

Composing pattern-based components and verifying correctness

Jing Dong^{a,*}, Paulo S.C. Alencar^b, Donald D. Cowan^b, Sheng Yang^a

^a Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA

^b School of Computer Science, The University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

Received 10 July 2006; received in revised form 5 March 2007; accepted 8 March 2007

Available online 14 March 2007

Abstract

Designing large software systems out of reusable components has become increasingly popular. Although liberal composition of reusable components saves time and expense, many experiments indicate that people will pay for this (liberal composition) sooner or later, sometimes paying even a higher price than the savings obtained from reusing components. Thus, we advocate that more rigorous analysis methods to check the correctness of component composition would allow combination problems to be detected early in the development process so that people can save the considerable effort of fixing errors downstream. In this paper we describe a rigorous method for component composition that can be used to solve combination and integration problems at the (architectural) design phase of the software development lifecycle. In addition, we introduce the notion of composition pattern in order to promote the reuse of composition solutions to solve routine component composition problems. Once a composition pattern is proven correct, its instances can be used in a particular application without further proof. In this way, our proposed method involves reusing compositions as well as reusing components. We illustrate our approach through an example related to the composition of design patterns as design components. Structural and behavioral correctness proofs about the composition of design patterns are provided. Case studies are also presented to show the applications of the composition patterns.

© 2007 Elsevier Inc. All rights reserved.

Keywords: Design pattern; Formal specification and verification; Integration; Modeling; Design component; Composition pattern; Temporal logic; Object-Z

1. Introduction

Component-based software development (Nierstrasz and Dami, 1995; Szyperski, 1998; Brown and Wallnau, 1998; Dean and Vigder, 1997; Ning, 1996; Alencar et al., 2002) aims at reducing costs, risks and time-to-market. The idea of composing pre-existing components saves time on developing these systems. In contrast with traditional software development, where system integration is often the tail end of an implementation effort, component assembly and integration is the centerpiece of the approach; thus, component assembly and integration, rather than imple-

mentation, become the focus of system construction. Design components (Keller and Schauer, 1998) have been proposed to reify good design practice, such as design patterns (Gamma et al., 1995; Pree, 1995; Buschmann et al., 1996; Fowler, 1997; Coplien and Schmidt, 1995), from conceptual design building blocks into a tangible and composable form. Design components focus on achieving component-based problem solving instead of component-based implementation.

Although liberal composition of reusable components saves time and expense, many experiments (Garlan et al., 1995; Hissam, 1998) indicate that people will pay for this (liberal composition) sooner or later, sometimes even a higher price than the savings obtained from reusing components. For example, individual components can make certain assumptions about each other that do not hold when the components are put together. Furthermore, there

* Corresponding author.

E-mail addresses: jdong@utdallas.edu (J. Dong), palencar@csg.uwaterloo.ca (P.S.C. Alencar), dcowan@csg.uwaterloo.ca (D.D. Cowan), syang@utdallas.edu (S. Yang).

may be undesirable interactions among the components: properties that hold for the individual components sometimes do not hold anymore after the composition takes place or one is able to deduce properties about the individual components that did not hold before the composition. Instances of problems of this nature include design components (Keller and Schauer, 1998; Dong et al., 2000), commercial-off-the-shelf (COTS) component combination (Hissam, 1998; Dean and Vigder, 1997), software upgrades (Gluch and Weinstock, 1997), and feature interactions (Zave, 1997; Braithwaite and Atlee, 1994). The undetected faults in software compositions may trigger the errors that cause system service failures (Avizienis et al., 2004). Software development and integration faults are at the root of most common computer security problems (Viega and McGraw, 2002). Common security holes and vulnerabilities, such as buffer overflow, are not typically created by the attackers who simply exploit them. The real root cause of the security problems in software systems is the result of bad software design, development and integration.

A portion of the savings from reusing components should be allocated to the composition of components; that is, we should pay for the composition before it is too late and the price is too high. Therefore, ensuring correct composition of components is crucial to the success of component-based software development. The correctness of a composition is proven once, but the composition can be used many times. These proven compositions constitute the composition patterns which can be applied without concern for the inconsistency among their components.

The correctness of composition is important in reducing the time to provide fixes, optimizations, and upgrades to a system. If a composition does not accurately model the components, a developer could easily be misled into making changes that appear to be minor and localized but that, in fact, have widespread consequences. Therefore, a design component should describe explicitly the structural and behavioral elements, interfaces and connections that are required in the target composition, and perhaps more importantly, those that are not intended to appear in the target composition.

Design patterns are typically described in terms of several aspects, such as intent, motivation, applicability, structure, behavior, consequences, known uses, and related patterns. Although not all aspects of a design pattern can be formalized, some functional aspects (structure and behavior) are amenable to formal specifications and rigorous reasoning. In the rest of this paper, we use the phrase “formalizing design patterns to mean formalizing the structural and behavioral aspects of design patterns.

In this paper, we focus on the structural and behavioral aspects of a design pattern component. This allows us to reason about the properties of component composition. In particular, our work provides a semantic description of a component. Hence, we can state, prove and compose the properties of components on a rigorous basis. It also helps us focus on the structural and behavioral aspects of

the component-based design, freeing us from both ambiguous and language-related syntactic characteristics. By using a rigorous characterization, a software designer can determine the correctness of a component composition therefore preventing integration faults and saving the considerable effort of fixing errors downstream in the software development process. We illustrate our approach through an example related to the composition of design patterns as design components. Structural and behavioral correctness proofs about the composition of design patterns are provided. Although we approach the formalization in terms of examples, the underlying principles are applicable to any design pattern that involves class structures and inter-object collaborations.

The main contribution of this paper is a methodology for the correctness-preserving composition of design components. It is important because we want to be able to integrate existing design components into a single design. It is expected to lead to extensive and systematic reuse of design knowledge and proofs with fewer design errors, and ultimately to a design synthesis tool similar to those used for very large integrated circuit design.

The rest of this paper is organized as follows. Section 2 defines basic concepts and notations. Section 3 illustrates the problem by presenting two design patterns and their composition. Section 4 describes the design pattern components using Object-Z. Section 5 defines design components as first-order logic theories. Section 6 describes how to reason about the correctness of the composition mapping in terms of a correctness criterion. Section 7 defines the behavioral semantics of the patterns in a temporal logic and proves the behavioral correctness of the composition in terms of safety and fairness properties. Section 8 presents two case studies to illustrate our approach. The last two sections discuss related work and summarize the ideas of this paper.

2. Basic concepts and notations

Before we can consider the correctness of the composition of two components, we must first decide on the meaning of the components. Suppose that we have a component A containing three parts, say (X, Y, P) , and a component B also containing three parts, say (M, N, P) , as shown in Fig. 1. The parts P and Y in component A are related by a relation R . The parts M and P in component B are also related by R . Now suppose that we want to integrate the

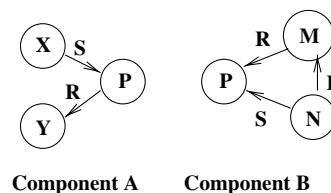


Fig. 1. Two components.

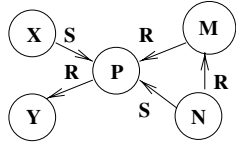


Fig. 2. Composition of component A and component B.

components A and B using P as the overlapping part in the composition as shown in Fig. 2. If relation R is transitive, the parts M and Y are also related by R in the composition of A and B. This derived relation is undesirable in some situations because it may result in an inconsistency in the composition. For example, there may be a case in which Y encapsulates the data of a manager who only wants the system administrator P to have access to his data in component A. Thus, the relation R is defined as access right, that is, xRy denotes that x has the access right to y. In the component B, M has the access right to P. Suppose that access right is transitive. Therefore, the fact that M has the access right to Y can be derived in the composition of the components A and B. This fact contradicts the constraint of component A that only P has the access right to Y. Note, the derived relation R from N to P in component B is desired because we assume that a component itself is correct and complete, and we are only concerned about the correctness of compositions. Now, let us consider another example. Suppose relation S is an inheritance relation, and it is required, in component B, that P inherits only from N, i.e. $S(N,P) \wedge S(N',P) \Rightarrow N = N'$. The composition of components A and B results in multiple-inheritance which is undesirable for component B in this case.

Therefore, we make a completeness assumption about a given component. Informally, the assumption is that, if a fact of a component is not explicitly specified or can be deduced from the component, then this fact is not intended to be true of the component. For the first example, it is not possible to infer the existence of a relation R between parts M and Y from the constraints of component A, so we assume that there should be no relation R between M and Y in the composition of component A and B. For the second example, it is also impossible to infer multiple-inheritance relations from the constraints of component B.

Because of the completeness assumption, we must prove not only that no components lose properties after composition, but also that no new properties about each component can be inferred from the composition.

A composition is an association between the constants, functions, and predicates of all components and their composition. Let P_1, P_2, \dots, P_n denote components, C denotes their composition, then the composition mapping M is defined as $M: P_1 \times P_2 \times \dots \times P_n \rightarrow C$.

