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.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20XX ACM 0000-0000/20XX/0000-0001 $5.00

ACM Journal Name, Vol. XX, No. XX, XX 20XX, Pages 1–38.
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 it 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

<table>
<thead>
<tr>
<th>ID</th>
<th>Domain</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>Class diagram</td>
<td>The set of class nodes in the class diagram</td>
</tr>
<tr>
<td>assocs</td>
<td>Class diagram</td>
<td>The set of association relations in the class diagram</td>
</tr>
<tr>
<td>inherits</td>
<td>Class diagram</td>
<td>The set of inheritance relations in the class diagram</td>
</tr>
<tr>
<td>compag</td>
<td>Class diagram</td>
<td>The set of composite and aggregate relations in the class diagram</td>
</tr>
<tr>
<td>name</td>
<td>Class node</td>
<td>The name of the class</td>
</tr>
<tr>
<td>attr</td>
<td>Class node</td>
<td>The attributes contained in the class node</td>
</tr>
<tr>
<td>opers</td>
<td>Class node</td>
<td>The operations contained in the class node</td>
</tr>
<tr>
<td>sig</td>
<td>Message</td>
<td>The signature of the message</td>
</tr>
</tbody>
</table>

Functions defined based on induced functions

- \( X \cdashrightarrow Y \): Class \( X \) inherits class \( Y \) directly or indirectly
- \( X \rightarrow Y \): Class \( X \) is an interface
- \( CDR(X) \): Class \( X \) is an interface
- \( sub(X) \): Class \( X \) has a subclass of \( X \)
- \( calls(x,y) \): Operation \( x \) calls operation \( y \)
- \( isAbstract(op) \): Operation \( op \) is abstract
- \( fromClass(m) \): Message \( m \) is sent from
- \( toClass(m) \): Message \( m \) is sent to
- \( X \approx Y \): 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.

\[
\begin{align*}
ClassDiag & ::= \text{classes} : \text{Class}^+, \text{assocs} : \text{Rel}^+, \text{inherits} : \text{Rel}^+, \text{compag} : \text{Rel}^* \\
\text{Class} & ::= \text{name} : \text{String}, [\text{atts} : \text{Property}^*], [\text{opers} : \text{Operation}^*] \\
\text{Rel} & ::= \text{name} : \text{String}, \text{source} : \text{End}, \text{end} : \text{End} \\
\text{End} & ::= \text{node} : \text{Class}, \text{name} : \text{String}, \text{mult} : \text{Multiplicity}
\end{align*}
\]

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 \( \text{classes}, \text{assocs}, \text{inherits} \) and \( \text{compag} \) are field names. Each field name is a function. For example, \( \text{classes} \) is a function from a \( \text{ClassDiag} \) to the set of class nodes in the model. Functions \( \text{assocs}, \text{inherits} \) and \( \text{compag} \) are mappings from a class diagram to the sets of association, inheritance and composite/aggregate relations in the model. The non-terminal \( \text{Class} \) in the definition of \( \text{End} \) 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; \( V = \{ v_1 : T_1, \ldots, v_n : T_n \} \) is the set of free variables in the predicates \( Pr_s \) and \( Pr_d \). For each \( i \in \{1, \ldots, n\}, v_i \) 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 \( \mathcal{P}(T) \) (i.e. a power set of \( T \)), to represent a set of elements of the type \( T \), or \( \mathcal{P}(\mathcal{P}(T)) \), etc.

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

\[
\exists v_1 : T_1 \cdots \exists v_n : T_n \cdot (Pr_s \land Pr_d)
\]  

(1)

Given a pattern specification \( P \), we write \( \text{Spec}(P) \) to denote the predicate (1) above, \( Vars(P) \) for the set of variables declared in \( V \), and \( Pred(P) \) for the predicate \( Pr_s \land Pr_d \).

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.

```
Fig. 1. Specification of Object Adapter Pattern
```

<table>
<thead>
<tr>
<th>Specification 1. (Object Adapter Pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Components</strong></td>
</tr>
<tr>
<td>(1) Target, Adapter, Adaptee \in classes,</td>
</tr>
<tr>
<td>(2) requests \subseteq Target.opers,</td>
</tr>
<tr>
<td>(3) specreqs \subseteq Adaptee.opers</td>
</tr>
<tr>
<td><strong>Static Conditions</strong></td>
</tr>
<tr>
<td>(1) Adapter \rightarrow^+ Target, Adapter \rightarrow^+ Adaptee,</td>
</tr>
<tr>
<td>(2) CDR(Target)</td>
</tr>
<tr>
<td><strong>Dynamic Conditions</strong></td>
</tr>
<tr>
<td>(1) \forall o \in requests \cdot \exists o' \in specreqs \cdot (calls(o, o'))</td>
</tr>
</tbody>
</table>

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.

ACM Journal Name, Vol. XX, No. XX, XX 20XX.
An Algebra of Design Patterns

SPECIFICATION 2. (Composite)

Components

(1) Component, Composite ∈ classes,
(2) Leaves ⊆ classes,
(3) ops ⊆ Component.opers

Static Conditions

(1) ops ≠ ∅
(2) ∀o ∈ ops.isAbstract(o),
(3) ∀l ∈ Leaves · (l →∈+ Component ∧ ¬(l →→+ Component))
(4) isInterface(Component)
(5) Composite →∈+ Component
(6) Composite →→+ Component
(7) CDR(Component)

Dynamic Conditions

(1) any call to Composite causes follow-up calls
∀m ∈ messages · ∃o ∈ ops · toClass(m) = Composite ∧ m.sig ≈ o ⇒
   ∃m’ ∈ messages · calls(m, m’) ∧ m’.sig ≈ m.sig
(2) any call to a leaf does not causes follow-up calls
∀m ∈ messages · ∃o ∈ ops · toClass(m) ∈ Leaves ∧ m.sig ≈ o ⇒
   ¬∃m’ ∈ messages · calls(m, m’) ∧ m’.sig ≈ 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 \( m \models Spec(P) \).

To prove such a conformance we just need to give an assignment \( \alpha \) of variables in \( V \) to elements in \( m \) and evaluate \( Pred(P) \) in the context of \( \alpha \). 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 \( \alpha \).

**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 \( \alpha \) from \( Vars(P) \) to the elements in \( m \) such that \( Eva_\alpha(m, Pred(P)) = true \).

It is worth noting that the evaluation of \( Eva_\alpha(m, pr) \) is independent of the assignment \( \alpha \) if \( pr \) contains no free variables; thus the subscript \( \alpha \) 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 \( m \models q \).

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 \( P \preceq Q \), 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 \( P \preceq Q \) and \( Q \preceq P \).

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. \( P \preceq Q \), if \( \text{Spec}(P) \Rightarrow \text{Spec}(Q) \), and
2. \( P \approx Q \), if \( \text{Spec}(P) \Leftrightarrow \text{Spec}(Q) \). □

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 \( \text{Vars}(P) = \text{Vars}(Q) \). Then \( P \preceq Q \) if \( \text{Pred}(P) \Rightarrow \text{Pred}(Q) \), and \( P \approx Q \) if \( \text{Pred}(P) \Leftrightarrow \text{Pred}(Q) \). □

Specialisation is a pre-order with bottom \( \text{FALSE} \) and top \( \text{TRUE} \) defined as follows.

