

Ensuring Structure and Behavior Correctness in Design Composition

Jing Dong, Paulo S.C. Alencar, Donald D. Cowan
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{jdong,palencar,dcowan}@csg.uwaterloo.ca

Abstract

The design of a large component-based software system typically involves the composition of different components. Instead of relying on a free composition of components, 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 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 the utility of our approach through an example related to the composition of design patterns as design components. Structural and behavioral correctness proofs about the composition of some design patterns are provided.

1 Introduction

One of the main goals of component-based software development [19, 22, 6] is to reduce 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, implementation has given way to assembly and integration as the focus of system construction.

Although free composition of reusable components saves time and expense, many experiences [10, 12] indi-

cate that people will pay for this (free 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. In this case, we are dealing with a problem called architectural mismatch. Furthermore, there 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 [14], commercial-off-the-shelf (COTS) component combination [12], software upgrades [11], and feature interactions [23, 5]. 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 concerning on the inconsistency among their components.

In this paper, we focus on the structure and behavior aspects of a 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 structure and behavior 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 saving considerable effort of fixing errors downstream in the software development process.

Design components [14] have been proposed to reify

good design practice, such as design patterns, 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. We illustrate the utility of our approach through an example related to the composition of design patterns as design components. Structural and behavioral correctness proofs about the composition of some design patterns are provided. However, we notice that our approach is not restricted to design components and, thus, can be applied to other kinds of components as well.

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 Figure 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 components A and B using P as the overlapping part in the composition as shown in Figure 2. If relation R is transitive, the parts Y and M will also be related by R in the composition of A and B. This derived relation is undesired in some situations because it results in an inconsistency in the composition. For example, there may be a case in which Y encapsulates the data of a manager who wants nobody but 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 Y¹. Now, let's consider another example. Suppose relation S is 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 undesired 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 deducible 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 com-

¹ The derived relation R from P to N in component B is desired because we assume that a component itself is correct and complete, and we only concern about the correctness of compositions.

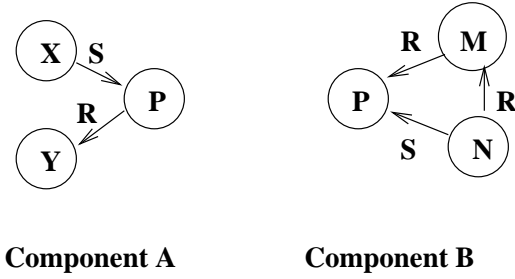


Figure 1. Two Components

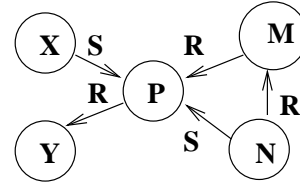


Figure 2. Composition of Component A and Component B

position of component A and B. For the second example, it is also impossible to infer multiple-inheritance relations from the constraints of component B. In general, a component can contain an unbounded number of facts. Composition is possible under the completeness assumption if certain syntactic constraints are satisfied.

Because of the completeness assumption, we must prove not only that no components lose any 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, T denotes their composition, then the *composition mapping* C is defined as $C : P_1 \times P_2 \times \dots \times P_n \rightarrow T$.

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

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

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

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

That is, if a sentence is not in a component, its image through the composition mapping can not be in 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 can introduce new objects and new components.

3 Illustration of the Problem

Suppose that we want to compose two design patterns [9] as two design components. To illustrate the correctness problem, we focus on the composition of the *Composite* pattern and the *Iterator* pattern.

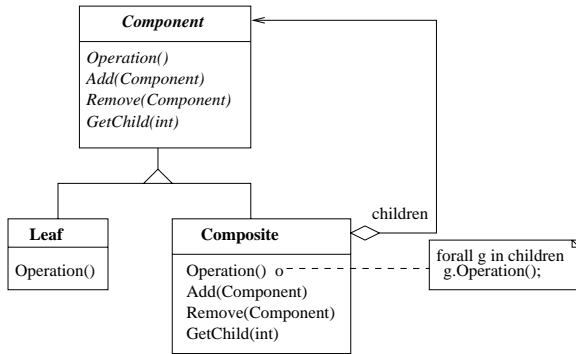


Figure 3. The Composite Pattern

A standard *Composite* pattern is depicted in Figure 3. 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 can not. 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.