Let θ be the theories associated with a component and θ' be those with the composition of components. Let M be the composition mapping from θ to θ' . Then, we must have, for every sentence S,

$$\text{if } S \in \theta \text{ then } M(S) \in \theta' \tag{1}$$

In order to require that the composition does not add new facts to its components, we require that

$$\text{if } S \notin \theta \text{ then } M(S) \notin \theta' \tag{2}$$

That is, if a sentence is not in the theory of a component, its image through the composition mapping cannot be in the theory of the composition system.

We call a composition with property (1) and (2) a faithful composition. Note that a composition can contain facts not related to its components. Therefore, a composition of two components can introduce new objects and new components that do not belong to the two original components as long as it satisfies the faithful composition requirements.

The main reason for the completeness assumption is to define the correctness criteria. The completeness assumption is rather strong for the following reasons: first, although the addition of new facts that do not originally belong to any component into (or the removal of old facts that originally belong to some component from) the composition may not always cause errors, it is considered as a fault so that the software designers may decide whether the fault may cause errors. On the other hand, if it is considered as a correct composition, the designers may not be informed of the possible errors in the software systems. Second, software development is generally a large process including several steps. Composing components is just one such step. The designers can always add or remove facts after the composition if desired. When the developers provide the detailed designs and implement the systems, new facts are normally added and some old facts may be removed as the development process progresses, since more information is generally available at the later stages of the development process. This addition/removal of facts by the developers/designers is intended after the composition. In contrast, the addition/removal of facts during the composition may not be intended. Excluding these facts from the correct composition may allow the designers to be confident about the correctness of the composition. Third, the completeness assumption can be weakened by allowing the addition of new facts during the composition if necessary, i.e., by removing property (2) from the definition of faithful composition. In this case, however, the designers have to make sure the additional facts are intended. In general, a component can contain an unbounded number of facts.

3. Illustration of the problem

Suppose that we want to compose two design patterns as two design components. To illustrate the correctness problem, we focus on the composition of the Composite pattern and the Iterator pattern.

A standard Composite pattern (Gamma et al., 1995) is depicted in Fig. 3. The class diagram is intended to convey visual information of the design component; it is not a

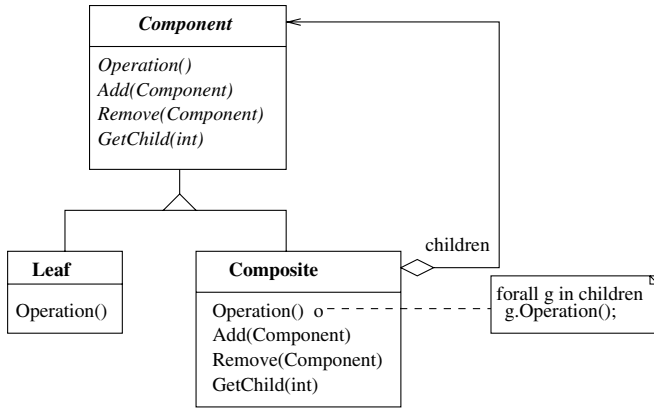


Fig. 3. The Composite pattern.

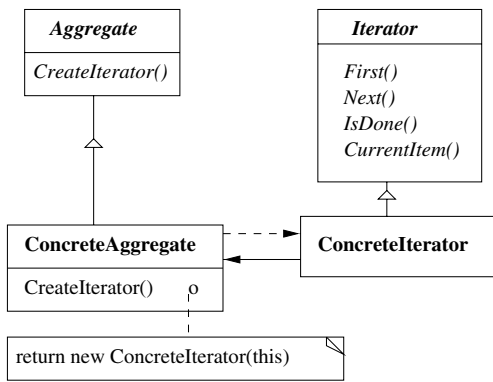


Fig. 4. The Iterator pattern.

Fig. 4 shows the class diagram of the *Iterator* pattern (Gamma et al., 1995). The *Iterator* class is an abstract class which provides the interfaces of the operations, such as *First*, *Next*, *IsDone*, *CurrentItem*, to access the elements of an aggregate object sequentially without exposing its underlying representation. The *ConcreteIterator* class inherits the operation interfaces from the *Iterator* class and defines concrete operations which access the corresponding concrete aggregate. The *Aggregate* class defines a common interface for all aggregates that the *Iterator* accesses. The *ConcreteAggregate* class defines an operation to create the corresponding concrete *Iterator*.

A composition of the *Iterator* pattern and the *Composite* pattern is shown in Fig. 5. Both patterns share the *CompositeAggregate* and *ComponentAggregate* classes; that is, the composition maps the *Composite* class in the *Composite* pattern to the *CompositeAggregate* class, and it maps the *ConcreteAggregate* class in the *Iterator* pattern to the *ComponentAggregate* class as well. It also maps both the *Component* class of the *Composite* pattern and the *Aggregate* class of the *Iterator* pattern into the *ComponentAggregate* class.

Having described an illustrative problem, we are interested in the following questions: Is this composition *faithful* under the completeness assumption? Does this composition behave correctly? When a composition of design components is faithful, we define it as a composition pattern which can be reused. In this way, not only individual design pattern can be reused, but also the faithful compositions of patterns can be considered as a component and reused.

Design patterns are generally described in terms of their structural and behavioral aspects that are suitable to be specified by different formal languages. Most of formal specification languages are appropriate to describe only either structure or behavior of a system. In the following, we specify the structural aspect of design patterns using Object-Z (Duke et al., 1995) and reason about their structural composition in the first-order logic theories. Object-Z is designed to specify object-oriented systems, which fits our goal of specifying design patterns. To reason about structural composition, first-order logic is used. In contrast, the behavioral aspect of design patterns are specified

formal description of the design component. The *Component* class is an abstract class which defines the interfaces of the pattern. The *Composite* and the *Leaf* classes are concrete classes defining the attributes and operations of the concrete components. The *Composite* class can contain a group of children whereas the *Leaf* class cannot. The *Composite* pattern is often used to represent part-whole hierarchies of objects. The goal of this pattern is to treat compositions of objects and individual objects in the composite structure uniformly.

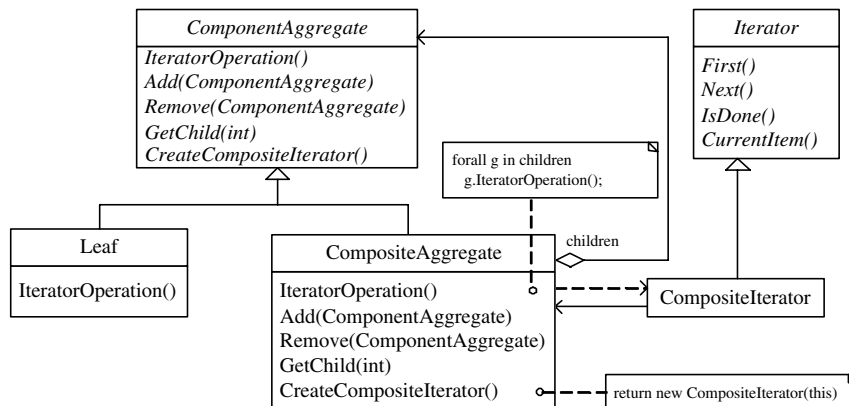


Fig. 5. Composition of the *Iterator* pattern and the *Composite* pattern.

and reasoned about in the Temporal Logic of Action (TLA) (Lampert, 1994).

4. Structural representation

We describe the *Composite* pattern and the *Iterator* pattern in Object-Z (Duke et al., 1995) which is an extension to the Z formal specification language (Spivey, 1992) designed to facilitate specification of data and algorithms in an object-oriented style. An object-oriented specification describes a system as a collection of interacting objects, each of which has a prescribed structure and behavior. Such decomposition improves the clarity of large specifications and facilitates the separation of concerns. Object-Z has been applied to the specification of real-time systems (Mahony and Dong, 1998), a mobile phone system (Rose and Duke, 1993), a button console (Rose, 1992), and the denotational semantics of programming languages (Dong et al., 1994).

Before we use Object-Z to specify design patterns and their compositions, let us review the symbols and their semantics in Object-Z, which are used in this paper. $\downarrow A$ denotes class A and all its descendant classes. $a?$ and $a!$ denote input and output, respectively. They are often used to describe the input parameters and the return values (output). $\uparrow(op)$ represents the visibility of some operations, i.e., only the operations appearing inside the bracket are visible. It is often used with $A[\text{redef: } op]$ to describe that the operation op of parent class A is overwritten (redefined) in this class. $\Delta(a)$ denotes that an operation contains a subset of the state variables. When the operation is applied to an object of the class, those variables not in the $\Delta(a)$ remain unchanged. The prime ' is used to define different states where x and x' represent the current and next states, respectively. Class A may have generic parameters T corresponding to arbitrary types, represented as $A[T]$. The generic parameters may be used to define local types and constants as well as to declare variables in the state and operation schemas. $\langle \rangle$ indicates an empty sequence. $a \frown b$ symbolizes a sequence where a comes before b . $\#$ denotes the number of a sequence or a set.