**Definition 5.** (\( \text{TRUE} \) and \( \text{FALSE} \) patterns)

Pattern \( \text{TRUE} \) is the pattern such that for all models \( m \), \( m \models \text{TRUE} \). Pattern \( \text{FALSE} \) is the pattern such that for no model \( m \), \( m \models \text{FALSE} \). □

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

\[
\begin{align*}
P \preceq P & \quad (2) \\
(P \preceq Q) \land (Q \preceq R) \Rightarrow (P \preceq R) & \quad (3) \\
\text{FALSE} \preceq P & \preceq \text{TRUE} \quad (4)
\end{align*}
\]

### 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 \( \text{vars}(c) \) of free variables in \( c \) is included in \( \text{Vars}(P) \); i.e. formally \( \text{vars}(c) \subseteq \text{Vars}(P) \). 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. \( \text{Vars}(P[c]) = \text{Vars}(P) \),
2. \( \text{Pred}(P[c]) = (\text{Pred}(P) \land c) \). □
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 \( \text{Composite}_1 \) is the variant of the \( \text{Composite} \) pattern that has only one leaf:

\[
\text{Composite}_1 \triangleq \text{Composite} \left[ |\text{Leaves}| = 1 \right].
\]

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. \( \text{Vars}(P) \cap \text{Vars}(Q) = \emptyset \). The superposition of \( P \) and \( Q \), written \( P \star Q \), is defined as follows.

1. \( \text{Vars}(P \star Q) = \text{Vars}(P) \cup \text{Vars}(Q) \);
2. \( \text{Pred}(P \star Q) = \text{Pred}(P) \land \text{Pred}(Q) \).

Informally, \( P \star Q \) 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 \( P \star Q \) 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 \( x \in \text{Vars}(P) \) be a component of pattern \( P \) and \( x' \notin \text{Vars}(P) \). The systematic renaming of \( x \) to \( x' \) is written as \( P[x \leftarrow x'] \). Obviously, for all models \( m \), we have that \( m \models P \Rightarrow m \models P[x \leftarrow x'] \) because \( \text{Spec}(P) \) 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. \( \text{Vars}(P) \cap V = \emptyset \)), and \( c \) be a predicate with variables in \( \text{Vars}(P) \cup V \). The extension of pattern \( P \) with components \( V \) and linkage condition \( c \), written as \( P \#(V \bullet c) \), is defined as follows.

1. \( \text{Vars}(P \#(V \bullet c)) = \text{Vars}(P) \cup V \);
2. \( \text{Pred}(P \#(V \bullet c)) = \text{Pred}(P) \land c \).

For any predicate \( p \), let \( p[x \leftarrow c] \) 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, $xs : P(T)$ be a variable in $\text{Vars}(P)$ and $x : T$ be a variable not in $\text{Vars}(P)$. Then the flattening of $P$ on variable $x$, written $P \downarrow xs \setminus x$, is defined as follows.

1. $\text{Vars}(P \downarrow xs \setminus x) = (\text{Vars}(P) - \{xs : \mathbb{P}(T)\}) \cup \{x : T\}$.
2. $\text{Pred}(P \downarrow xs \setminus x) = \text{Pred}(P)[xs \setminus \{x\}]$. □

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, $\text{Leaves} \subseteq \text{classes}$ in the specification of the $\text{Composite}$ pattern is the same as $\text{Leaves} : \mathbb{P}(\text{classes})$. Applying the flatten operator on $\text{Leaves}$, the $\text{Composite}_1$ pattern can be equivalently expressed as follows.

$$\text{Composite} \downarrow \text{Leaves} \setminus \text{Leaf}$$

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

$$(P \downarrow x_1 \setminus x'_1) \downarrow x_2 \setminus x'_2 \approx (P \downarrow x_2 \setminus x'_2) \downarrow x_1 \setminus x'_1.$$ (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. $X = \{x_1 : \mathbb{P}(T_1), \ldots, x_n : \mathbb{P}(T_n)\} \subseteq \text{Vars}(P)$, $n \geq 1$ and $X' = \{x'_1 : T_1, \ldots, x'_n : T_n\}$ such that $X' \cap \text{Vars}(P) = \emptyset$. Then we write $P \downarrow X \setminus X'$ to denote $P \downarrow x_1 \setminus x'_1 \downarrow \cdots \downarrow x_n \setminus x'_n$.

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 $P \downarrow x \setminus x'$. 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 $\text{Vars}(P)$ and $xs : \mathbb{P}(T)$ be a variable not in $\text{Vars}(P)$. Then the generalisation of $P$ on variable $x$, written $P \uparrow x \setminus xs$, is defined as follows.

1. $\text{Vars}(P \uparrow x \setminus xs) = (\text{Vars}(P) - \{x : T\}) \cup \{xs : \mathbb{P}(T)\}$,
2. $\text{Pred}(P \uparrow x \setminus xs) = \forall x \in xs \cdot \text{Pred}(P)$. □

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 $\text{Composite}_1$ on the component $\text{Leaf}$, we can obtain the pattern $\text{Composite}$. Formally,

$$\text{Composite} \approx \text{Composite}_1 \uparrow \text{Leaf} \setminus \text{Leaves}.$$ 

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 $\text{Vars}(P) = \{x_1 : T_1, \ldots, x_n : T_n\}$, $n > 0$. Let $X = \{x_1, \ldots, x_k\}$, $1 \leq k < n$, be a subset of $\text{Vars}(P)$. The lifting of $P$ with $X$ as the key, written $P \uparrow X$, is the pattern defined as follows.
(1) \( \text{Vars}(P \uparrow X) = \{xs_1 : \mathbb{P}T_1, \ldots, xs_n : \mathbb{P}T_n\} \).

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

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, \( \text{Adapter} \uparrow \text{Target} \) is the following pattern.

