of design decisions concerning the selection and composition of patterns. The first design problem in this process was to design the structure of the framework in such a way that the requests issued by clients can be objectified. The $Command$ pattern was applied to address this problem. In particular, an abstract class Command declares a set of abstract methods to execute client requests, and a set of ConcreteCommand classes derived from the Command class implements the concrete commands that applications handle. Figure 7 shows the structure of the $Command$ pattern.

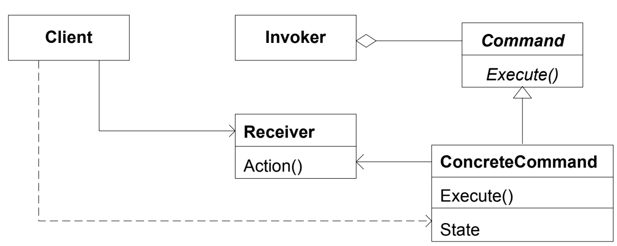


Fig. 7. Structure of Command Pattern

When a client issues a specific request, it instantiates a corresponding ConcreteCommand object and invokes one of its methods inherited from the abstract Command class. The ConcreteCommand object then performs the requested operation on the application and returns the results, if any, to the client. This is a simplified version of the general $Command$ pattern that makes the Client also be the Invoker. This design decision can be formally represented as an expression in our operators on design patterns as follows.

$$RHF_1 \triangleq Command[Invoker = Client, Receiver \backslash Application]$$

To coordinate independent requests from multiple clients, the $CommandProcessor$ pattern shown in Figure 8 is composed with the $Command$ pattern. This composition of patterns can be formally expressed as follows.

$$\begin{aligned} RHF_2 \triangleq\ & RHF_1 * CommandProcessor \\ & [Command = Component \wedge Client = CommandProcessor] \end{aligned}$$

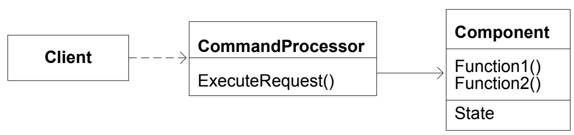


Fig. 8. Structure of Command Processor Pattern

To support the undoing of actions performed in response to requests, the $Memento$ pattern was further composed with the design, since that is a common usage of the pattern. The structure of the $Memento$ pattern is shown in Figure 9. Copies of the state of the application are created by the Originator as instances of the Memento class. The Caretaker maintains the copies by holding the copies over time, and if required, passes them back to the Originator.

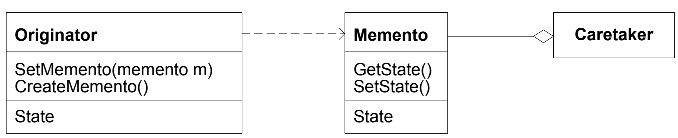


Fig. 9. Structure of Memento Pattern

In the request-handling framework, the originator is the application, whose states are stored in a new component that plays the role of Memento in the $Memento$ pattern. Conceptually, the command is the caretaker that creates mementos before executing a request, maintains these mementos and when necessary passes them back to the application so that the concrete commands can be rolled back when an undo operation is invoked. However, an alternative design decision is to include a separate caretaker class and connect it to the Command class so that every ConcreteCommand object can use an instance of the caretaker class to create, maintain and restore an instance of the Memento class. That design decision can be represented formally as follows.

$$\begin{aligned} RHF_3 \triangleq\ & RHF_2 * Memento \\ & [Originator = Application, Command \longrightarrow Caretaker] \end{aligned}$$

A further item of functionality required for the request-handling framework is a mechanism for logging requests. Different users may want to log the requests different; some may want to log every request, some may want to log just one particular type of requests, and yet more may not want to log any request at all. The design problem is to satisfy all the different logging needs of different users in a flexible and efficient manner. The solution is to apply the $Strategy$ pattern, which is depicted in Figure 10.