As shown in Fig. 6, the *Component* class is a common interface for both *Leaf* class and *Composite* class. It defines four operations which can be redefined in its subclasses. The first operation (*Operation*) defines a virtual function which is overwritten by both *Leaf* and *Composite* subclasses to define what should be done when the current component is visited. The *Add* or *Remove* operation defines the mechanism to add or delete a component of a composition structure. Declaration $c:\downarrow C$ introduces c as a reference to an object of class C or any derivative of C . In this example, $child?:\downarrow Component$ defines the *child* as a reference to the class *Component*, where “?” denotes input and “!” denotes output. The input of *GetChild* is an index number of the desired child. Its output is the required child. It is worth pointing out that the *Component* class is an abstract class which does not define what kind of data structure to

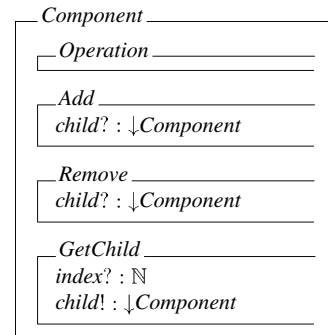


Fig. 6. The *Component* class.

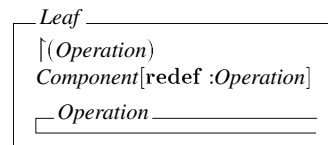


Fig. 7. The *Leaf* class.

store the children of a component and how to add or remove a child. All this information should be available in the *Composite* class.

The *Leaf* class, shown in Fig. 7, inherits from the *Component* class. However, it hides the *Add*, *Remove* and *GetChild* operations because it does not have any child. The only visible information is *Operation* which is defined in the *Leaf* class to perform the operations while this *Leaf* is visited. $\uparrow(Operation)$ denotes that only methods inside the bracket (*Operation*) are visible to the *Leaf* class. $Component[\text{redef: } Operation]$ denotes that the *Component* class is the parent of the *Leaf* class which only redefine the *Operation*.

The *Composite* class, shown in Fig. 10, also inherits from the *Component* class and redefines all its methods: *Operation*, *Add*, *Remove*, *GetChild*. Since each composite class in the Composite pattern may contains a number of components, it declares a state variable (*children*) and an initialization function. The state variable declares the *children* type as a sequence¹ which keep a reference to each component of a composite. Initially, the *children* sequence is empty.

The *Operation* enumerates all components of the current composite and performs their operations. The precondition of *Operation* is that the *children* should not be empty ($children \neq \langle \rangle$).

The *Add* operation takes the *child* as an input and appends it to the *children* sequence. This operation overwrites the *Add* operation in its parent class (*Component*). The precondition of this operation is that there should

¹ While there is no formal reason for choosing a sequence as the type of children, the choice was made for clarity. Clearly, we could have chosen other data structures.

not be the same *child* in the *children* sequence before the addition. The result of adding a *child* is denoted by *children'*.

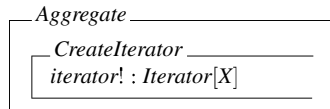


Fig. 8. The *Aggregate* class.

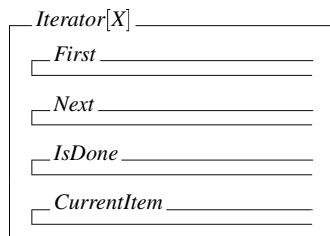


Fig. 9. The *Iterator* class.

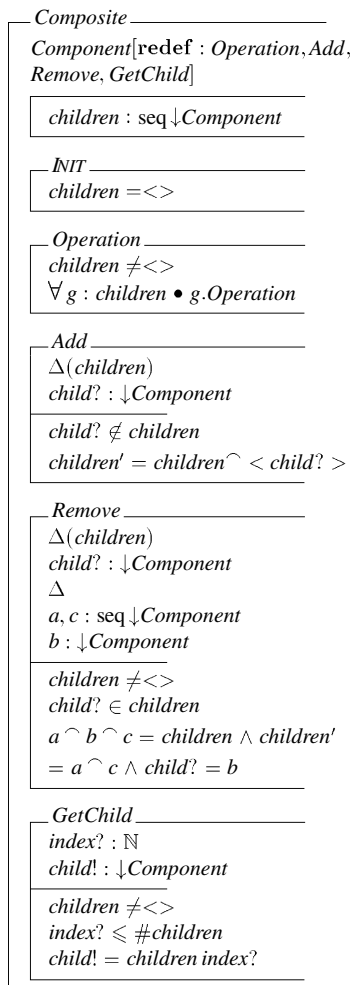


Fig. 10. The *Composite* class.

The *Remove* operation defines the removal of a component from its composite in the Composite pattern in which there is a sequence of components that is kept in an attribute, called *children*. The *Remove* operation deletes a *child* (component) from the *children* sequence of the corresponding composite. It overwrites the *Remove* operation in its parent class (Component). The precondition of removing is that the *children* sequence is not empty and the deleting *child* is in the *children* sequence. Suppose we want to delete a *child* called *b* in a sub-sequence of $a \frown b \frown c$. Then the result of removal is $a \frown c$ where *b* is removed from the sequence.

The *GetChild* operation returns the request component in a composite. It outputs the *child* which is the *index*-th *child* in the *children* sequence (Represented as $child! = children\ index?$ in Object-Z). However, it requires that the *children* sequence is not empty ($children \neq \langle \rangle$) and the index number from input is less than the number of children in the sequence ($index? \leq \#children$).

Similarly to the specification of the Composite pattern in Object-Z described previously, Figs. 8 and 9 shows the *Aggregate* class and the *Iterator* class of the Iterator pattern represented in Object-Z.

5. Design components as theories

In this section, we represent components and compositions as first-order logic theories. Consider the representations of the *Composite* pattern and the *Iterator* pattern in Object-Z. They contain predicates for describing classes, state variables, methods, and their relations. More precisely, the following sorts denote the first-class objects in a pattern: *class* and *object*, where the *class* is the set of all objects of a class. We also make use of sorts *bool* and *int*. The signature for the *Composite* pattern is:

Add: class \rightarrow bool
 Remove: class \rightarrow bool
 GetChild: class \times int \rightarrow bool
 Operation: class \rightarrow bool
 Variable: class \times object \rightarrow bool
 Inherit: class \times class \rightarrow bool

These predicates are used to represent a *Composite* pattern component in ordinary first-order logic. Sorts can be represented as both unary and binary predicates. Each predicate can be either true or false. Thus, they are mapped to *bool*.

An example of well-formedness axiom is that a component can be got if it has been added into an aggregate:

$\forall v[\text{Add}(v) \Rightarrow \text{GetChild}(v, \text{int})]$

Design pattern components are expressed in Object-Z. In order to reason about them, they are translated into first-order logic theories by means of simple rules that generate theories associated with the design components. For the Composite pattern component, the translation is in

Table 1
Partial *Composite* pattern and *Iterator* pattern theories

θ_C	θ_I
AbstractClass(Component)	AbstractClass(Aggregate)
Operation(Component)	CreateIterator
Add(Component)	Class(ConcreteAggregate)
Remove(Component)	CreateIterator
	\Rightarrow New(ConcreteIterator)
GetChild(Component, int)	AbstractClass(Iterator)
Class(Leaf)	First
Operation(Leaf)	Next
Class(Composite)	IsDone
Variable(Component, Children)	CurrentItem
Operation(Composite)	Class(ConcreteIterator)
$\Rightarrow [\forall g[\text{Children}(g)$	
$\Rightarrow \text{Operation}(g)]]$	
$\forall v [\text{Add}(v) \Rightarrow \text{Children}(v)]$	Variable(Aggregate, aggregates)
$\forall v [\text{Children}(v) \Rightarrow \text{Remove}(v)]$	
$\exists v [\text{Children}(v) \wedge \text{GetChild}(v, \text{int})]$	
Inherit(Component, Leaf)	Inherit(Aggregate, ConcreteAggregate)
Inherit(Component, Composite)	Inherit(Iterator, ConcreteIterator)

terms of the previously defined signature for the *Composite* pattern. Table 1 contains (partial) theory θ_C associated with the *Composite* pattern. The theory θ_C is divided into three class groups and one relation group. The first group defines the abstract class *Component* and four method interfaces. The second group corresponds to the *Leaf* class. The third group contains theories about the *Composite* class, which include the definition of a state variable and the operations applied to it. The last group defines two inheritance relations. The first *class* in each inheritance relation is the parent class and the second *class* is the child class.

The signature for the *Iterator* pattern is:

CreateIterator: \rightarrow bool
 New: class \rightarrow bool
 First: \rightarrow bool
 Next: \rightarrow bool
 IsDone: \rightarrow bool
 CurrentItem: \rightarrow bool
 Variable: class \times object \rightarrow bool
 Inherit: class \times class \rightarrow bool

θ_I denotes the theory of the *Iterator* pattern which is divided into five groups. The first four groups contain theories about four classes in the pattern. The last group contains two inheritance relations.