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

\[
\text{Pred}(\text{Adapter} \uparrow \text{Target}) = \forall \text{Target} \in \text{Targets} \cdot \exists \text{Adapter} \in \text{Adapters} \cdot \exists \text{Adaptee} \in \text{Adaptees} \cdot \text{Pred}(\text{Adapter}).
\]

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

<table>
<thead>
<tr>
<th>SPECIFICATION 3. (Lifted Object Adapters Pattern)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Components</strong></td>
</tr>
<tr>
<td>(1) Targets, Adapters, Adaptees ( \subseteq \text{classes} ),</td>
</tr>
<tr>
<td><strong>Conditions</strong></td>
</tr>
<tr>
<td>(1) ( \forall \text{Adaptee} \in \text{Adaptees} \cdot \exists \text{specreqs} \in \text{Adaptee} \cdot \text{Adaptee} \cdot \text{opers} ),</td>
</tr>
<tr>
<td>(2) ( \forall \text{Target} \in \text{Targets} \cdot \exists \text{requests} \in \text{Target} \cdot \text{opers} ),</td>
</tr>
<tr>
<td>(3) ( \forall \text{Target} \in \text{Targets} \cdot \text{CDR} (\text{Target}) ),</td>
</tr>
<tr>
<td>(4) ( \forall \text{Target} \in \text{Targets} \cdot \exists \text{Adapter} \in \text{Adapters} \cdot \exists \text{Adaptee} \in \text{Adaptees} )</td>
</tr>
<tr>
<td>( (a) \text{Adapter} \rightarrow \text{Target}, )</td>
</tr>
<tr>
<td>( (b) \text{Adapter} \rightarrow \text{Adaptee}, )</td>
</tr>
<tr>
<td>( (c) \forall o \in \text{Target} \cdot \text{requests} \cdot \exists o' \in \text{Adaptee} \cdot \text{specreqs} \cdot (\text{calls}(o, o')) )</td>
</tr>
</tbody>
</table>

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, \( \text{Adapter} \uparrow \text{Target} \) is the pattern that contains a number of \text{Targets} of adapted classes. Each of these has a dependent \text{Adapter} and \text{Adaptee} class configured as in the original \text{Adapter} pattern. In other words, the component \text{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 \( \text{vars}(c), \text{vars}(c_1) \), and \( \text{vars}(c_2) \subseteq \text{Vars}(P) \), the following equalities hold.

\[
P[c_1] \approx P[c_2], \text{ if } c_1 \Rightarrow c_2 \tag{6}
\]

\[
P[c_1][c_2] \approx P[c_1 \land c_2] \tag{7}
\]
An Algebra of Design Patterns

\[ P[true] \approx P \]  
\[ P[false] \approx FALSE \]

**Proof.**

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

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

\[
\text{Pred}(P[c_1]) = \text{Pred}(P) \land c_1 \\
\Rightarrow \text{Pred}(P) \land c_2 \\
= \text{Pred}(P[c_2]).
\]

So by Lemma 4, we have that \( P[c_1] \preceq P[c_2] \).

Similarly, we can prove that

\[
\text{Pred}(P[c_1][c_2]) \Leftrightarrow \text{Pred}(P[c_1 \land c_2]),
\]

and

\[
\text{Pred}(P[true]) \Leftrightarrow \text{Pred}(P).
\]

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

Law (9) holds because \( \text{Pred}(P[false]) = \text{Pred}(P) \land false \), 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

\[ P[c] \preceq P \]  
\[ P[c][c] \approx P[c] \]  
\[ P[c_1][c_2] \preceq P[c_2][c_1] \]

**Proof.**

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

\[ P[c] \preceq P[true] \]
\[ \Leftrightarrow P \]  
(by Law (6))

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

For Law (12), we have that

\[ P[c_1][c_2] \preceq P[c_2 \land c_1] \]
\[ \approx P[c_2 \land c_1] \]  
(by Law (7))

\[ \Leftrightarrow P[c_2 \land c_1] \]  
(by Law (7))

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

\[
(P * Q) \preceq P \quad (13)
\]
\[
Q \preceq P \Rightarrow P * Q \approx Q \quad (14)
\]

PROOF.
Let $P$ and $Q$ be patterns with

\[
\text{Vars}(P) \cap \text{Vars}(Q) = \emptyset. \quad (15)
\]

Assume that

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

Then, we have that

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

For Law (13), we have that

\[
\text{Spec}(P * Q) = \exists x_1 : T_1 \cdots x_m : T_m, y_1 : T'_1 \cdots y_n : T'_n \cdot (\text{Pred}(P) \land \text{Pred}(Q)) \quad \text{(by Def. 1)}
\]
\[
\Leftrightarrow \exists x_1 : T_1 \cdots x_m : T_m \cdot \text{Pred}(P) \land \exists y_1 : T'_1 \cdots y_n : T'_n \cdot \text{Pred}(Q) \quad \text{(by (15))}
\]
\[
\Rightarrow \exists x_1, \ldots, x_m \cdot \text{Pred}(P) \quad \text{(by logic)}
\]
\[
= \text{Spec}(P) \quad \text{(by Def. 1)}
\]

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

For Law (14), assume that $Q \preceq P$. By Lemma 5(1), we have that

\[
\text{Spec}(Q) \Rightarrow \text{Spec}(P). \quad (16)
\]

Therefore, we have that

\[
\text{Spec}(P * Q) = \exists x_1 : T_1 \cdots x_m : T_m, y_1 : T'_1 \cdots y_n : T'_n \cdot (\text{Pred}(P) \land \text{Pred}(Q)) \quad \text{(by Def. 1)}
\]
\[
\Leftrightarrow \exists x_1 : T_1 \cdots x_m : T_m \cdot \text{Pred}(P) \land \exists y_1 : T'_1 \cdots y_n : T'_n \cdot \text{Pred}(Q) \quad \text{(by (15))}
\]
\[
= \text{Spec}(P) \land \text{Spec}(Q) \quad \text{(by Def. 1)}
\]
\[
= \text{Spec}(Q) \quad \text{(by (16))}
\]

That is, $P * Q \approx Q$. \hfill \square

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 $\preceq$ according to (4).
Law (18) also follows from Law (14), since $\FALSE$ is bottom in $\preceq$ according to (4).
Law (19) follows Law (14), since $\preceq$ is reflexive according to (3). \hfill \square

ACM Journal Name, Vol. XX, No. XX, XX 20XX.
The following theorem proves that the superposition operator is commutative and associative.

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

$$P * Q \approx Q * P$$

(20)

$$(P * Q) * R \approx P * (Q * R)$$

(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 $\text{Vars}(P)$ and to each other, $c_1$ and $c_2$ be any given predicates such that $\text{vars}(c_1) \subseteq \text{Vars}(P) \cup X$ and $\text{vars}(c_2) \subseteq \text{Vars}(P) \cup Y$. The extension operation has the following properties.

$$P \#(X \cdot c_1) \lessapprox P$$

(22)

$$P \#(X \cdot c_1) \lessapprox Q \#(X \cdot c_1), \text{ if } P \lessapprox Q$$

(23)

$$P \#(X \cdot c_1) \lessapprox P \#(X \cdot c_2), \text{ if } c_1 \Rightarrow c_2$$

(24)

$$P \approx \text{TRUE} \#(\text{Vars}(P) \cdot \text{Pred}(P))$$

(25)

$$P \#(X \cdot c_1) \#(Y \cdot c_2) \approx P \#(X \cup Y \cdot c_1 \land c_2)$$

(26)

$$P \#(X \cdot c_1) \#(Y \cdot c_2) \approx P \#(Y \cdot c_2) \#(X \cdot c_1)$$

(27)

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

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

$$P \#(\emptyset \cdot \text{True}) \approx P,$$

(28)

and for all sets $X$ of variables,

$$P \#(X \cdot \text{False}) \approx \text{FALSE}.$$  

(29)

4.4 Laws of Flattening and Generalisation

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

$$P \uparrow \emptyset \approx P,$$

(30)

$$P \downarrow \emptyset \approx P.$$  

(31)