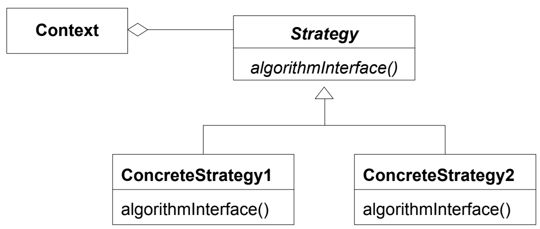


Fig. 10. Structure of Strategy Pattern

The $Strategy$ pattern was applied as follows: the CommandProcessor passes the ConcreteCommand objects it receives to a LoggingContext object that plays the Context role in $Strategy$. This object implements the invariant parts of the logging service and delegates the computation of customer-specific logging aspects to the ConcreteLoggingStrategy object, which plays the role of ConcreteStrategy in the $Strategy$ pattern. An abstract class Logging offers a common protocol for all ConcreteLoggingStrategy classes so that they can be exchanged without modifying LoggingContext. This design can be represented formally as follows.

$$\begin{aligned} RHF_4 \triangleq\ & RHF_3 * Strategy \\ & [Context \backslash LoggingContext, Strategy \backslash Logging, \\ & \ ConcreteStrategies \backslash ConcreteLoggingStrategies] \\ & [CommandProcessor \longrightarrow LoggingContext] \end{aligned}$$

The final step of the design process is to support compound commands. A ConcreteCommand object may be an aggregate of other ConcreteCommand objects that are executed in a particular order. The design pattern that provides this structure is $Composite$. Compound commands can be represented as the composite objects and atomic commands as leaf objects. Thus, we have the following formal expression of the design.

$$\begin{aligned} RHF_5 \triangleq\ & RHF_4 * Composite \\ & [Leaves = ConcreteCommands, Component = Command] \\ & [Composite \backslash CompositeCommand] \end{aligned}$$

An optimisation of the above design is to merge the LoggingContext and CommandProcessor components rather than separating them. The separate Caretaker in the Memento pattern can also be merged into the Command class. Such merging of components is called *pattern interwoven* in [Buschmann et al. 2007a]. The final result is as follows.

$$\begin{aligned} RHF \triangleq\ & RHF_5[Caretaker = Command] \\ & [CommandProcessor = LoggingContext] \end{aligned}$$

7.2 Verification and Validation of Design Result

The current practice in pattern-oriented design is to manually work out the final result of the design process and depict it in a class diagram for the structure. Figure 11 shows the result of the above design process given in [Buschmann et al. 2007a].

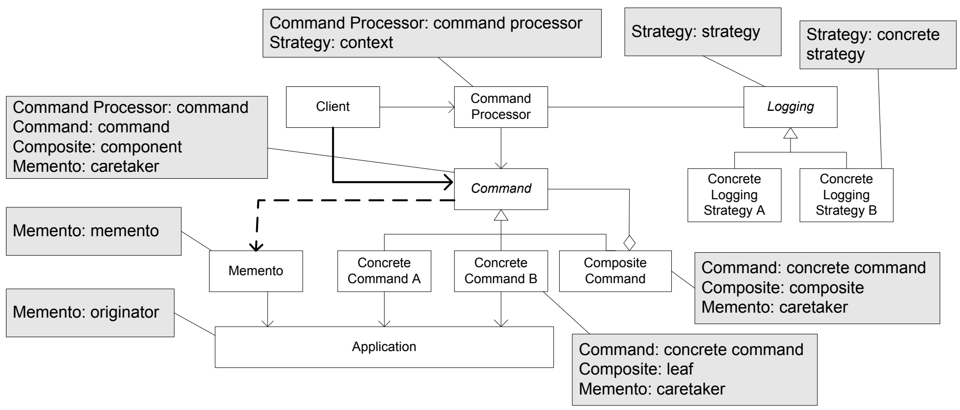


Fig. 11. Original Design of the Request-Handling Framework

This diagram can be directly translated into the following pattern expression.

$$
\begin{aligned}
RHF^o \triangleq\ & Command * CommandProcessor * Memento * Strategy * Composite \\
& [Originator \backslash Application] \\
& [Strategy \backslash Logging] \\
& [ConcreteStrategies \backslash ConcreteLoggingStrategies] \\
& [Context \backslash CommandProcessor] \\
& [(CommandProcessor.Command = Command.Command \\
& \quad = Composite.Component = Memento.Caretaker) \backslash Command] \\
& [Leaves \backslash ConcreteCommands] \\
& [Composite \backslash CompositeCommands]
\end{aligned}
$$

Applying the algebra of design patterns, we can now formally validate and verify the correctness of the above design against the design decisions and the definitions of the patterns by proving the following equation.

$$RHF \approx RHF^o \qquad (68)$$

To decide if equation (68) holds, we substitute the definitions of $RHF_1, ..., RHF_5$ into RHF, simplify the expression by applying the algebraic laws, and obtain the following.