6. Composition mapping

In order to prove the faithfulness of the composition of design components, we must specify a mapping between the individual component and the composition. A composition mapping is an association between formulas of the

language of the theory of each individual component and formulas of the language of the theory of composition. A composition mapping is determined using two different mappings:

- A name mapping associates the classes and objects declared in a pattern component with the classes and objects declared in the composition of this pattern and other patterns.
- A signature mapping connects the constructs of a pattern component with those of a composition. More specifically, it maps uninstantiated predicates of a component to uninstantiated formulas in the composition.

Name mappings are relatively simple whereas signature mappings can be complicated, but need to be defined and proved only once. A name mapping is determined by the identifier associations in a given composition. Let N_C be the name mapping from the *Composite* pattern to its composition:

Component \mapsto *ComponentAggregate*

Composite \mapsto *CompositeAggregate*

N_C defines that the *Component* and *Composite* classes are mapped to the *ComponentAggregate* and *CompositeAggregate* classes, respectively. Observe that not every association appears in the name mapping. We omit the identifiers that map to themselves. For example, the *Leaf* class maps to itself. The name mapping N_I from the *Iterator* pattern is shown as follows:

Aggregate \mapsto *ComponentAggregate*

ConcreteAggregate \mapsto *CompositeAggregate*

ConcreteIterator \mapsto *CompositeIterator*

Let S_C denote the signature mapping from the *Composite* pattern to its composition:

Operation(t) \mapsto *Iterator Operation*(t)

The *Operation* method in the *Composite* pattern is mapped to the *IteratorOperation* method in the composition. Observe that not every association appears in the signature mapping. We omit the predicates that map to themselves. Similarly, the signature mapping S_I from the *Iterator* pattern is shown as follows:

CreateIterator \mapsto *CreateCompositeIterator*

A composition mapping M is determined by a name mapping N and a signature mapping S in the following way. For every predicate P , every variable v , all terms t_1, t_2, \dots, t_n , and all formulas A and B of each pattern component, a composition mapping M is defined as follows:

$$M(P(t_1, t_2, \dots, t_n)) = S(P)(N(t_1), N(t_2), \dots, N(t_n))$$

$$M(\neg A) = \neg(M(A))$$

$$M(A \wedge B) = M(A) \wedge M(B)$$

$$M(A \vee B) = M(A) \vee M(B)$$

$$M(A \Rightarrow B) = M(A) \Rightarrow M(B)$$

$$M(\forall v A) = \forall v M(A)$$

$$M(\exists v A) = \exists v M(A)$$

Let M_C and M_I be the composition mappings from the Composite and Iterator pattern theories to their composition theories, respectively. Let θ'_C be the theory resulting from applying the mapping M_C to the *Composite* pattern theory θ_C . Let θ'_I be the theory resulting from applying the mapping M_I to the *Iterator* pattern theory θ_I . Both the basic facts and the general well-formedness axioms in θ_C and θ_I must be mapped. For example, the predicate *Operation(Component)* is mapped to *IteratorOperation(ComponentAggregate)* and the formula *CreateIterator* \Rightarrow *New(ConcreteIterator)* is mapped to *CreateCompositeIterator* \Rightarrow *New(CompositeIterator)* in the following way.

$$\begin{aligned} &M_C(\text{Operation}(\text{Component})) \\ &= S_C(\text{Operation})(N_C(\text{Component})) \\ &= \text{IteratorOperation}(\text{ComponentAggregate}) \\ &M_I(\text{CreateIterator} \Rightarrow \text{New}(\text{ConcreteIterator})) \\ &= S_I(\text{CreateIterator}) \Rightarrow S_I(\text{New})(N_I(\text{ConcreteIterator})) \\ &= \text{CreateCompositeIterator} \Rightarrow \text{New}(\text{CompositeIterator}) \end{aligned}$$

As discussed previously, two pattern theories can be composed only in ways that preserve faithfulness. More precisely, if

$$M_C : \theta_C \rightarrow \theta'_C \text{ and } M_I : \theta_I \rightarrow \theta'_I$$

are faithful mappings, then we want

$$M_C \cup M_I : \theta_C \cup \theta_I \rightarrow \theta'_C \cup \theta'_I$$

to be a faithful mapping.²

This property requires the composition to satisfy the following two general conditions:

1. The composition mapping must guarantee that mapping M_C and M_I agree on shared objects and parts. For a sentence S , we require that

$$\forall S \in \theta_C \cap \theta_I \bullet M_C(S) = M_I(S)$$
2. It must not be possible to infer new facts about the *Composite* pattern and the *Iterator* pattern from their composition. That is, for language L_C of θ_C and L_I of θ_I , if S is a sentence of $L_C \cup L_I$, and

$$\theta'_C \cup \theta'_I \vdash M_C \cup M_I(S)$$

then we must prove that

$$M_C(\theta_C) \cup M_I(\theta_I) \vdash M_C \cup M_I(S)$$

The proof of the first condition is straightforward for this case since there are no shared objects before the composition.

Table 2 shows the theory resulting from the composition of the two patterns. It is a formal counterpart of the structure shown in Fig. 5. Through the name mapping, both the *Composite* class in the *Composite* pattern and the *ConcreteAggregate* class in the *Iterator* pattern are mapped to one class, called *CompositeAggregate*. This causes the union of the theories about the methods in the *Composite* class with the theories about the methods in the *ConcreteAggregate*, shown in the third group in Table 2. Similarly, the *Component* class of the *Composite* pattern and the *Aggregate* class of the *Iterator* pattern are mapped into the *ComponentAggregate* class in the composition. Since these two sets of theories have no common terms, there are no derived facts. The composition mapping also results in the union of all inheritance relations. These inheritance relations are at two hierarchical levels, thus, there is no transition in these relations. Since there are no derived facts about the composition of the two patterns in this case, it is impossible to further infer new facts about each pattern. Therefore, the second condition holds.

Table 2
Composition theory of *Composite* pattern and *Iterator* pattern

θ
AbstractClass(ComponentAggregate)
IteratorOperation(ComponentAggregate)
Add(ComponentAggregate)
Remove(ComponentAggregate)
GetChild(ComponentAggregate, int)
CreateCompositeIterator
Class(Leaf)
IteratorOperation(Leaf)
Class(CompositeAggregate)
Variable(ComponentAggregate, Children)
IteratorOperation(CompositeAggregate) \Rightarrow $[\forall g$ [Children(g) \Rightarrow Operation(g)]]
$\forall v$ [Add(v) \Rightarrow Children(v)]
$\forall v$ [Children(v) \Rightarrow Remove(v)]
$\exists v$ [Children(v) \wedge GetChild(v, int)]
CreateCompositeIterator \Rightarrow New(CompositeIterator)
AbstractClass(Iterator)
First
Next
IsDone
CurrentItem
Class(CompositeIterator)
Variable(ComponentAggregate, aggregates)
Inherit(ComponentAggregate, Leaf)
Inherit(ComponentAggregate, CompositeAggregate)
Inherit(Iterator, CompositeIterator)

² The union of two theories, denoted by \cup , is the deductive closure of the set union of these theories.

7. Behavioral correctness

The correctness of behavioral composition is concerned with the safety and fairness of behavioral properties. We would like to know that the composition of two components behaves properly. This requires a definition of the semantics of both components. We choose an axiomatic style of semantic definition suitable for describing both safety and fairness properties. In particular, the Temporal Logic of Actions (TLA) (Lamport, 1994), a temporal logic, is used to define the semantics of the *Composite* pattern and the *Iterator* pattern.

In TLA, semantics is provided by assigning a meaning $\llbracket F \rrbracket$ to each syntactic object F . This semantics is defined in terms of a mapping from the set of variable names to the collection of values.

We use the following TLA notations: f denotes a list of variables in the old states, whereas f' denotes a list of variables in the new state. $\Box F$ denotes always F , and $\Diamond F$ denotes eventually F will be true. \mathcal{A} describes an action which relates the old state to the new state. $\llbracket \mathcal{A} \rrbracket_f$ is equivalent to $\mathcal{A} \vee (f' = f)$ which represents that either action \mathcal{A} is taken or there is no change on state function f . $\langle \mathcal{A} \rangle_f$ is equivalent to $\mathcal{A} \wedge (f' \neq f)$ which represents that action \mathcal{A} is taken and the state function f has been changed. *Enabled* \mathcal{A} means that it is possible to take the action \mathcal{A} . Weak fairness, $WF_f(\mathcal{A})$, is defined as $\Box \Diamond \langle \mathcal{A} \rangle_f \vee \Box \Diamond \neg \text{Enabled} \langle \mathcal{A} \rangle_f$. It asserts that eventually action \mathcal{A} must either be taken or become impossible to be taken. In the proof, we make use of the following TLA axiom in Lamport (1994):

$$\text{STL5. } \vdash \Box(F \wedge G) \equiv (\Box F) \wedge (\Box G) \quad (3)$$

The semantics of the *Composite* pattern describes the behavior related to inserting or removing objects of an aggregate. All objects in the aggregate have a common type. The fairness condition is that eventually an insertion or deletion occurs unless both are impossible. Removing an object from an empty aggregate is one reason of impossibility.