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

**Theorem 5.**
Let \( P \) be any given pattern, \( X, Y \subseteq \text{Vars}(P) \) and \( X \cap Y = \emptyset \). We have that

\[
\begin{align*}
(P \downarrow X) \downarrow Y & \approx (P \downarrow Y) \downarrow X & (32) \\
(P \downarrow X) \downarrow Y & \approx P \downarrow (X \cup Y) & (33) \\
(P \uparrow X) \uparrow Y & \approx (P \uparrow Y) \uparrow X & (34) \\
(P \uparrow X) \uparrow Y & \approx P \uparrow (X \cup Y) & (35)
\end{align*}
\]

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 \( \text{vars}(c) \subseteq \text{Vars}(P) \), we have that

\[
P[c] * Q \approx (P * Q)[c].
\]

**Proof.**

\[
\begin{align*}
(P[c] * Q)[C_C] & \approx ((P * Q)[C_I])[C_C] & \text{(by Law (36))} \\
& \approx (P * Q)[C_C][C_I] & \text{(by Law (12))}
\end{align*}
\]

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 \( X \subseteq \text{Vars}(P) \), we have that

\[
\begin{align*}
(P \uparrow X) * Q & \approx (P * Q) \uparrow X, & (37) \\
(P \downarrow X) * Q & \approx (P * Q) \downarrow X. & (38)
\end{align*}
\]
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 \( \text{Vars}(P) \cap \text{Vars}(Q) = \emptyset \) and \( x_s \notin \text{Vars}(P) \cup \text{Vars}(Q) \). By the definitions of the * and \( \Uparrow \) operators, we have that

\[
\text{Vars}((P \uparrow x \\setminus x_s) \ast Q)
= \text{Vars}(P \uparrow x \\setminus x) \cup \text{Vars}(Q) \quad (\text{by Def. 7})
\]

\[
= ((\text{Vars}(P) \setminus \{ x : T \}) \cup \{ x_s : \mathbb{P}(T) \}) \cup \text{Vars}(Q) \quad (\text{by Def. 10})
\]

\[
= (\text{Vars}(P) \cup \text{Vars}(Q)) \setminus \{ x : T \} \cup \{ x_s : \mathbb{P}(T) \} \quad (\text{by set theory})
\]

\[
= \text{Vars}((P \ast Q) \uparrow x \\setminus x_s) \quad (\text{by Def. 7 and 10})
\]

And,

\[
\text{Pred}((P \uparrow X) \ast Q)
= \text{Pred}(P \uparrow X) \land \text{Pred}(Q) \quad (\text{by Def. 7})
\]

\[
= (\forall x \in x :: \text{Pred}(P)) \land \text{Pred}(Q) \quad (\text{by Def. 10})
\]

\[
\Leftrightarrow (\forall x \in x :: (\text{Pred}(P) \land \text{Pred}(Q)))
= \text{Pred}((P \ast Q) \uparrow x \\setminus x_s) \quad (\text{by first order logic})
\]

(40)

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. \(\Box\)

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 \( X \subseteq \text{Vars}(P) \cup \text{Vars}(Q) \). The following equations hold.

\[
(P \ast Q) \uparrow X \approx (P \uparrow X_P) \ast (Q \uparrow X_Q), \quad (39)
\]

\[
(P \ast Q) \downarrow X \approx (P \downarrow X_P) \ast (Q \downarrow X_Q), \quad (40)
\]

where \( X_P = X \cap \text{Vars}(P) \), \( X_Q = X \cap \text{Vars}(Q) \). \(\Box\)

**Proof.**

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

\[
(P \ast Q) \uparrow X \approx (P \ast Q) \uparrow (X_P \cup X_Q)
\]

\[
= (P \ast Q) \uparrow X_P \uparrow X_Q \quad (\text{by Law (35)})
\]

\[
= ((P \uparrow X_P) \ast Q) \uparrow X_Q \quad (\text{by Law (37)})
\]

\[
= (P \uparrow X_P) \ast (Q \uparrow X_Q) \quad (\text{by Law (37)})
\]

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

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

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

**Theorem 8.** Let \( X \subseteq \text{Vars}(P) \), we have that

\[
(P \uparrow X) \ast Q \approx ((P \ast Q) \uparrow X) \downarrow \text{Vars}(Q) \uparrow . \quad (41)
\]

ACM Journal Name, Vol. XX, No. XX, XX 20XX.
Therefore, by Lemma 4, the theorem is true.

**PROOF.**

Let \( V_P = Vars(P) = \{x_1 : T_1, \cdots, x_n : T_n\} \), \( X = \{x_1 : T_1, \cdots, x_k : T_k\} \), \( n \leq n \), \( V_Q = Vars(Q) = \{y_1 : R_1, \cdots, y_m : R_m\} \).

By the definitions of \( \ast \) and \( \uparrow \), we have that

\[
Vars((P \uparrow X) \ast Q) = Vars(P \uparrow X) \cup Vars(Q) = V_P^\uparrow \cup V_Q.
\]  (42)

\[
Vars((P \ast Q) \uparrow X) = (V_P \cup V_Q)^\uparrow = V_P^\uparrow \cup V_Q^\uparrow
\]  (43)

Therefore, we have that

\[
Vars((P \ast Q) \uparrow X) \downarrow Vars(Q)^\uparrow
\]

\[
= (Vars((P \ast Q) \uparrow X) - Vars(Q)^\uparrow) \cup Vars(Q), \quad \text{by Def. 9}
\]

\[
= (V_P^\uparrow \cup V_Q^\uparrow - V_Q^\uparrow) \cup Vars(Q), \quad \text{by (43))}
\]

\[
= V_P^\uparrow \cup Vars(Q), \quad \text{(by } V_P \cap V_Q = \emptyset\text{)}
\]

\[
= Vars((P \uparrow X) \ast Q).
\]  (44)

By the definitions of \( \ast \) and \( \uparrow \), we also have that

\[
Pred((P \uparrow X) \ast Q) = Pred(P \uparrow X) \land Pred(Q)
\]

\[
= (\forall x_1 \in x_1 \cdots \forall x_k \in x_s k \cdot \exists x_{k+1} \in x_s k+1 \cdots \exists x_n \in x_s n \cdot Pred(P)) \land Pred(Q)
\]

\[
\iff \forall x_1 \in x_1 \cdots x_k \in x_s k \cdot \exists x_{k+1} \in x_s k+1 \cdots \exists x_n \in x_s n \cdot (Pred(P) \land Pred(Q))
\]

Similarly, we have

\[
Pred((P \ast Q) \uparrow X) = \forall x_1 \in x_1 \cdots \forall x_k \in x_s k.
\]

\[
\exists x_{k+1} \in x_s k+1 \cdots \exists x_n \in x_s n \cdot \exists y_1 \in y s_1 \cdots \exists y_m \in y s_m \cdot (Pred(P) \land Pred(Q))
\]