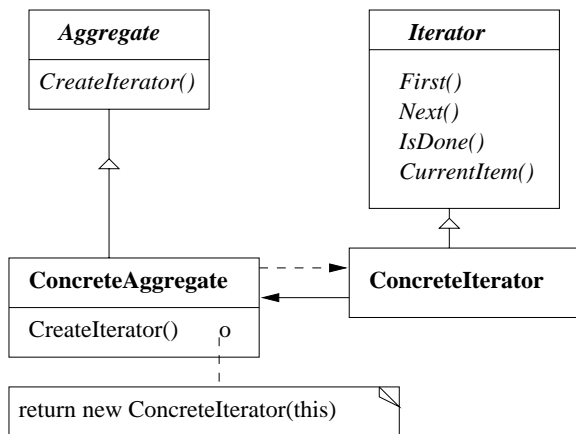


Figure 4. The Iterator Pattern

Figure 4 shows the *Iterator* pattern. The *Iterator* class is an abstract class which provides the interfaces of the oper-

ations, 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*.

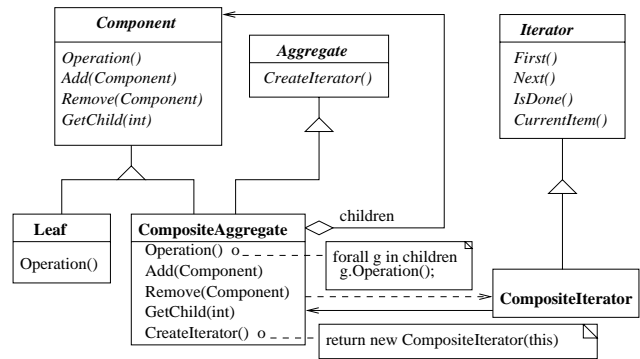


Figure 5. Composition of the Iterator Pattern and the Composite Pattern

The composition of the *Iterator* pattern and the *Composite* pattern is shown in Figure 5. Both patterns share the *CompositeAggregate* class; 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 *CompositeAggregate* class as well. 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?

4 First-Order Logic Representation

We want to leave open the choice of the specification language for components and compositions. Therefore, we represent components and compositions as first-order logic theories, but the correctness of composition in no way depends on this choice.

The representations of the *Composite* pattern and the *Iterator* pattern 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*. 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

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

Table 1 contains (partial) theories associated with the two patterns. θ_C denotes the theory of the *Composite* pattern and θ_I denotes the theory of the *Iterator* 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 on 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 theory θ_I is divided into five groups. The first four groups contain theories about four classes in the pattern. The last group contains two inheritance relations.

5 Composition Mapping

The name mapping is applied during the composition of the two patterns. A *name mapping* associates the classes and objects declared in a pattern with the classes and objects declared in the composition of this pattern and other patterns. The name mapping C_1 from the *Composite* pattern is shown as following:

Composite \mapsto CompositeAggregate

The name mapping C_2 from the *Iterator* pattern is shown as following:

ConcreteAggregate \mapsto CompositeAggregate
 ConcreteIterator \mapsto CompositeIterator

Let θ'_C be the theory resulting from applying the mapping C_1 to the *Composite* pattern theory θ_C . Let θ'_I be the theory resulting from applying the mapping C_2 to the *Iterator* pattern theory θ_I . The two pattern theories can be composed only in ways that preserve faithfulness. More precisely, if

$$C_1 : \theta_C \rightarrow \theta'_C \text{ and } C_2 : \theta_I \rightarrow \theta'_I$$

are faithful mappings, then we want

$$C_1 \cup C_2 : \theta_C \cup \theta_I \rightarrow \theta'_C \cup \theta'_I$$

to be a faithful mapping².