We now define a formula Φ that represents the semantics of the *Composite* pattern, meaning that $\sigma \llbracket \Phi \rrbracket$ equals true iff the behavior σ represents a possible application of the *Composite* pattern. The formula Φ is defined in

Table 3. The quoted boldface symbols are logic constants. The symbol \triangleq means *equal by definition*.

In the *Composite* pattern, the *Composite* class is an aggregate class that contains many children with the type of *Component*. We model all the *children* of a *Composite* class as a bag with the initial value of “emptybag”. The behavior of the *Composite* pattern may start at an initial state “ready”. The predicate $Init_\Phi$ asserts the initial condition that *children* is an empty bag and it is ready to take the next action.

The *add* operation in the *Composite* pattern is used to insert the components of the composite. The *Composite* class is responsible for keeping track of all its components (*children*). Thus, when the operation of the composite is invoked, all operations of its components are invoked. The *add* operation is modeled by two predicates: S_{add} (start adding) and E_{add} (end adding). The action S_{add} changes the “ready” state to the “add” state. Initially, the *child* to be added must be in the set **Component** that is the set of all possible component objects that can be inserted into a composite. At this moment, the *children* set stays unchanged. The action E_{add} inserts a *child* component into the *children* set, and changes the state back to “ready” state. The plus sign (+) in action E_{add} represents the insertion of the *child* component into the *children* set according to its internal order, for example, the *child* component is appended to the end if the *children* is a queue, or pushed on the top if it is a stack. We leave open the choice of the aggregation methods of the *children* set to avoid over-specification because the *Composite* pattern in no way depends on this choice.

The components in a composite can be removed from it in the *Composite* pattern so that they are not invoked by the operation of the composite. The removal of components is defined by two predicates: S_{remove} (start removal) and E_{remove} (end removal). Similarly to the addition, the minus sign (−) in the semantics of action E_{remove} represents the removal of the *child* component from the *children* set which keeps its internal order.

The *Composite* pattern also provides an operation (Get-Child) that allows the user to get a particular component in the composite. This operation is useful especially when the composite invokes all its components which, in turn, invoke their own components recursively. Similarly, the

Table 3
Semantics of the *Composite* pattern

$Init_\Phi$	\triangleq	$op = \text{“ready”} \wedge children = \text{“emptybag”}$
S_{add}	\triangleq	$op = \text{“ready”} \wedge op' = \text{“add”} \wedge child' \in \text{Component} \wedge children' = children$
E_{add}	\triangleq	$op = \text{“add”} \wedge op' = \text{“ready”} \wedge children' = children + child$
S_{remove}	\triangleq	$op = \text{“ready”} \wedge op' = \text{“remove”} \wedge child' \in \text{Component} \wedge children' = children$
E_{remove}	\triangleq	$op = \text{“remove”} \wedge op' = \text{“ready”} \wedge child \in children \wedge children' = children - child$
$S_{getchild}$	\triangleq	$op = \text{“ready”} \wedge op' = \text{“get”} \wedge index' \in \text{Index} \wedge children' = children$
$E_{getchild}$	\triangleq	$op = \text{“get”} \wedge op' = \text{“ready”} \wedge child' = children(index) \wedge children' = children$
$N_{children}$	\triangleq	$E_{add} \vee E_{remove}$
N	\triangleq	$N_{children} \vee S_{add} \vee S_{remove} \vee S_{getchild} \vee E_{getchild}$
u	\triangleq	$\langle op, child, children, index \rangle$
Φ	\triangleq	$Init_\Phi \wedge \Box [N]_u \wedge WF_u(N_{children})$

GetChild operation is described by two predicates: $S_{getchild}$ and $E_{getchild}$. The action $S_{getchild}$ changes the “ready” state to the “get” state. Initially, an *index* of the particular child component needs to be provided. This *index* must be in the set **Index** that is the set of all possible indices of the component objects in a composite. The *children* set stays unchanged. The action $E_{getchild}$ gets the particular *child* from the *children* bag based on the provided *index*, and change the state back to “ready”. This action does not change the *children* bag. In this action, the “*children(index)*” refers to the *index* child in the *children* bag. The order of the children can be ordered so that they are added into the composite or any other user defined order. To avoid overspecification, we leave it open.

The behavior of the Composite pattern has to start in the initial state, it must always be possible to add or remove a component in an aggregate, and the aggregate eventually responds to insertion or deletion requests and updates itself if it is possible to do so (fairness). Liveness properties are preserved by requiring fairness.

The semantics of the Iterator pattern describes the behavior of traversing an aggregate of objects. It defines the traversing methods which are independent of the structure of the aggregate on which they traverse. These traversing methods require primitive behaviors, such as *first* and *next*. An internal state memorizes the current position of traversal. Multiple-traversals can be performed concurrently provided that enough state variables are available for recording current position of each traverse. The fairness condition is that traversing steps will eventually proceed and update the corresponding internal state variables.

The semantic theory of the Iterator pattern, called Ψ , is defined similarly in Table 4. It models the current position of the iteration. Initially, the position is set to “start”. The action M_{first} may set the position of the iteration to the beginning (“start”) of an aggregate. The action M_{next} defines the behavior of advancing to the next element in an aggregate, where $Next(position)$ returns the next position

Table 4
Semantics of the *Iterator* pattern

$Init_{\Psi}$	\triangleq	$position = \text{“start”}$
M_{first}	\triangleq	$position' = \text{“start”}$
M_{next}	\triangleq	$position' = Next(position)$
M_{isdone}	\triangleq	$position = \text{“last”} \wedge position' = position$
$M_{current}$	\triangleq	$val' = aggregate(position) \wedge position' = position$
M_1	\triangleq	$M_{first} \vee M_{next}$
M	\triangleq	$M_1 \vee M_{isdone} \vee M_{current}$
v	\triangleq	$\langle position, val \rangle$
Ψ	\triangleq	$Init_{\Psi} \wedge \Box[M]_v \wedge WF_t(M_1)$

from the current position. The action M_{isdone} moves the position to the end (“last”) of the aggregate. The action $M_{current}$ returns the val' that contains the element at the current position in an aggregate.

The composition semantics of the two patterns, called Σ , is described in Table 5. The initial states of the composition includes the initial values of *op*, *children*, and *position* from both design patterns. The *add*, *remove* and *getchild* operations are defined similarly with the *position* unchanged. The *first*, *next*, *isDone*, and *current* operations are defined with the *children* and *op* unchanged.

In order to prove the correctness of this composition, we need to show the composition is faithful. In other words, we want to show that Σ is equivalent to $\Phi \wedge \Psi$. Therefore, we need first to prove that $Init_{\Sigma}$ is equivalent to $Init_{\Phi} \wedge Init_{\Psi}$ which is straightforward. The second step is to show that $\Box[W]_w \equiv \Box[N]_u \wedge \Box[M]_v$. Due to the TLA axiom (3), we only need to show that $[W]_w \equiv [N]_u \wedge [M]_v$.

$$[N]_u \wedge [M]_v \quad (4)$$

$$\equiv (N \vee (u' = u)) \wedge (M \vee (v' = v)) \quad (5)$$

$$\equiv (N \wedge (v' = v)) \vee (M \wedge (u' = u)) \vee (N \wedge M) \vee (u' = u \wedge v' = v) \quad (6)$$

Based on the previous definition of $[\mathcal{A}]_f \equiv \mathcal{A} \vee (f' = f)$, we get (5) from (4). By expanding (5), we get (6). Since