By the definition of \( \downarrow \), we have that

\[
Pred((P \ast Q) \uparrow X) \downarrow Vars(Q)^\uparrow
\]

\[
\iff \forall x_1 \in x_1 \cdots x_k \in x_s k \cdot \exists x_{k+1} \in x_s k+1 \cdots \exists x_n \in x_s n \cdot Pred(P) \land Pred(Q)\]

\[
\exists y_1 \in \{y_1\} \cdots \exists y_m \in \{y_m\} \cdot (Pred(P) \land Pred(Q))
\]

\[
\iff \forall x_1 \in x_1 \cdots x_k \in x_s k \cdot \exists x_{k+1} \in x_s k+1 \cdots \exists x_n \in x_s n \cdot (Pred(P) \land Pred(Q))\]

\[
\iff \forall x_1 \in x_1 \cdots x_k \in x_s k \cdot \exists x_{k+1} \in x_s k+1 \cdots \exists x_n \in x_s n \cdot (Pred(P) \land Pred(Q)\]

\[
= Pred((P \uparrow X) \ast Q)
\]

Therefore, by Lemma 4, the theorem is true. \( \square \)

**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

\[
P \uparrow X \approx (P \uparrow X) \downarrow (V - X^\uparrow)
\]  (44)

\[
(P \downarrow X \setminus X') \downarrow (X'(\setminus X) \approx P
\]  (45)

\[
(P \downarrow X \setminus X') \uparrow X'(\setminus X) \approx P
\]  (46)

ACM Journal Name, Vol. XX, No. XX, XX 20XX.
where \( V = \text{Vars}(P \uparrow X) \). □

4.7 Laws Connecting Restriction to Generalisation, Flattening and Lifting

**Theorem 10.**

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

1. \( \text{vars}(c) \cap X = \{x_1, \ldots, x_m\}, m \leq k; \)
2. \( X - \text{vars}(c) = \{y_1, \ldots, y_u\}; \) and
3. \( \text{Vars}(P) - (X \cup \text{vars}(c)) = \{z_1, \ldots, z_v\}. \)

We have that

\[
P[c] \uparrow X \approx (P \uparrow X)[c_T]
\]

\[
P[c] \uparrow X \approx (P \uparrow X)[c_I]
\]

\[
P[c] \downarrow X \approx (P \downarrow X)[c_b]
\]

where

- \( c_T = \forall x_1 \in xs_1, \ldots, \forall x_m \in xs_m \cdot c, \) and \( \{xs_1 : P(T_1), \ldots, xs : P(T_m)\} \);
- \( c_I = c(x_1', \ldots, x_m', x_{m+1}, \ldots, x_k), x_i' : T_i' \) and \( T_i = P(T_i') \) for \( i = 1, \ldots, m; \)
- \( c_b = \forall x_1 \in xs_1 \ldots \forall x_n \in xs_n \cdot \forall y_1 \in y_{s_1} \ldots \forall y_u \in y_{s_u} ;
\]
\[
\exists x_{m+1} \in xs_{n+1} \ldots \exists x_n \in xs_n \cdot \exists z_1 \in zs \cdot \ldots \cdot \exists z_v \in zs_v \cdot (P(\text{Pred}(P) \land c)).
\]

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, \( X, Y \subseteq \text{Vars}(P), X \cap Y = \emptyset, \) and \( c \) be any predicate such that \( \text{vars}(c) \subseteq X \cup Y \). We have that

\[
((P[c] \uparrow X) \downarrow Y) \approx ((P \uparrow X) \downarrow Y)[c'],
\]

where

- \( c' = \forall x_1 \in xs_1, \ldots, \forall x_m \in xs_m \cdot c, \)
- \( \text{vars}(c) \cap X = \{x_1 : T_1, \ldots, x_m : T_m\}, \) and \( xs_i : P(T_i) \) for \( i = 1 \ldots m. \) □

**Proof.**

By Law (49), we have that

\[
(P[c] \uparrow X) \approx (P \uparrow X)[c_T]
\]

where

- \( c_T = \forall x_1 \in xs_1 \ldots \forall x_n \in xs_n \forall y_1 \in y_{s_1} \ldots \forall y_u \in y_{s_u} ;
\]
\[
\exists x_{m+1} \in xs_{n+1} \ldots \exists x_n \in xs_n \exists z_1 \in zs \cdot \ldots \cdot \exists z_v \in zs_v \cdot (P(\text{Pred}(P) \land c)).
\]

Because \( \text{vars}(c) \subseteq X \cup Y \) and \( X \cap Y = \emptyset, \) and by Law (50), we have that

\[
(P \uparrow X)[c_T] \downarrow Y \approx ((P \uparrow X) \downarrow Y)[(c_T)_b])
\]
where, assuming that \( Y = \{ y_{u'1}, \ldots, y_u \} \cup \{ x_{m+1}, \ldots, x_n \} \cup \{ z_{v'1}, \ldots, z_v \} \), we have

\[
(c)_{\delta} = \forall x_1 \in xs_1 \cdots \forall x_n \in xs \cdot \\
\forall y_1 \in ys_1 \cdots \forall y_{u'} \in ys \cdot \\
\forall y_{u'1} \in \{ y_{u'1} \} \cdots \forall y_u \in \{ y_u \} \cdot \\
\exists x_{m+1} \in \{ x_{m+1} \} \cdots \exists x_n \in \{ x_n \} \cdot \\
\exists z_1 \in zs_1 \cdots z_{v'} \in zs \cdot \\
\exists z_{v'1} \in \{ z_{v'1} \} \cdots \exists z_v \in \{ z_v \} \cdot (Pred(P) \wedge c) \\
= \forall x_1 \in xs_1 \cdots \forall x_n \in xs \cdot \\
\forall y_1 \in ys_1 \cdots \forall y_{u'} \in ys \cdot \\
\exists z_1 \in zs_1 \cdots z_{v'} \in zs \cdot (Pred(P) \wedge c)
\]

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

\[
\forall x_1 \in xs_1 \cdots \forall x_n \in xs \cdot c \wedge \\
\forall x_1 \in xs_1 \cdots \forall x_n \in xs \cdot \\
\forall y_1 \in ys_1 \cdots \forall y_{u'} \in ys \cdot \\
\exists z_1 \in zs_1 \cdots z_{v'} \in zs \cdot Pred(P)
\]

By definition of lifting and flattening, we have that

\[
Pred(P \uparrow X \downarrow Y) = \forall x_1 \in xs_1 \cdots \forall x_n \in xs \cdot \\
\forall y_1 \in ys_1 \cdots \forall y_{u'} \in ys \cdot \\
\exists z_1 \in zs_1 \cdots z_{v'} \in zs \cdot (Pred(P))
\]

Therefore, \((P \uparrow X \downarrow Y)[(c)_{\delta}] \approx (P \uparrow X \downarrow Y)[c]\). The theorem is true. \(\square\)