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

1. The composition mapping must guarantee that mapping C_1 and C_2 agree on shared objects and parts. For a sentence S , we require that

$$\forall S \in \theta_C \cap \theta_I \bullet C_1(S) = C_2(S)$$

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

$$\theta'_C \cup \theta'_I \vdash C(S)$$

then we must prove that

$$C(\theta_C) \cup C(\theta_I) \vdash C(S)$$

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 Figure 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. 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 all at two hierarchy levels, and, thus, there is no transition in these relations. It is worth mentioning that the composition results in multiple inheritance relations of the *CompositeAggregate* class. 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.

6 Correctness of Behavior Properties

The correctness of behavior composition is concerned with the safety and fairness of behavior properties. We

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

θ_C	θ_I
AbstractClass(Component) Operation(Component) Add(Component) Remove(Component) GetChild(Component, int)	AbstractClass(Aggregate) CreateIterator
Class(Leaf) Operation(Leaf)	Class(ConcreteAggregate) CreateIterator \rightarrow New(ConcreteIterator)
Class(Composite) Variable(Component, Children) Operation(Composite) \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)]	AbstractClass(Iterator) First Next IsDone CurrentItem
Inherit(Component, Leaf) Inherit(Component, Composite)	Class(ConcreteIterator) Variable(Aggregate, aggregates)
	Inherit(Aggregate, ConcreteAggregate) Inherit(Iterator, ConcreteIterator)

Table 1. Partial Composite Pattern and Iterator Pattern Theories

θ
AbstractClass(Component) Operation(Component) Add(Component) Remove(Component) GetChild(Component, int)
Class(Leaf) Operation(Leaf)
Class(CompositeAggregate) Variable(Component, Children) Operation(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)] CreateIterator \rightarrow New(CompositeIterator)
AbstractClass(Aggregate) CreateIterator
AbstractClass(Iterator) First Next IsDone CurrentItem
Class(CompositeIterator) Variable(Aggregate, aggregates)
Inherit(Component, Leaf) Inherit(Aggregate, CompositeAggregate) Inherit(Component, CompositeAggregate) Inherit(Iterator, CompositeIterator)

Table 2. Composition Theory of Composite Pattern and Iterator Pattern

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) [15], a temporal logic, is used to define the semantics of the *Composite* pattern and the *Iterator* pattern.

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.

The semantics of the *Iterator* pattern describes the behavior of traversing an aggregate of objects. It defines the traversing methods which are independent on 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 traversing. Multiple-traversing can be performed provided that enough state variables are available for recording current position of each traverse. The fairness condition is that traversing steps will eventually proceed which will update the corresponding internal state variables.

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, $\text{WF}_f(\mathcal{A})$, 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:

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

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 $\stackrel{\Delta}{\equiv}$ means *equal by definition*. The predicate Init_{Φ} asserts the initial condition, that *children* is an empty bag and it is ready to take the next action. The action S_{add} changes the “ready” state to the “add” state. The adding *child* must

be in the set **Component**, the set of all possible objects that can be inserted. 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 at the top if it is a stack. We leave open the choice of the aggregation methods of the *children* set to avoid overspecification because the *Composite* pattern in no way depends on this choice. Similarly, 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 actions, S_{remove} , E_{remove} , $S_{getchild}$, and $E_{getchild}$, are defined in a similar manner. 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).

The semantic theory of the *Iterator* pattern, called Ψ , is defined similarly in Table 4.

The composition semantics of the two patterns, called Σ , is described in Table 5. We want to show that Σ is equivalent to $\Phi \wedge \Psi$. Therefore, we need first to prove that Init_{Σ} is equivalent to $\text{Init}_{\Phi} \wedge \text{Init}_{\Psi}$ 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 $[W]_w \equiv [N]_u \wedge [M]_v$.

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

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

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

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

Since 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,

$$\equiv (N \vee M) \vee (N \wedge M) \vee (u' = u \wedge v' = v) \quad (7)$$

$$\equiv (N \vee M) \vee (u' = u \wedge v' = v) \quad (8)$$

$$\equiv (N \vee M) \vee (w' = w) \quad (9)$$

$$\equiv [W]_w \quad (10)$$

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

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.