$$
\begin{aligned}
RHF \;\approx\; & Command * CommandProcessor * Memento * Strategy * Composite \\
& [Invoker \backslash Client, Receiver \backslash Application, Strategy \backslash Logging, \\
& \;Context \backslash LoggingContext, Composite \backslash CompositeCommand, \\
& \;ConcreteStrategies \backslash ConcreteLoggingStrategies] \\
& [Command \longrightarrow Caretaker] \\
& [Command = Component \wedge Originator = Application \wedge \\
& \;Command.Client = CommandProcessor \wedge Originator = Application \wedge \\
& \;Leaves = ConcreteCommands \wedge Component = Command \wedge \\
& \;CommandProcessor = LoggingContext \wedge Caretaker = Command]
\end{aligned}
$$

Comparing RHF with RHF^o, we can see that all the restriction predicates in RHF^o are included in RHF, except

$$CommandProcessor.Command = Command.Command,$$

which we believe is a typo since there is no element in the $CommandProcessor$ pattern called Command [Buschmann et al. 2007b]. It should be replaced by the following.

$$CommandProcessor.Component = Command.Command.$$

Moreover, some of the restrictions in RHF are missing from RHF^o. These are:

$$
\begin{array}{lcl}
Application & = & Receiver \\
CommandProcessor & = & Invoker \\
CommandProcessor & = & Command.Client
\end{array}
$$

where $Command.Client$ denotes the Client component in the $Command$ pattern.

Other more serious errors in the diagram in [Buschmann et al. 2007a] are listed in the next section.

7.3 Formal Derivation of Designs

The expressions defining RHF can be transformed into the canonical form by following the normalisation process given in Section 6. This derives the structural and dynamic features of the designed system. Here, we only give the derivation of the structural features of the design. The dynamic features can be derived in the exactly same way, but for the sake of space, they are omitted. The full details of the formal specification of the request-handling framework can be found in [Bayley and Zhu 2011].

First, for the sake of simplicity and space, we only take a small part of the pattern specifications and make the following definitions.

$$
\begin{array}{lll}
Command & \triangleq & TRUE\#(\{Command, Client, Invoker, Receiver : Class, \\
& & \qquad\qquad ConcreteCommands : \mathbb{P}(Class)\} \\
& & \bullet(Client \longrightarrow Command \wedge Invoker \longrightarrow Command \\
& & \quad \forall CC \in ConcreteCommands \cdot (CC \longrightarrow Receiver \wedge \\
& & \qquad CC \longrightarrow\!\!\triangleright\, Command \wedge \neg isAbstract(CC))) \\
ComProc & \triangleq & TRUE\#(\{Client, CommandProcessor, Component : Class\} \\
& & \bullet(Client \longrightarrow CommandProcessor \wedge \\
& & \quad CommandProcessor \longrightarrow Component) \\
Memento & \triangleq & TRUE\#(\{Caretaker, Memento, Originator : Class\} \\
& & \bullet(Caretaker \diamond\!\!\longrightarrow Memento \wedge Originator \longrightarrow memento) \\
Strategy & \triangleq & TRUE\#(\{Context, Strategy : Class, ConcreteStrategies : \mathbb{P}(Class)\} \\
& & \bullet(Context \diamond\!\!\longrightarrow Strategy \wedge isInterface(Strategy) \\
& & \quad \forall CS \in ConcreteStrategies \cdot (CS \longrightarrow\!\!\triangleright\, Strategy))) \\
Composite & \triangleq & TRUE\#(\{Component, Composite : Class, Leaves : \mathbb{P}(Class)\} \\
& & \bullet(isInterface(Component) \wedge Composite \longrightarrow\!\!\triangleright^{*} Component \wedge \\
& & \quad Composite \diamond\!\!\longrightarrow^{+} Component \\
& & \forall Lf \in Leaves \cdot (Lf \longrightarrow\!\!\triangleright\, Component))).
\end{array}
$$

Then, the following can be derived, following the normalisation process by applying the algebraic laws.

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

The result can be graphically presented as in Figure 12.

Comparing Figure 12 with the original diagram of [Buschmann et al. 2007a] shown in Figure 11, we found that the original solution given in [Buschmann et al. 2007a] seems have treated the memento as being created by the caretaker, but in fact it is created by the originator instead. Also, the client should only send requests to CommandProcessor rather than to Command directly. Therefore, the association from Client to Command should be deleted from the original design.