### 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, \( X = \{ x_1 : T_1, \ldots, x_k : T_k \} \) 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. \(^2\)

\[
P \# (X \bullet c) \approx P[\exists X \cdot c] \quad (52)
\]

\[
P \downarrow (xs \cdot x) \approx P \# (\{ x : T \} \bullet (x \cdot \{ x \})), \text{ where } xs : \mathbb{P}(T) \in Vars(P) \quad (53)
\]

\[
P \uparrow x \cdot xs \approx P \# (\{ x : \mathbb{P}(T) \} \bullet (\forall x \in xs \cdot Pred(P)) \wedge x \in xs), \text{ where } x : T \in Vars(P) \quad (54)
\]

\[
P[c] \approx P \# (\emptyset \bullet c) \quad (55)
\]

\[
P \approx TRUE \# (Vars(P) \bullet Pred(P)) \quad (56)
\]

\[
P \star Q \approx P \# (Vars(Q) \bullet Pred(Q)) \quad (57)
\]

\[
P \uparrow X \approx P \# (Vars(P \uparrow X) \bullet Pred(P \uparrow X)) \quad (58)
\]

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

ACM Journal Name, Vol. XX, No. XX, XX 20XX.
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

\[
\text{Spec}(P\#(X \cdot c)) = \exists (\text{Vars}(P) \cup X) \cdot (\text{Pred}(P) \wedge c)
\]
\[
\iff \exists \text{Vars}(P) \cdot (\text{Pred}(P) \wedge (\exists X \cdot c))
\]
\[
\iff \text{Spec}(P[\exists X \cdot c])
\]

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

\[
\text{Spec}(P \downarrow (xs \setminus x)) = \exists x : T \cdot \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot p(\{x\}, x_1, \cdots, x_n)
\]
\[
\iff \exists xs : \mathbb{P}(T) \cdot \exists x : T \cdot \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot (p(xs, x_1, \cdots, x_n) \wedge \exists x : T \cdot (xs = \{x\}))
\]
\[
\iff \text{Spec}(P\#(\{x : T\} \bullet (xs = \{x\})))
\]

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

\[
\text{Spec}(P \uparrow x \setminus xs) = \exists x : \mathbb{P}(T) \cdot \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot (\forall x \in xs \cdot \text{Pred}(P))
\]
\[
\iff \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot \exists xs : \mathbb{P}(T) \cdot (\forall x \in xs \cdot \text{Pred}(P))
\]
\[
\iff \exists x : T \cdot \exists x_1 : T_1 \cdots \exists x_n : T_n \cdot (\text{Pred}(P) \wedge \exists xs : \mathbb{P}(T) \cdot (\forall x \in xs \cdot \text{Pred}(P)))
\]
\[
\iff \text{Spec}(P\#(xs : \mathbb{P}(T)) \bullet \forall x \in xs \cdot \text{Pred}(P))
\]

\(\square\)

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

COROLLARY 7.

\[
P\#(X \cdot c_1 \wedge c_2) \approx P\#(X \cdot c_1)[c_2]
\]
\[
P\#(X \cdot c) \approx P\#(X \cdot \text{true})[c]
\]

PROOF.
For Law (59), we have that

\[
P\#(X \cdot c_1)[c_2] \approx P\#(X \cdot c_1)(\emptyset \cdot c_2), \quad \text{by (55)}
\]
\[
\approx P\#(X \cup \emptyset \cdot c_1 \wedge c_2), \quad \text{by (27)}
\]
\[
\approx P\#(X \cdot (c_1 \wedge c_2)
\]

For Law (60), it is a special case of Law (59) with \(c_1 = \text{true}\). \(\square\)

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.

\[
\text{Composite}_1 \triangleq \text{Composite}[||\text{Leaves}|| = 1] \quad (61)
\]

\[
\text{Composite}_1 \triangleq \text{Composite} \downarrow \text{Leaves}\setminus\text{Leaf} \quad (62)
\]

\[
\text{Composite} \approx \text{Composite}_1 \uparrow \text{Leaf}\setminus\text{Leaves} \quad (63)
\]

We now first prove that these two definitions of the Composite\(_1\) pattern are equivalent. That is, the following equation is true.

\[
\text{Composite}[||\text{Leaves}|| = 1] \approx \text{Composite} \downarrow \text{Leaves}\setminus\text{Leaf}.
\]

\text{PROOF.}

\[
\text{Composite} \downarrow \text{Leaves}\setminus\text{Leaf}
\]
\[
\approx \text{Composite}[\exists\text{Leaf} : \text{class} \cdot (\text{Leaves} = \{\text{Leaf}\}), \text{ (by Law(52))}
\]
\[
\approx \text{Composite}[||\text{Leaves}|| = 1], \quad \text{ (by set theory)}
\]

\square

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,

\[
\text{Composite} \approx (\text{Composite}_1 \uparrow \text{Leaf}\setminus\text{Leaves}) \downarrow \text{Leaves}\setminus\text{Leaf}.
\]

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.

\[
\text{Composite}_1 \approx (\text{Composite}_1 \uparrow \text{Leaf}\setminus\text{Leaves}) \downarrow \text{Leaves}\setminus\text{Leaf}.
\]

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.

\text{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:

\[
\text{OneAdaptedLeaf} \triangleq (\text{Adapter} \ast \text{Composite})[\text{Target} \in \text{Leaves}][\text{Target}\setminus\text{AdaptedLeaf}]
\]

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

ACM Journal Name, Vol. XX, No. XX, XX 20XX.
components in the composite part of the pattern (i.e. the components in the Composite pattern remain unchanged). Formally, this is defined as follows.

\[
\text{(OneAdaptedLeaf} \uparrow (\text{AdaptedLeaf}\backslash \text{AdaptedLeaves}))
\]
\[
\Downarrow \{\text{Composites}, \text{Components}, \text{Leaveses}\}
\]

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.