Table 5
Composition semantics of *Composite* pattern and *Iterator* pattern

$Init_{\Sigma}$	\triangleq	$op = \text{“ready”} \wedge children = \text{“emptybag”} \wedge position = \text{“start”}$
S_{add}^{Σ}	\triangleq	$op = \text{“ready”} \wedge op' = \text{“add”} \wedge child' \in Component \wedge children' = children \wedge position' = position$
E_{add}^{Σ}	\triangleq	$op = \text{“add”} \wedge op' = \text{“ready”} \wedge children' = children + child \wedge position' = position$
S_{remove}^{Σ}	\triangleq	$op = \text{“ready”} \wedge op' = \text{“remove”} \wedge child' \in Component \wedge children' = children \wedge position' = position$
E_{remove}^{Σ}	\triangleq	$op = \text{“remove”} \wedge op' = \text{“ready”} \wedge child \in children \wedge children' = children - child \wedge position' = position$
$S_{getchild}^{\Sigma}$	\triangleq	$op = \text{“ready”} \wedge op' = \text{“get”} \wedge index' \in Index \wedge children' = children \wedge position' = position$
$E_{getchild}^{\Sigma}$	\triangleq	$op = \text{“get”} \wedge op' = \text{“ready”} \wedge child' = children(index) \wedge children' = children \wedge position' = position$
M_{first}^{Σ}	\triangleq	$position' = \text{“start”} \wedge op' = op \wedge children' = children$
M_{next}^{Σ}	\triangleq	$position' = Next(position) \wedge op' = op \wedge children' = children$
M_{isdone}^{Σ}	\triangleq	$position = \text{“last”} \wedge position' = position \wedge op' = op \wedge children' = children$
$M_{current}^{\Sigma}$	\triangleq	$val' = aggregate(position) \wedge position' = position \wedge op' = op \wedge children' = children$
$N_{children}^{\Sigma}$	\triangleq	$E_{add}^{\Sigma} \vee E_{remove}^{\Sigma}$
N^{Σ}	\triangleq	$N_{children}^{\Sigma} \vee S_{add}^{\Sigma} \vee S_{remove}^{\Sigma} \vee S_{getchild}^{\Sigma} \vee E_{getchild}^{\Sigma}$
M_1^{Σ}	\triangleq	$M_{first}^{\Sigma} \vee M_{next}^{\Sigma}$
M^{Σ}	\triangleq	$M_1^{\Sigma} \vee M_{isdone}^{\Sigma} \vee M_{current}^{\Sigma}$
W	\triangleq	$N^{\Sigma} \vee M^{\Sigma}$
w	\triangleq	$\langle op, child, children, index, position, val \rangle$
Σ	\triangleq	$Init_{\Sigma} \wedge \Box[W]_w \wedge WF_w(N_{children}^{\Sigma}) \wedge WF_w(M_1^{\Sigma})$

all actions in Φ and Ψ are atomic and $v \cap u = \phi$, it is safe to assume that all variables in v are unchanged when an action in N is taken. Thus $(N \wedge (v' = v)) \equiv N$. Symmetrically, $(M \wedge (u' = u)) \equiv M$.

Therefore,

$$\equiv (N \vee M) \vee (N \wedge M) \vee (u' = u \wedge v' = v) \tag{7}$$

$$\equiv (N \vee M) \vee (u' = u \wedge v' = v) \tag{8}$$

$$\equiv (N \vee M) \vee (w' = w) \tag{9}$$

$$\equiv [W]_w \tag{10}$$

The last step is to show the equivalence of the weak fairness conditions. Since $w = u \cup v$, $WF_u(N_{children}) \equiv WF_w(N_{children})$. Similarly, $WF_v(M_1) \equiv WF_w(M_1)$. It is straightforward to show that $WF_w(N_{children}) \equiv WF_w(N_{children}^\Sigma)$ and $WF_w(M_1) \equiv WF_w(M_1^\Sigma)$.

There is nothing special about our choice of variable names, or in the particular way of writing formulas such as Σ . There are many ways of writing equivalent logic formulas. Thus, some variable names in Σ can be substituted by other names. For example, *aggregate* can be substituted by *children* and this substitution is denoted by $\Sigma(\text{aggregate}/\text{children})$; that is, Σ is equivalent to $\Sigma(\text{aggregate}/\text{children})$. This kind of substitution allows the Iterator pattern to traverse the *children* aggregate (in the Composite pattern) as a concrete *aggregate*.

In this section, we have formalized the semantics of the Composite pattern and the Iterator pattern as TLA theories. Furthermore, we have proved the conjunction of these theories forms the theories of the composition of these two patterns. This proof is sufficient to establish that the composition of these patterns behaves correctly.

8. Case studies

Let us consider a case study of the display of a large map, like a world map. A large map may consist of several regions, such as countries. Each region is a map which can be further divided into smaller regions, such as states or provinces, which contain even smaller regions, such as cities. Eventually, a map is composed of the primitive elements,

such as lines, legends, and text. For instance, a map of the US contains the map of each state. The map of Texas contains the map of each county which in turn consists of the map of each city, such as Dallas. To display a map, all its regions need to be shown. Each region, in turn, shows its own sub-regions recursively until the primitive elements are reached. The Composite pattern can be used to display a map recursively.

A group of maps may be organized based on their temporal relationships. For instance, a collection of US maps may be sorted in chronological order. In addition, a group of maps can be arranged based on some categories, such as in alphabetical order of country/city names and map types (street map, bus map, and satellite map). To traverse a collection of maps without exposing the internal structure, the Iterator pattern can be used to show a group of maps in a particular order.

Fig. 11 depicts the design that displays a group of maps recursively in a particular order. A composition of the Composite and Iterator patterns are applied in the design. The MapAggregate class is an abstract class that is an interface representing both primitives, such as the Text, Line, and Legend in a map, and their containers, such as a map. The Text, Line, Legend classes are the Leaf class of the Composite pattern. The MapAggregate class is also the Aggregate class of the Iterator pattern which defines the interface of the CreateMapIterator operation. The Map class plays the role of the Composite in the Composite pattern and the role of ConcreteAggregate in the Iterator pattern. It defines a composite of map primitives and maps, which are also the concrete aggregate. It implements the operations defined in the abstract class MapAggregate. When a map is displayed, all its regions are displayed recursively until the display of the map primitives.

A collection of maps may be organized in different ways, such as in temporal order or alphabetical order according to the category to which they belong. The Iterator pattern is applied to traverse all available maps sorted in chronological order or classified in some other categories. The Iterator class defines an interface to access and traverse all maps. The TemporalIterator and CategoryIterator

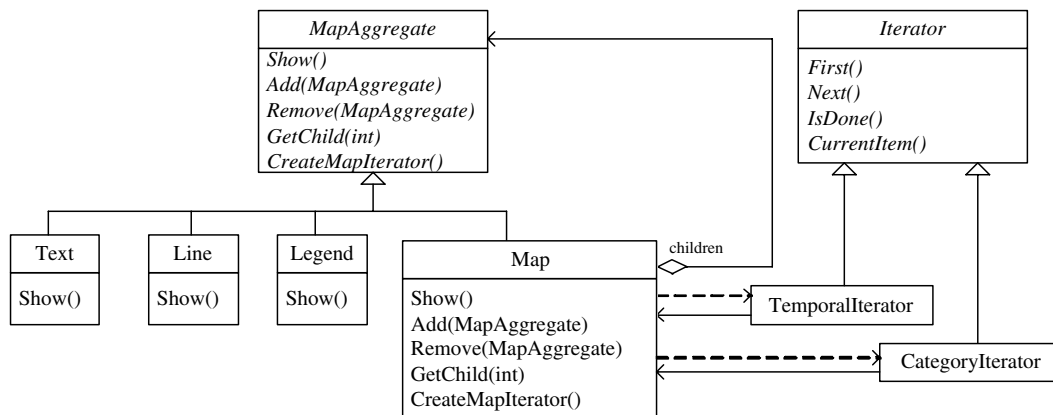


Fig. 11. The design of a map drawing system.

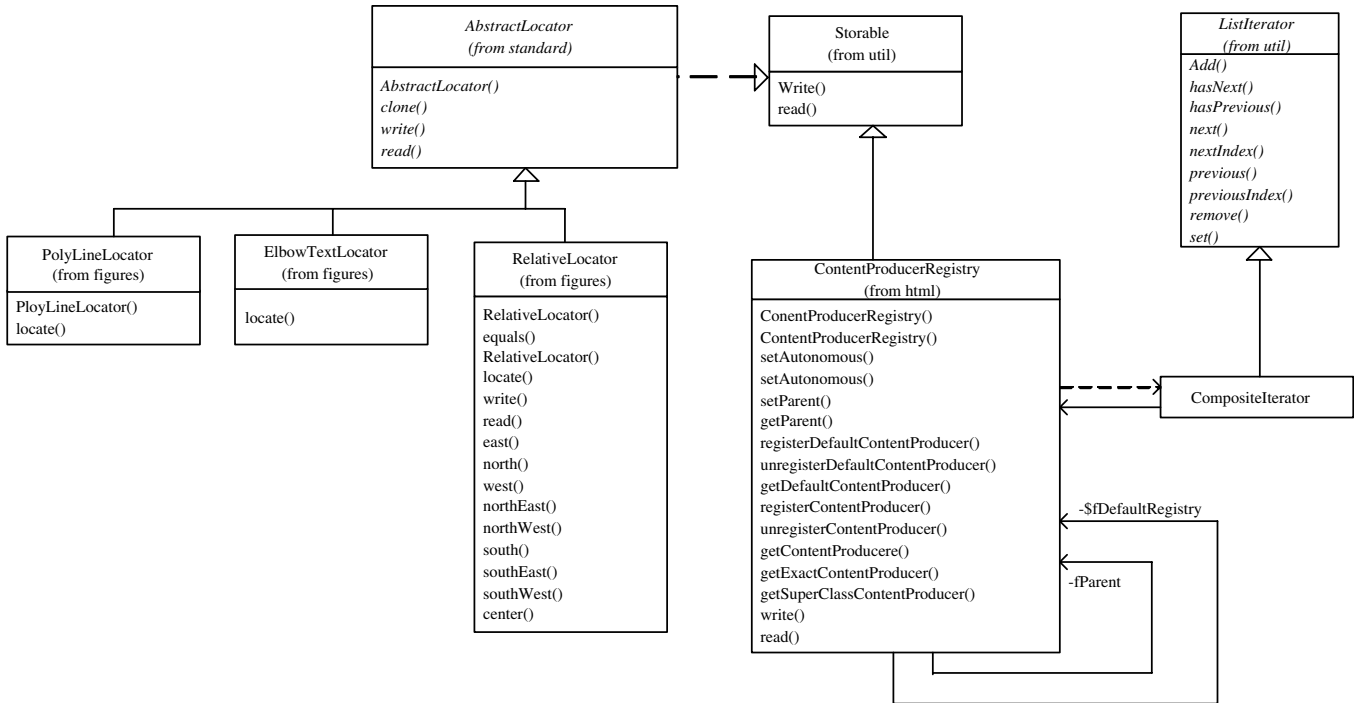


Fig. 12. A composite pattern instance from the JHotDraw system.