In summary, the case study clearly demonstrates that

(1) design decisions in the application of design patterns can be precisely represented in pattern expressions,

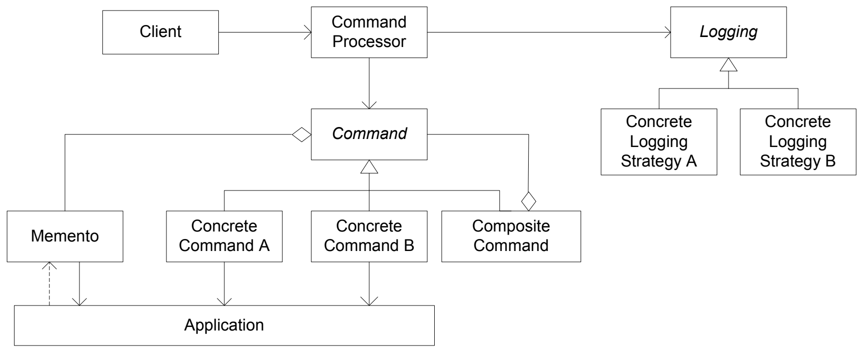


Fig. 12. Derived Design of the Request-Handling Framework

(2) correct uses of design patterns can be formally verified and validated by proving equivalence between pattern expressions. Errors in the manual application of patterns can be detected by disproving the equality between pattern expressions.
(3) moreover, designs can be formally derived from the formal representation of the design decisions through application of the algebra of design patterns. An example of this is using the normalisation process.

8. CONCLUSION

In this paper, we proved a set of algebraic laws that the operators on design patterns obey. We demonstrated their use in proving the equivalence of some pattern compositions. These operators and algebraic laws form a formal calculus of design patterns that enable us to reason about pattern compositions. We have also proved that the set of algebraic laws is complete and we have presented a normalisation process for pattern expressions. We demonstrated the application of the algebra to pattern-oriented software design with a real-world example: the design of an extensible request-handling framework. We demonstrated the applicability of pattern operators to formally and precisely representing design decisions in a pattern-oriented design process.

We also demonstrated the applicability of the algebra in two practical scenarios. In the verification and validation scenario, manual designs are checked against the formal representation of decision decisions in the form of an expression made of pattern compositions and instantiations and the formal specifications of design patterns. In the derivation scenario, designs are formally derived from design decisions and formal specifications of patterns. The work reported in this paper advances the pattern-oriented software design methodology by improving its rigorousness and laying a solid theoretical foundation. It is built on top of the huge amount of research in the literature about software design patterns and their formal specifications. It sheds a new light on the formal and automated software verification and validation at design stage and on the derivation of designs from high level design decisions and design knowledge encoded in design patterns.

Although the calculus is developed in our own formalisation framework, we believe that it can be easily adapted to others, such as that of Eden's approach, which also uses first-order logic but no specification of behavioural features [Gasparis et al. 2008], Taibi's approach, which is a mixture of first-order logic and temporal logic [Taibi et al. 2003], and that of [Lano et al. 1996], etc., and finally, the approaches based on graphic meta-modelling languages, such as RBML [France et al. 2004] and DPML [Mapelsden et al. 2002]. However, the definitions of the operators and proofs of the laws are more concise and readable in our formalism. Dong et al's approach [Alencar et al. 1996; Dong et al. 1999; 2000; Dong et al. 2004; Dong et al. 2007] to the formal specification of patterns is very similar to ours in the way that they also use formal predicate logic to specify the structural and behavioural features of patterns. However, their definition of pattern composition is different from ours. They define pattern compositions and instantiations separately, but both as name mappings. More recently, Dong et al. [Dong et al. 2011] studied the commutability of pattern instantiation and integration, but their results focus on the commutability conditions for instantiation and integration rather than general algebraic laws. Moreover, their definition of pattern instantiation and integration does not cover complicated forms of composition where one-to-many overlaps are needed.

For future work, we are developing automated software tools based on the algebra of design patterns to support pattern-oriented software design. The normalisation process given in the constructive proof of the completeness of the algebraic laws implies that any two pattern compositions can be proved equivalent by using a theorem prover.

REFERENCES

ALENCAR, P. S. C., COWAN, D. D., AND DE LUCENA, C. J. P. 1996. A formal approach to architectural design patterns. In *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods (FME'96)*, M.-C. Gaudel and J. Woodcock, Eds. Lecture Notes In Computer Science. Springer-Verlag, 576 – 594.

ALUR, D., CRUPI, J., AND MALKS, D. 2003. *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd ed. Prentice Hall.