<table>
<thead>
<tr>
<th>SPECIFICATION 4. (ManyAdaptedLeaves)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Components</strong></td>
</tr>
<tr>
<td>(1) Component, Composite (\in) classes,</td>
</tr>
<tr>
<td>(2) Leaves, AdaptedLeaves, Adapters, Adaptees (\subseteq) classes,</td>
</tr>
<tr>
<td>(3) (\text{ops} \subseteq \text{Component}.\text{opers})</td>
</tr>
<tr>
<td><strong>Static Conditions</strong></td>
</tr>
<tr>
<td>(1) (\text{ops} \neq \emptyset)</td>
</tr>
<tr>
<td>(2) (\forall o \in \text{ops}.\text{isAbstract}(o)),</td>
</tr>
<tr>
<td>(3) (\forall l \in \text{Leaves}.(l \rightarrow^+ \text{Component} \wedge \neg(l \rightarrow^+ \text{Component})))</td>
</tr>
<tr>
<td>(4) (\forall l \in \text{AdaptedLeaves}.(l \rightarrow^+ \text{Component} \wedge \neg(l \rightarrow^+ \text{Component})))</td>
</tr>
<tr>
<td>(5) isInterface(\text{Component}),</td>
</tr>
<tr>
<td>(6) (\text{Composite} \rightarrow^+ \text{Component})</td>
</tr>
<tr>
<td>(7) (\text{Composite} \rightarrow^* \text{Component})</td>
</tr>
<tr>
<td>(8) (\text{CDR}(\text{Component}))</td>
</tr>
<tr>
<td>(9) (\forall \text{Adaptee} \in \text{Adaptees} \cdot (\exists \text{specreqs} \in \text{Adaptee}.\text{opers}),</td>
</tr>
<tr>
<td>(10) (\forall \text{AdLeaf} \in \text{AdaptedLeaves} \cdot \exists \text{requests} \in \text{AdLeaf}.\text{opers}),</td>
</tr>
<tr>
<td><strong>Dynamic Conditions</strong></td>
</tr>
<tr>
<td>(1) any call to (\text{Composite}) causes follow-up calls</td>
</tr>
<tr>
<td>(\forall m \in \text{messages} \cdot \exists o \in \text{ops} \cdot (\text{toClass}(m) = \text{Composite} \wedge m.\text{sig} \approx o \Rightarrow \exists m' \in \text{messages} \cdot \text{calls}(m, m') \wedge m'.\text{sig} \approx m.\text{sig}))</td>
</tr>
<tr>
<td>(2) any call to a leaf or an adapted leaf does not cause follow-up calls</td>
</tr>
<tr>
<td>(\forall m \in \text{messages} \cdot (\exists o \in \text{ops} \cdot (\text{toClass}(m) \in \text{Leaves} \cup \text{AdaptedLeaves} \wedge m.\text{sig} \approx o) \Rightarrow \neg \exists m' \in \text{messages} \cdot \text{calls}(m, m') \wedge m'.\text{sig} \approx m.\text{sig}))</td>
</tr>
<tr>
<td>(3) (\forall \text{AdLeaf} \in \text{AdaptedLeaves} \cdot (\exists \text{Adapter} \in \text{Adapters}, \text{Adaptee} \in \text{Adaptees}.)</td>
</tr>
<tr>
<td>(a) (\text{Adapter} \rightarrow \text{AdLeaf},)</td>
</tr>
<tr>
<td>(b) (\text{Adapter} \rightarrow \text{Adaptee},)</td>
</tr>
<tr>
<td>(c) (\forall o \in \text{AdLeaf}.\text{requests} \cdot (\exists o' \in \text{Adaptee}.\text{specreqs} \cdot (\text{calls}(o, o')))</td>
</tr>
</tbody>
</table>

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,

\[
\text{ManyAdaptedLeaves} \triangleq \]

ACM Journal Name, Vol. XX, No. XX, XX 20XX.
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 $V_C = \{\text{Composites, Components, Leaveses}\}$.

$$
\frac{(((\text{Adapter} \uparrow \text{Target}\text{\backslash Targets})) \ast \text{Composite})\text{\uparrow} \text{Target}\text{\backslash Targets}}{\text{Targets} \subseteq \text{Leaves}} \Downarrow V_C

\left(\text{Targets} \text{\backslash AdaptedLeaves}\right)
$$

(65)

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

$$\forall \text{Target} \in \text{Targets} \cdot (\text{Target} \in \text{Leaves}),$$

by (51), we have that

$$
\frac{(((\text{Adapter} \ast \text{Composite}) \text{\uparrow} \text{Target}\text{\backslash Targets}) \Downarrow V_C}{\text{Targets} \subseteq \text{Leaves}} \left(\text{Targets} \text{\backslash AdaptedLeaves}\right)
$$

(66)

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

$$
\frac{((\text{Adapter} \ast \text{Composite})\text{\uparrow} \text{Target} \in \text{Leaves}[\text{Target} \in \text{AdaptedLeaf}])}{\text{AdaptedLeaf} \text{\backslash AdaptedLeaves} \Downarrow V_C}
$$

(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:
An Algebra of Design Patterns

(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 $\Gamma_n$ 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 $m$,

$$m \models E_1(P_1, \cdots, P_k) \iff m \models E_2(P_1, \cdots, P_k).$$

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

$$\text{TRUE} \#(V \cdot c),$$

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
\[
E \approx TRUE\#(V_1 \bullet c_1) * TRUE\#(V_2 \bullet c_2) \quad \text{(by induction hypothesis)}
\]
\[
\approx TRUE[\exists V_1 \cdot c_1] * TRUE[\exists V_2 \cdot c_2] \quad \text{(by Law (52))}
\]
\[
\approx (TRUE * TRUE)[\exists V_1 \cdot c_1][\exists V_2 \cdot c_2] \quad \text{(by Theorem 6)}
\]
\[
\approx TRUE[\exists V_1 \cdot c_1][\exists V_2 \cdot c_2] \quad \text{(by Law (17))}
\]
\[
\approx TRUE((\exists V_1 \cdot c_1) \land (\exists V_2 \cdot c_2)) \quad \text{(by Law (7))}
\]
\[
\approx TRUE[\exists V_1 \cdot \exists V_2 \cdot (c_1 \land c_2)] \quad \text{(by logic)}
\]
\[
\approx TRUE[\exists (V_1 \cup V_2) \cdot (c_1 \land c_2)] \quad \text{(by Law (52))}
\]

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
\[
E = E'[c]
\]
\[
\approx TRUE\#(V' \bullet c')[c] \quad \text{(by induction hypothesis)}
\]
\[
\approx TRUE\#(V' \bullet c' \land c) \quad \text{(by Law (59))}
\]

Therefore, the lemma is true in this case.

Case \( \uparrow \): Suppose that \( E = E' \uparrow X \setminus XS \) 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 \( XS = \{x_{s1} : \mathbb{P}(T_1), \cdots , x_{sk} : \mathbb{P}(T_k)\} \). Then, we have that
\[
E = E' \uparrow X \setminus XS
\]
\[
\approx TRUE\#(V' \bullet c') \uparrow X \setminus XS \quad \text{(hypothesis)}
\]
\[
\approx TRUE\#(V' \bullet c') \uparrow (XS \cup \forall x_1 \in x_{s1} \cdots x_k \in x_{sk} \cdot c') \quad \text{(by Law (54))}
\]
\[
\approx TRUE\#(V' \cup XS \bullet c' \land \forall x_1 \in x_{s1} \cdots x_k \in x_{sk} \cdot c') \quad \text{(by Law (26))}
\]

Therefore, the lemma is true in this case.

Case \( \downarrow \): Suppose that \( E = E' \downarrow XS \setminus X \) 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 :
form, say, \( \exists c \) expressions. We can now prove the following uniqueness property of the canonical forms of pattern expressions.

### 6.2 The Completeness Theorem

Let \( E_1 \) and \( E_2 \) be any two given pattern expressions, with canonical forms \( TRUE \#(V_1 \cdot c_1) \) and \( TRUE \#(V_2 \cdot c_2) \), 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,

\[
E_1 = TRUE \#(V_1 \cdot Pr_1), \quad E_2 = TRUE \#(V_2 \cdot Pr_2).
\]

Therefore, the lemma is true in this case.