7 Related Works

The work of C.A.R. Hoare is one of the major sources of inspiration for the research described in this paper. Hoare [13] 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. [18] introduced the concept of faithfulness in reasoning about software architecture refinement and composition.

Composition has been studied by Abadi and Lamport [1]. Their results are applicable to any domain, whereas ours are specialized in the domain of component-based software development. It is easy to state general criteria for the correctness of composition. However, it requires to prove that no new facts about each component are inferred from the composition and the composition behaves correctly.

Rangarajan, Alexander and Abu-Ghazaleh [20] 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, but it can not 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 [18].

Riehle [21] 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.

Formalizing design patterns and architecture patterns has been proposed in [3, 17]. Although Mikkonen [17] has dis-

cussed the composition of two design patterns based on a formal method, his approach relies on a specific specification language (DisCo). The correctness depends on the refinement correctness of this language since the composition is achieved in terms of refinement. Our approach does not rely on any specification language. It does not restrict to first-order logic representation either. Moreover, his approach focuses on formalizing design patterns, whereas our work deals with a more general approach based on design components [14] and their composition. Several formal approaches to component-based software development have been proposed, which focus on the consistent extension of components [16], on formal contract for components [7], and on a viewpoint-based approach to gluing components [4]. The approach proposed in this paper focuses on the correct composition of components.

8 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 behavior 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. In this way, besides proposing reusing components, we also 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 and save time to correct them later. Second, it promotes reuse since the correct composition patterns can be reused many times. 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 as 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.

Although we approach the correct composition in terms of examples on design patterns, we notice that the underlying principles are applicable to any design components. Moreover, the correctness of the composition of other components can also be ensured when the structure and behavior properties of the components are represented through formal theories. Although this may cost higher prices for com-

$Init_{\Phi}$	\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	$Init_{\Phi} \wedge \square[N]_u \wedge WF_u(N_{children})$

Table 3. Semantics of the Composite 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	$Init_{\Psi} \wedge \square[M]_v \wedge WF_v(M_1)$

Table 4. Semantics of the Iterator Pattern

$Init_{\Sigma}$	\triangleq	$op = \text{"ready"} \wedge children = \text{"emptybag"} \wedge position = \text{"start"}$
S_{add}^{Σ}	\triangleq	$op = \text{"ready"} \wedge op' = \text{"add"} \wedge child' \in \text{Component} \wedge children' = children \wedge position' = position$
E_{add}^{Σ}	\triangleq	$op = \text{"add"} \wedge op' = \text{"ready"} \wedge children' = children + child \wedge position' = position$
S_{remove}^{Σ}	\triangleq	$op = \text{"ready"} \wedge op' = \text{"remove"} \wedge child' \in \text{Component} \wedge children' = children \wedge position' = position$
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	$op = \text{"ready"} \wedge op' = \text{"get"} \wedge index' \in \text{Index} \wedge children' = children \wedge position' = position$
$E_{getchild}^{\Sigma}$	\triangleq	$op = \text{"get"} \wedge op' = \text{"ready"} \wedge child' = children(index) \wedge children' = children \wedge position' = position$
M_{first}^{Σ}	\triangleq	$position' = \text{"start"} \wedge children' = children$
M_{next}^{Σ}	\triangleq	$position' = Next(position) \wedge children' = children$
M_{isdone}^{Σ}	\triangleq	$position = \text{"last"} \wedge position' = position \wedge children' = children$
$M_{current}^{\Sigma}$	\triangleq	$val' = aggregate(position) \wedge position' = position \wedge children' = children$
$N_{children}^{\Sigma}$	\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	$Init_{\Sigma} \wedge \square[W]_w \wedge WF_w(N_{children}^{\Sigma}) \wedge WF_w(M_1^{\Sigma})$

Table 5. Composition Semantics of Composite Pattern and Iterator Pattern

plex components, the cost can be amortized in the usages of the components or offsetted by the safety requirement in critical software systems. In addition to an example about the correct composition of design patterns (when things go well) presented in this paper, we have also described an example in which design pattern combination problems are found (when things go wrong) in [2].