BAYLEY, I. AND ZHU, H. 2007. Formalising design patterns in predicate logic. In *5th IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, London, UK, 25–36.

BAYLEY, I. AND ZHU, H. 2008a. On the composition of design patterns. In *Proceedings of the Eighth International Conference on Quality Software (QSIC 2008)*. IEEE Computer Society, Oxford, UK, 27–36.

BAYLEY, I. AND ZHU, H. 2008b. Specifying behavioural features of design patterns in first order logic. In *32nd IEEE International Conference on Computer Software and Applications (COMPSAC 2008)*. IEEE Computer Society, Turku, Finland, 203–210.

BAYLEY, I. AND ZHU, H. 2010a. A formal language of pattern composition. In *Proceedings of The 2nd International Conference on Pervasive Patterns (PATTERNS 2010)*. XPS (Xpert Publishing Services), Lisbon, Portugal, 1–6.

BAYLEY, I. AND ZHU, H. 2010b. Formal specification of the variants and behavioural features of design patterns. *Journal of Systems and Software 83,* 2 (Feb.), 209–221.

BAYLEY, I. AND ZHU, H. 2011. A formal language for the expression of pattern compositions. *International Journal on Advances in Software 4,* 3-4. (In Press).

BLEWITT, A., BUNDY, A., AND STARK, I. 2005. Automatic verification of design patterns in Java. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*. ACM Press, Long Beach, California, USA, 224–232.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007a. *Pattern-Oirented Software Archiecture: On Patterns and Pattern Languages*. Vol. 5. John Wiley & Sons Ltd.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007b. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Vol. 4. John Wiley & Sons Ltd., West Sussex, England.

DIPIPPO, L. AND GILL, C. D. 2005. *Design Patterns for Distributed Real-Time Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

DONG, J., ALENCAR, P. S., AND COWAN, D. D. 2000. 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)*. IEEE CS Press, Edinburgh, Scotland, 279–287.

DONG, J., ALENCAR, P. S., COWAN, D. D., AND YANG, S. 2007. Composing pattern-based components and verifying correctness. *Journal of Systems and Software 80,* 11 (November), 1755–1769.

DONG, J., ALENCAR, P. S. C., AND COWAN, D. D. 1999. Correct composition of design components. In *Proceedings of the 4th International Workshop on Component-Oriented Programming in conjunction with ECOOP99*.

DONG, J., PENG, T., AND ZHAO, Y. 2010. Automated verification of security pattern compositions. *Information and Software Technology 52,* 3 (March), 274C295.

DONG, J., PENG, T., AND ZHAO, Y. 2011. On instantiation and integration commutability of design pattern. *The Computer Journal 54,* 1 (January), 164–184.

DONG, J., S.C.ALENCAR, P., AND COWAN, D. 2004. A behavioral analysis and verification approach to pattern-based design composition. *Software and Systems Modeling 3*, 262–272.

DONG, J., YANG, S., AND ZHANG, K. 2007. Visualizing design patterns in their applications and compositions. *IEEE Transactions on Software Engineering 33,* 7 (July), 433–453.

DONG, J., ZHAO, Y., AND PENG, T. 2007. 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. CSREA Press, Las Vegas Nevada, USA, 621–627.

DOUGLASS, B. P. 2002. *Real Time Design Patterns: Robust Scalable Architecture for Real-time Systems*. Addison Wesley, Boston, USA.

EDEN, A. H. 2001. Formal specification of object-oriented design. In *International Conference on Multidisciplinary Design in Engineering, Montreal, Canada*.

FOWLER, M. 2003. *Patterns of Enterprise Application Architecture*. Addison Wesley, Boston, USA.

FRANCE, R. B., KIM, D.-K., GHOSH, S., AND SONG, E. 2004. A uml-based pattern specification technique. *IEEE Trans. Softw. Eng. 30,* 3, 193–206.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GASPARIS, E., EDEN, A. H., NICHOLSON, J., AND KAZMAN, R. 2008. The design navigator: charting Java programs. In *Proc. of ICSE'08*. Vol. Companion Volume. 945–946.

GRAND, M. 1999. *Patterns in Java, volume 2*. John Wiley & Sons, Inc., New York, NY, USA.

GRAND, M. 2002a. *Java Enterprise Design Patterns*. John Wiley & Sons, Inc., New York, NY, USA.

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

HANMER, R. S. 2007. *Patterns for Fault Tolerant Software*. Wiley, West Sussex, England.

HOHPE, G. AND WOOLF, B. 2004. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, Boston, USA.