classes are the ConcreteIterator that implements the Iterator interface and keeps track of the current position in the traversal according to chronological order or the category of the map, respectively. The MapAggregate class defines an interface to create a map Iterator so that the Map class may implement the interface to return an instance of the TemporalIterator or CategoryIterator. The Map class participates in both the Composite and Iterator patterns. It plays the role of Composite and the role of ConcreteAggregate in the Composite and Iterator patterns, respectively. The MapAggregate class also participates in both the Composite and Iterator patterns. It plays the role of Component and the role of Aggregate in the Composite and Iterator patterns, respectively.

Since we have proved both structurally and behaviorally such composition is correct in the previous sections, we may apply such composition as a composition pattern without further proofs in this case study. The main goal of this map display case study is to illustrate our approach and demonstrate that the mechanisms used for combining patterns are applicable to the specification of complete systems that utilize patterns as their skeletons.

Recent work (Shi and Olsson, 1995) on design pattern discovery has actually recovered an instance of the composition of the Composite and Iterator pattern in a large open source software system, called JHotDraw JHotDraw, with 530 classes. As another case study, Fig. 12 is a partial class diagram that shows the composition pattern. In this design, the Storable class and its descendants form an instance of the Composite pattern, whereas the ListIterator class with its child is the Iterator pattern instance. In particular, the

Storable and ContentProductRegistry classes play the roles of Component and Composite, respectively, in the Composite pattern. The AbstractLocator class with its children is the Leaf.

Since Java provides an interface with its implementation of the Iterator pattern, the design of JHotDraw reused it instead of implementing the Iterator pattern on its own. ListIterator is an interface defined in Java, which provides more operations than standard Iterator interface. The hasNext() operation returns true if this list iterator has more elements when traversing the list in the forward direction. The next() operation returns the next element in the list. The nextIndex() returns the index of the element that would be returned by a subsequent call to next(). This Iterator pattern provided by Java as a library class has been directly reused in the ContentProductRegistry class by JHotDraw. In this way, the Composite and Iterator patterns are composed to form a composition pattern as described previously.

This second case study actually shows that our approach on the composition patterns in this paper is applicable to read-world systems.

9. Related work

The work of C.A.R. Hoare is one of the major sources of inspiration for the research described in this paper. Hoare (1972) applied relative correctness reasoning into a logic framework. Hoare’s technique involves a proof of only theory interpretation, and not of faithfulness. Moriconi et al. (1995) introduced the concept of faithfulness in

reasoning about software architecture refinement and composition.

Formalizing design patterns and architecture patterns has been done in Alencar et al. (1996), Mikkonen (1998), Alencar et al. (1999), Chinnasamy et al. (1999), Dong et al. (2000, 2004, 2006), Saeki (2000), Eden and Hirshfeld (2001), Taibi and Ngo (2003), Soundarajan and Hallstrom (1995). Different from our previous work (Dong et al., 2004, 2006) that analyzed the composition of design pattern instances by model checking techniques, this work defines the correctness criteria based on the faithfulness concept and proves the correctness of the composition of design patterns. The proven compositions can be considered as composition patterns that can be reused over and over. Initial work on this direction has been investigated in Dong et al. (2000).

Mikkonen (1998) has discussed the composition of two design patterns based on a specific specification language (DisCo). The behavior of each pattern is formalized as a layer in DisCo. The composition of design patterns is defined as a refinement on the layers of specifications. The refinement is property-preserving such that the refinement of one pattern by another one preserves all properties of these patterns. The correctness of a composition of design patterns is ensured by defining property-preserving refinement. In contrast, we provide a faithful composition criterion and ensure the correctness of a composition of design patterns by proofs, instead of by definitions that are more restrictive by refinement of DisCo. Different from the DisCo language, we use Object-Z and TLA to specify both the structural and behavioral aspects of a design pattern, respectively. In addition, his approach focused on the behavioral aspect, whereas our approach includes both structural and behavioral aspects of design patterns.

Saeki (2000) formally specifies design patterns and their compositions based on the Language of Temporal Ordering Specification (LOTOS). In particular, he specifies the behavioral aspect of the Command and Composite patterns and their combination. In addition to the behavioral aspect, we are interested in the structural aspect. Our work also presents methods for reason about the compositions of design patterns, rather than just specifications.

Eden and Hirshfeld (2001) present a formal specification approach based on logics with some graphical notations to improve the readability of the specifications. Their approach also concentrates on the specifications of design patterns with visual notations, instead of reasoning about their compositions.

Taibi and Ngo (2003) propose specifying the structural aspect of design patterns in the First-Order Logic (FOL) and the behavioral aspect in the Temporal Logic of Action (TLA) which is similar to our ideas presented in Dong et al. (2000). While the particular ways of using FOL and TLA are somewhat different from ours, their approach is also restricted to the specifications of single design pattern. Although their later work (Taibi, 2006) specifies the com-

positions of design patterns, no verification and analysis of the compositions are discussed.

Soundarajan and Hallstrom (1995) formally specify the structural and behavioral aspects of design patterns in terms of responsibilities and rewards. Following the ideas of the Design by Contract (Meyer, 1992), the structural and behavioral specifications are captured as the responsibilities, whereas the rewards capture the benefits of applying the pattern with the expected behavior in a system. Similar to our goal of not over-specifying, their formal specifications retain flexibility. While their focus is on the single pattern specifications, our work includes the specifications as well as the reasoning about correctness of the compositions of design patterns.

Composition has been studied by Abadi and Lamport (1993). Their results are applicable to any domain, whereas ours are specialized to the domain of component-based software development. Instead of stating general criteria for the correctness of composition, we define the correctness criteria for the composition of design pattern components. Thus, it requires proving that no new facts about each component are inferred from the composition and the composition behaves correctly.

Rangarajan et al. (1995) provided a number of proof obligations for discovering interface and compositional inconsistencies during the component-based software design process. The formal model was based on the components connected through input and output ports. The proof obligations require that each component in a system is connected with a system input through an input trace (activation) and a system output through an output trace (liveness). This analysis, which is similar to reachability analysis in graph theory, leads to the identification of redundant components within a system. However, it cannot ensure that a design is behaviorally correct. Instead of concentrating on defining a complete set of proof obligations, our emphasis is on providing a method for specifying components and reasoning about compositions under a correctness criterion based on the faithfulness concept.

Riehle (1995) proposed an analysis method for the composition of design patterns. Role diagrams were introduced to describe the patterns, and a role relation matrix was used to visually depict the composition constraints. His work was restricted to deal with patterns based on object collaborations, and lacked generality and formal treatment of the correctness of composition.

10. Conclusions

We have described a method for component composition that is relatively correct under a completeness assumption. The notion of faithful composition was introduced in the process of developing a theory of correct composition. We also defined behavioral semantics of components in a temporal logic and reasoned about the correctness of the composition in terms of safety and fairness properties in a rigorous way. Furthermore, we introduced the concept

of composition pattern as the main technique for codifying reusable composition solutions to routine component composition problems. Once a composition pattern is proven correct, its instances can be used in a particular application without further proof as shown in the case studies. In this way, besides proposing reusing components, we propose reusing compositions.

Ensuring correctness at design level in component-based software development is a relevant issue for four major reasons. First, it allows us to find inconsistency in the composition early in the development process always an advantage. Second, it promotes reuse since the correct composition patterns can be reused many times. Although the cost of proving a composition pattern is relatively high, the results of the proofs can be reused over and over which amortizes the cost of later verifications. Third, the correctness of large software system design can be decomposed into smaller components whose correctness can be proved separately. Fourth, critical software systems often require high levels of assurance with respect to the correctness of their design and implementation. Ensuring correctness in the development of these systems is a crucial factor in the success of the produced software.