Our approach can be applied within component evolution and extension 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 need to 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 carried in the same way. Furthermore, the components in a composition pattern may be included in a “related components” entry of component specifications such as the one discussed in [8]. Although the idea of “plug and play” is appealing, many components can not easily achieve it. Correctness composition proofs, such as the ones shown in this paper, clear the way towards “plug and play”.

References

- [1] M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.
- [2] P. Alencar, D. Cowan, J. Dong, and C. Lucena. A Pattern-Based Approach to Structural Design Composition. *Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference (COMPSAC), Phoenix USA*, pages 160–165, October 1999.
- [3] P. Alencar, D. Cowan, and C. Lucena. A Formal Approach to Architectural Design Patterns. *Proceedings of the Third International Symposium of Formal Methods Europe*, pages 576–594, 1996.
- [4] P. Alencar, D. Cowan, C. Lucena, and L. Nova. A Model for Gluing Components. *Proceedings of the 3rd International Workshop on Component-Oriented Programming, in conjunction with ECOOP’98*, pages 101–108, 1998.
- [5] K. Braithwaite and J. Atlee. Towards Automated Detection of Feature Interactions. *Proceedings of the International Workshop in Telecommunication Systems*, pages 36–59, 1994.
- [6] A. W. Brown and K. C. Wallnau. An Examination of the Current State of CBSE: A Report on the ICSE Workshop on Component-Based Software Engineering. *Proceedings of the ICSE Workshop on Component-Based Software Engineering*, April 1998.
- [7] M. Büchi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. *Proceedings of the Second International Workshop on Component-Oriented Programming, in conjunction with ECOOP’97*, pages 23–32, 1997.
- [8] J. Dong, P. Alencar, and D. Cowan. A Component Specification Template for COTS-based Software Development. *Proceedings of the International Workshop on Ensuring Successful COTS Development, in conjunction with ICSE-21, Los Angeles, USA*, May 1999.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.
- [10] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why It’s Hard to Build Systems out of Existing Parts. *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, April 1995.
- [11] D. Gluch and C. Weinstock. editors. *Workshop on the State of the Practice in Dependably Upgrading Critical Systems, also CMU/SEI-97-SR-014*, April 1997.
- [12] S. A. Hissam. Experience Report: Correcting System Failure in a COTS Information System. *Proceedings of the International Conference on Software Maintenance, Bethesda, USA*, pages 170–176, Nov. 1998.
- [13] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
- [14] R. K. Keller and R. Schauer. Design Components: Towards Software Composition at the Design Level. *Proceedings of the 20th International Conference on Software Engineering*, pages 302–311, 1998.
- [15] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):873–923, May 1994.
- [16] A. Mikhajlova. Consistent Extension of Components in Presence of Explicit Invariants. *Proceedings of the Third International Workshop on Component-Oriented Programming, in conjunction with ECOOP’98*, pages 19–27, 1998.
- [17] T. Mikkonen. Formalizing Design Pattern. *Proceedings of the 20th International Conference on Software Engineering*, pages 115–124, 1998.
- [18] M. Moriconi, X. Qian, and R. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.
- [19] O. Nierstrasz and L. Dami. Component-Oriented Software Technology. *Object-Oriented Software Composition, ed. O. Nierstrasz and D. Tsichritzis, Prentice Hall*, pages 3–28, 1995.
- [20] M. Rangarajan, P. Alexander, and N. B. Abu-Ghazaleh. Using Automatable Proof Obligations for Component-Based Design Checking. *Proceedings of the IEEE International Conference and Workshop on Engineering of Computer-Based Systems, Nashville, Tennessee*, pages 304–310, March 1999.
- [21] D. Riehle. Composite Design Patterns. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), USA*, pages 218–228, October 1997.
- [22] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Longman, Reading, Mass., 1998.
- [23] P. Zave. Classification of Research Efforts in Requirements Engineering. *ACM Computing Surveys*, 29(4):315–321, December 1997.