HOU, D. AND HOOVER, H. J. 2006. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering 32,* 6 (June), 404–423.

KIM, D.-K. AND LU, L. 2006. 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)*. IEEE Computer Society, Stanford, California, USA, 47–56.

KIM, D.-K. AND SHEN, W. 2007. An approach to evaluating structural pattern conformance of UML models. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07)*. ACM Press, Seoul, Korea, 1404–1408.

KIM, D.-K. AND SHEN, W. 2008. Evaluating pattern conformance of UML models: a divide-and-conquer approach and case studies. *Software Quality Journal 16,* 3, 329–359.

LANO, K., BICARREGUI, J. C., AND GOLDSACK, S. 1996. Formalising design patterns. In *BCS-FACS Northern Formal Methods Workshop, Ilkley, UK*.

LAUDER, A. AND KENT, S. 1998. Precise visual specification of design patterns. In *Lecture Notes in Computer Science Vol. 1445*. ECOOP'98, Springer, 114–134.

MAPELSDEN, D., HOSKING, J., AND GRUNDY, J. 2002. Design pattern modelling and instantiation using dpml. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*. Australian Computer Society, Inc., 3–11.

MIKKONEN, T. 1998. Formalizing design patterns. In *Proc. of ICSE'98, Kyoto, Japan*. IEEE CS, 115–124.

NIERE, J., SCHÄFER, W., WADSACK, J. P., WENDEHALS, L., AND WELSH, J. 2002. Towards pattern-based design recovery. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2002)*. IEEE CS, Orlando, Florida, USA, 338–348.

NIJA SHI, N. AND OLSSON, R. 2006. Reverse engineering of design patterns from java source code. In *Proc. of ASE'06, Tokyo, Japan*. IEEE Computer Society, 123–134.

RIEHLE, D. 1997. Composite design patterns. In *Proceedings of the 1997 ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. ACM Press, Atlanta, Georgia, 218–228.

SCHUMACHER, M., FERNANDEZ, E., HYBERTSON, D., AND BUSCHMANN, F. 2005. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, West Sussex, England.

SMITH, J. M. 2011. The pattern instance notation: A simple hierarchical visual notation for the dynamic visualization and comprehension of software patterns. *Journal of Visual Languages and Computing 22,* 5 (Oct.), 355–374. doi:10.1016/j.jvlc.2011.03.003.

STEEL, C. 2005. *Applied J2EE Security Patterns: Architectural Patterns & Best Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

TAIBI, T. 2006. Formalising design patterns composition. *Software, IEE Proceedings 153,* 3 (June), 126–153.

TAIBI, T., CHECK, D., AND NGO, L. 2003. Formal specification of design patterns-a balanced approach. *Journal of Object Technology 2,* 4 (July-August).

TAIBI, T. AND NGO, D. C. L. 2003. Formal specification of design pattern combination using BPSL. *Information and Software Technology 45,* 3 (March), 157–170.

VLISSIDES, J. 1998. Notation, notation, notation. *C++ Report*.

VOELTER, M., KIRCHER, M., AND ZDUN, U. 2004. *Remoting Patterns*. John Wiley & Sons, West Sussex, England.

ZHU, H. 2010. 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)*. IEEE CS, Taipei, Taiwan, 95–104.

ZHU, H. 2012. An institution theory of formal meta-modelling in graphically extended bnf. *Frontier of Computer Science 6,* 1 (Jan.), 40–56.

ZHU, H. AND BAYLEY, I. 2010. Laws of pattern composition. In *Proceedings of 12th International Conference on Formal Engineering Methods (ICFEM 2010)*. LNCS, vol. 6447. Springer, Shanghai, China, 630–645.

ZHU, H., BAYLEY, I., SHAN, L., AND AMPHLETT, R. 2009. Tool support for design pattern recognition at model level. In *Proc. of COMPSAC'09*. IEEE Computer Society, Seattle, Washington, USA, 228–233.

ZHU, H. AND SHAN, L. 2006. Well-formedness, consistency and completeness of graphic models. In *Proc. of UKSIM'06*. Oxford, UK, 47–53.

ZHU, H., SHAN, L., BAYLEY, I., AND AMPHLETT, R. 2009. A formal descriptive semantics of UML and its applications. In *UML 2 Semantics and Applications*, K. Lano, Ed. John Wiley & Sons, Inc. ISBN-13: 978-0470409084.

Submitted on 2 Dec. 2011.
Revised version: 5 April 2012.