**Case \( \uparrow \):** Suppose that \( E = E' \uparrow X \setminus XS \) 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. \( E' \approx TRUE \#(V' \cdot c') \). Let \( V' = \{ x_1 : T_1, \ldots, x_n : T_n \} \) and \( X = \{ x_1 : T_1, \ldots, x_k : T_k \} \), where \( 0 < k \leq n \). Then, we have that

\[
E = E' \uparrow X \setminus XS
\]

\[
\approx TRUE \#(V' \cdot c') \uparrow X \setminus XS \quad \text{(hypothesis)}
\]

\[
\approx TRUE \#(V' \cdot c') \#(X \cdot x_s = \{ x_1 \} \land \cdots \land x_s = \{ x_k \}) \quad \text{(by Law (53))}
\]

\[
\approx TRUE \#(V' \cup X \cdot c' \land (x_s = \{ x_1 \} \land \cdots \land x_s = \{ x_k \})) \quad \text{(by Law (26))}
\]

where \( c'' = \forall x_1 \in x_s \cdots \forall x_k \in x_s \cdot \exists x_{k+1} \in x_{s+k+1} \cdots \exists x_n \in x_s \cdot c' \). Therefore, the lemma is true in this case.

**Case \( \# \):** Suppose that \( E = E'(V \cdot c) \) 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. \( E' \approx TRUE \#(V' \cdot c') \). Then, we have that

\[
E = E'(V \cdot c)
\]

\[
\approx TRUE \#(V' \cdot c') \#(V \cdot c) \quad \text{(hypothesis)}
\]

\[
\approx TRUE \#(V \cup V' \cdot (c \land c')) \quad \text{(by Law (26))}
\]

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. \( \Box \)

### 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 \( TRUE \#(V_1 \cdot c_1) \) and \( TRUE \#(V_2 \cdot c_2) \), 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,

\[
E_1 = TRUE \#(V_1 \cdot Pr_1), \quad E_2 = TRUE \#(V_2 \cdot Pr_2).
\]
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 \( \exists V_1 \cdot Pr_1 \Leftrightarrow \exists V_2 \cdot Pr_2 \).

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 \( TRUE\#(V \cdot c) \).
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 \( TRUE\#(V \cdot c) \), 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.

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.

Fig. 6. Illustration of the Proof of Completeness Theorem

The following example demonstrates how to prove the equivalence of two pattern expressions using this process. It also shows that it is sometimes 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
\[ P[\|X\| = 1] \approx TRUE \#(Vars(P) \bullet Pred(P))[\|X\| = 1] \quad (\text{by Law (56)}) \]
\[ \approx TRUE \#(Vars(P) \bullet (Pred(P) \land (\|X\| = 1))) \quad (\text{by Law (59)}) \]

For the right-hand-side, we have that
\[ P \parallel X \approx P\#(\{x : T\} \bullet X = \{x\}) \quad (\text{by Law (53)}) \]
\[ \approx P[\exists x : T : (X = \{x\})] \quad (\text{by Law (52)}) \]
\[ \approx TRUE \#(Vars(P) \bullet Pred(P))[\exists x : T : (X = \{x\})] \quad (\text{by Law (56)}) \]
\[ \approx TRUE \#(Vars(P) \bullet (Pred(P) \land \exists x : T : (X = \{x\}))) \quad (\text{by Law (59)}) \]

Because, in formal predicate logic and set theory, we can prove that
\[ \|X\| = 1 \iff \exists x : T : (X = \{x\}), \]
we have that the equation holds for all patterns \( P \). \( \square \)

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.

![Diagram of Command Pattern](image.png)

\[ \text{Fig. 7. Structure of Command Pattern} \]

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.

\[ RHF_1 \triangleq \text{Command}[\text{Invoker} = \text{Client}, \text{Receiver} = \text{Application}] \]

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.

\[ RHF_2 \triangleq RHF_1 \ast \text{CommandProcessor} \]
\[ [\text{Command} = \text{Component} \land \text{Client} = \text{CommandProcessor}] \]

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.

\[ RHF_3 \triangleq RHF_2 \ast \text{Memento} \]
\[ [\text{Originator} = \text{Application}, \text{Command} \rightarrow \text{Caretaker}] \]
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.

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.

\[
RHF_4 \equiv RHF_3 * \text{Strategy} \\
\quad \text{[Context\ LoggingContext, Strategy\ Logging,} \\
\quad \quad \text{ConcreteStrategies\ ConcreteLoggingStrategies]} \\
\quad \text{[CommandProcessor \rightarrow \ LoggingContext]} \]

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.

\[
RHF_5 \equiv RHF_4 * \text{Composite} \\
\quad \text{[Leaves = ConcreteCommands, Component = Command]} \\
\quad \text{[Composite\ CompositeCommand]} \]

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.

\[
RHF \equiv RHF_5 [\text{Caretaker = Command}] \\
\quad \text{[CommandProcessor = LoggingContext]} \]
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].

![Original Design of the Request-Handling Framework](image_url)

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

$$RH F^o \triangleq Command \ast CommandProcessor \ast Memento \ast Strategy \ast Composite$$

$$\begin{align*}
&\text{[Originator}\backslash\text{Application]} \\
&\text{[Strategy}\backslash\text{Logging]} \\
&\text{[ConcreteStrategies}\backslash\text{ConcreteLoggingStrategies]} \\
&\text{[Context}\backslash\text{CommandProcessor]} \\
&\text{[CommandProcessor.Command = Command.Command} \\
&\quad = \text{Composite.Component = Memento.Caretaker]} \\
&\text{[Leaves}\backslash\text{ConcreteCommands]} \\
&\text{[Composite}\backslash\text{CompositeCommands]} \\
\end{align*}$$

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.

$$RH F \approx RH F^o$$  \hspace{1cm} (68)

To decide if equation (68) holds, we substitute the definitions of $RH F_1, \ldots, RH F_5$ into $RH F$, simplify the expression by applying the algebraic laws, and obtain the following.
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.


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

$$Application = Receiver$$  
$$CommandProcessor = Invoker$$  
$$CommandProcessor = Command.Client$$

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

\[ RHF \approx TRUE \]

\[ \#( \{ Client, Application, CommandProcessor, Logging, Command, CompositeCommand, Memento : Class, ConcreteLoggingStrategies, ConcreteCommands : P(Class) \} \]

\[ \cdot (\{ Client \rightarrow CommandProcessor \} \]

\[ \forall CC \in \text{ConcreteCommands} \cdot (CC \rightarrow Application \]

\[ CC \rightarrow Command \land \lnot \text{isAbstract}(CC)) \}

\[ \text{CommandProcessor} \rightarrow \text{Command} \]

\[ \text{Application} \rightarrow \text{memento} \]

\[ \text{CommandProcessor} \rightarrow \text{Logging} \]

\[ \forall CL \in \text{ConcreteLoggingStrategies} \cdot (CL \rightarrow \text{Logging}) \]

\[ \text{isInterface}(\text{Command}) \land \\
\[ \text{isInterface}(\text{Logging}) \land \\
\[ (\text{CompositeCommand} \rightarrow \text{Command}) \land \\
\[ (\text{CompositeCommand} \rightarrow \text{Command}) \}) \]

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,
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]. How-
ever, 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. 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


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


H. Zhu and I. Bayley


Revised version: 5 April 2012.