Our approach can be applied within component evolution and extension (Dong et al., 2006) because the theories about the evolution of a component can be used to ensure the correctness of the composition of the evolved component with other components. Component upgrading can also take advantage of our approach because in this case additional (parts of) components can be added to existing components. The theories of the old version of a component are replaced by the theories of the new version of the component. The correctness proofs can be replaced in the same way. Although the idea of “plug and play” is appealing, many components cannot easily achieve it. Correctness composition proofs, such as the ones shown in this paper, are important to clear the way towards “plug and play”. Our formal approach described in this paper can be partially automated by reusing some components of our recent tools for visualizing (Dong et al., 2005), discovering (Dong et al., 2007) and evolving (Dong et al., 2006) design patterns because all these tools include a parser that can obtain design information from a class diagram. This design information may help to write the specifications of design patterns.

References

- Abadi, Martin, Lamport, Leslie, 1993. Composing specifications. *ACM Transactions on Programming Languages and Systems* 15 (1), 73–132.
- Alencar, Paulo, Cowan, Donald, Dong, Jing, Lucena, Carlos, 1999. A pattern-based approach to structural design composition. In: Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference (COMPSAC), Phoenix USA, pp. 160–165, October.
- Alencar, Paulo, Cowan, Donald, Lucena, Carlos, 1996. A formal approach to architectural design patterns. In: Proceedings of the Third International Symposium of Formal Methods Europe (FME), pp. 576–594.
- Alencar, Paulo S.C., Cowan, Donald D., Lucena, Carlos J.P., 2002. A logical theory of interfaces and objects. *IEEE Transactions on Software Engineering* 28 (6), 548–575.
- Avizienis, Algirdas, Laprie, Jean-Claude, Randell, Brian, Landwehr, Carl, 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1 (1), 11–33.
- Braithwaite, Ken H., Atlee, Joanne M., 1994. Towards automated detection of feature interactions. In: Proceedings of the International Workshop in Telecommunication Systems, pp. 36–59.
- Brown, Alan W., Wallnau, Kurt C., 1998. An examination of the current state of CBSE: a report on the ICSE workshop on component-based software engineering. In: Proceedings of the ICSE Workshop on Component-Based Software Engineering, April.
- Buschmann, Frank, Meunier, Regine, Rohnert, Hans, Sommerlad, Peter, Stal, Michael, 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley.
- Chinnasamy, S., Rajee, R.R., Liu, Z., 1999. Specification of design patterns: an analysis. In: Proceedings of The 7th International Conference on Advanced Computing and Communications, pp. 300–304.
- Coplien, James O., Schmidt, Douglas C., 1995. *Pattern Languages of Program Design*. Addison-Wesley Publishing Company.
- Dean, John C., Vigder, Mark R., 1997. System implementation using off-the-shelf software. In: Proceedings of the 9th Annual Software Technology Conference, April 1997.
- Dong, Jing, Alencar, Paulo, Cowan, Donald, 2000. Ensuring structure and behavior correctness in design composition. In: Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS), April 2000, Edinburgh UK, pp. 279–287.
- Dong, Jing, Alencar, Paulo, Cowan, Donald, 2004. A behavioral analysis and verification approach to pattern-based design composition. *International Journal of Software and Systems Modeling* 3 (4), 262–272.
- Dong, Jing, Alencar, Paulo, Cowan, Donald, 2006. Automating the analysis of design component contracts. *Software – Practice and Experience (SPE)* 36 (1), 27–71.
- Dong, Jing, Lad, Dushyant S., Zhao, Yajing, 2007. DP-miner: design pattern discovery using matrix. In: Proceedings of the 14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS), March 2007, Arizona, USA, pp. 371–380.
- Dong, Jing, Yang, Sheng, Zhang, Kang, 2005. VisDP: a Web Service for visualizing design patterns on demand. In: Proceedings of the IEEE International Conference on Technology Coding and Computing (ITCC), April 2005, pp. 385–391.
- Dong, Jing, Yang, Sheng, Zhang, Kang, 2006. A model transformation approach for design pattern evolutions. In: Proceedings of the Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS), March 2006, pp. 80–89.
- Dong, J.S., Duke, R., Rose, G., 1994. An object-oriented approach to the semantics of programming languages. In: Proceedings of the 17th Annual Computer Science Conference, pp. 767–775.
- Duke, R., Rose, G., Smith, G., 1995. Object-Z: a specification language advocated for the description of standards. *Computer Standards and Interfaces* 17 (September), 511–533.
- Eden, A.H., Hirshfeld, Y., 2001. Principles in formal specification of object-oriented architectures. In: Proceedings of the 11th CASCON, November 2001, Toronto, Canada.
- Fowler, Martin, 1997. *Analysis Patterns: Reusable Object Models*. Addison-Wesley.
- Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John, 1995. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company.

- Garlan, David, Allen, Robert, Ockerbloom, John, 1995. Architectural mismatch or why it's hard to build systems out of existing parts. In: Proceedings of the 17th International Conference on Software Engineering, April 1995, pp. 179–185.
- Gluch, D.P., Weinstock, C.B. (Eds.), 1997. Workshop on the State of the Practice in Dependably Upgrading Critical Systems, also CMU/SEI-97-SR-014, April 1997.
- Hissam, Scott A., 1998. Experience report: correcting system failure in a COTS information system. In: Proceedings of the International Conference on Software Maintenance, Bethesda, USA, November 1998, pp. 170–176.
- Hoare, C.A.R., 1972. Proof of correctness of data representations. *Acta Informatica* 1 (4), 271–281.
- JHotDraw. <http://www.jhotdraw.com/>.
- Keller, Rudolf K., Schauer Reinhard, 1998. Design components: towards software composition at the design level. In: Proceedings of the 20th International Conference on Software Engineering, pp. 302–311.
- Lamport, Leslie, 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16 (3), 873–923.
- Mahony, B., Dong, J.S., 1998. Blending Object-Z and timed CSP: an introduction to TCOZ. In: Proceedings of the 20th International Conference on Software Engineering, April, 1998, Kyoto, Japan, pp. 95–104.
- Meyer, Bertrand, 1992. Applying design by contract. *IEEE Computer* (October), 40–51.
- Mikkonen Tommi, 1998. Formalizing design pattern. In: Proceedings of the 20th International Conference on Software Engineering, pp. 115–124.
- Moriconi, Mark, Qian, Xiaolei, Riemenschneider, R.A., 1995. Correct architecture refinement. *IEEE Transactions on Software Engineering* 21 (4), 356–372.
- Nierstrasz, Oscar, Dami, Laurent, 1995. Component-oriented software technology. In: Nierstrasz, O., Tschritzis, D. (Eds.), *Object-Oriented Software Composition*. Prentice Hall, pp. 3–28.
- Ning, Jim Q., 1996. A component-based software development model. In: Proceedings of the IEEE 20th Annual International Computer Software & Applications Conference, August 1996, pp. 389–394.
- Pree, Wolfgang, 1995. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley Publishing Company.
- Rangarajan, Murali, Alexander, Perry, Abu-Ghazaleh, Nael B., 1999. Using automatable proof obligations for component-based design checking. In: Proceedings of the IEEE International Conference and Workshop on Engineering of Computer-Based Systems, March 1999, Nashville, Tennessee, pp. 304–310.
- Riehle, Dirk., 1997. Composite design patterns. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), October 1997, USA, pp. 218–228.
- Rose, G., 1992. Object-Z. In: Stepney, S., Barden, R., Cooper, D. (Ed.), *Object Orientation in Z*, Workshop in Computing, Springer Verlag, pp. 59–77.
- Rose, G., Duke, R., 1993. An Object-Z specification of a mobile phone system. In: Lano, K., Haughton, H. (Eds.), *Object-Oriented Specification Case Studies*. Prentice Hall, pp. 110–129.
- Saeki, Motoshi, 2000. Behavioral specification of GoF design patterns with LOTOS. In: Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC), December 2000, pp. 408–415.
- Shi, Nija, Olsson, Ron, 2006. Reverse engineering of design patterns from Java Source Code. In: Proceedings of the 21st IEEE International Conference on Automated Software Engineering, September 2006, Tokyo, Japan, pp. 123–134.
- Soundarajan, Neelam, Hallstrom, Jason O., 2004. Responsibilities and rewards: specifying design patterns. In: Proceedings of the 26th International Conference on Software Engineering, May 2004, pp. 666–675.
- Spivey, J.M., 1992. *The Z Notation, A Reference Manual*. Prentice Hall.
- Szyperski, Clemens, 1998. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Longman, Reading, MA.
- Taibi, Toufik, 2006. Formalizing design patterns composition. *IEE Proceedings Software* 153 (3), 127–136.
- Taibi, Toufik, Ngo, David C.L., 2003. Formal specification of design pattern combination using BPSL. *International Journal of Information and Software Technology (IST)* 45 (3), 157–170.
- Viega, John, McGraw, Gary, 2002. *Building Secure Software*. Addison Wesley Longman, Reading, MA.
- Zave, Pamela, 1997. Classification of research efforts in requirements engineering. *ACM Computing Surveys* 29 (4), 315–321.