

# On Instantiation and Integration Commutability of Design Pattern

JING DONG\*, TU PENG AND YAJING ZHAO

*Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA*

*\*Corresponding author: jdong@ieee.org*

**Design patterns capture expert design experience in generic design structure and behavior. To reuse design experience, a design pattern needs to be instantiated from its generic template to the application design in a particular context. It can be integrated with other patterns to solve multiple design problems. The instantiation and integration of design patterns are two important processes when a designer reuses design experience in an application. It is important to know whether the instantiation and integration commute because it can save considerable time and effort of software designers for trial-and-error. In this paper, we investigate the commutability of the instantiation and integration of design patterns. We provide rigorous proofs on the conditions when the order of these two design processes does not matter. Our results allow the software designers to choose the design processes with assurance of their equivalence. The benefits of our work include helping the designers to make informed design decisions based on the convergence of different design processes and reducing the possible design choices, and thus the complexity of software development.**

*Keywords: design pattern; commutability; logics; process algebra*

*Received 22 December 2008; revised 14 December 2009*

*Handling editor: Manfred Broy*

## 1. INTRODUCTION

Design patterns [1] document good practices to solve design problems arising frequently in software applications. They are recipes of solving particular design problems. They have been widely adopted in many software systems such as embedded real-time system [2], mobile agent application [3], database system [4] and ambient ecology environment [5]. The benefits of design patterns include the reuse of design rather than code, document of expert design experience, record of design tradeoffs, capture of design decisions and improvement of communication. A design pattern generally includes a group of participating classes and their relationships and collaborative behaviors. Each participating class in the group is defined generically in terms of the role it plays in the design pattern. A general design process of pattern-based development typically includes the selection of design patterns based on the users' requirements and the application of these patterns in the design. When the intent and motivation of a design pattern match with the users' requirements, the design pattern is selected. The application of design patterns normally involves the instantiation and integration of them. In this paper, we concentrate on the instantiation and integration of design patterns.

To use a design pattern in a particular application, one needs to instantiate it with the application domain information. This instantiation process may change the generic names of the participating classes into those reflecting the application. It may also change the number of the participating classes in some prescribed way. Nevertheless, such changes are not arbitrary and have to respect the constraints of the design pattern, which are normally described in the document of the pattern. Each design pattern may describe different variants of the solution. Each variant of the pattern may correspond to an instance of it.

A design pattern may be composed with other patterns to solve multiple design problems in a software application. The integration of two design patterns describes the particular ways that the two group of classes are combined, which may include stringing (connecting them by some relationships) or overlapping (overlapping them at some classes) [6, 7]. Such integration may happen before or after the pattern instantiation process. When the integration happens before the instantiation, it defines a generic integration that can be considered as an integration pattern [7]. The integration pattern describes a generic solution to several design problems, which can be instantiated in different applications. In the

ideal case, the integration of design patterns always happens before instantiation. When patterns are integrated after they are instantiated, it presents a particular application integration of pattern instances. In refactoring scenarios, for example, separated design pattern instances may be recovered from system source code, which may be integrated in the refactoring processes. In addition, large system designs may be built by different teams with different versions which contain instances of design patterns. The integration of the system designs from different teams may involve the integration of some design pattern instances in the respective system designs.

The instantiation and integration are two important processes when designers use design patterns to solve design problems. They can happen in any order. However, it is unclear whether these two processes commute. To solve the same design problems involving the instantiation and integration of design patterns, suppose two designers choose to work in different orders, i.e. one chooses to instantiate first and the other chooses to integrate first. It is crucial to know whether they reach the equivalent design solution. This is an important issue due to the following reasons. First, commutability analysis can help designers to detect defects in an early stage of the development by supporting the identification of integration problems through verification. Second, it helps reducing the complexity of software development since by knowing that instantiation and integration commute designers can delay instantiation and focus on integration, thus working on a higher, less detailed and complex level of abstraction. Third, it helps designers to make more informed design decisions by providing guidelines that they can use in practice in the process of deciding whether to have pattern integration or pattern instantiation first.

Design patterns are typically described in terms of several aspects such as intent, motivation, applicability, structure, behavior, consequences, known uses and related patterns. Only some functional aspects (structure and behavior) are related to the commutability of the instantiation and integration of design patterns. In the rest of this paper, for simplicity, we use the phrase ‘formalizing design patterns’ to mean formalizing the structural and behavioral aspects of design patterns. In addition, our applications focus on object-oriented design patterns rather than high-level architectural patterns.

In this paper, we investigate the commutability of the instantiation and integration processes based on our formal framework of design patterns. In particular, we provide rigorous proofs on the commutability of these two processes under certain conditions. More specifically, we separate the structural and behavioral aspects of design patterns and formally specify each aspect. We also define the instantiation and integration processes and prove the conditions of commutability. In addition, we discuss the applications of the commutability analysis.

The remainder of this paper is organized as follows: the next section describes the instantiation and integration of design patterns and provides motivating examples. Section 3 studies

the commutability of the structural aspect of design patterns. Section 4 investigates the behavioral commutability. Section 5 discusses the applications of commutability results. The last two sections cover related work and conclusions.

## 2. INSTANTIATION AND INTEGRATION

Consider the Composite pattern [1] that describes a nonlinear hierarchy with a part-whole relationship. The class diagram is shown in Fig. 1 that includes a group of classes organized with certain relationships. Each class in the group plays some particular role that is manifested by its name in the pattern. When the Composite pattern is used in a software application, the names of the classes, operations and attributes may be replaced with the names that have application domain information. Some role of a pattern, such as the Leaf, may be played by multiple classes. This process is the design pattern instantiation. The result of this process is an instance of the pattern.

The Composite pattern has been applied in many large real-world software systems. For example, an instance of the Composite pattern can be found in the Java.awt package [8] as shown in Fig. 2, where the Leaf role is played by three classes, Canvas, Checkbox and Choice. The Container class plays the role of Composite, whereas the Component class is the Component in the Composite pattern.

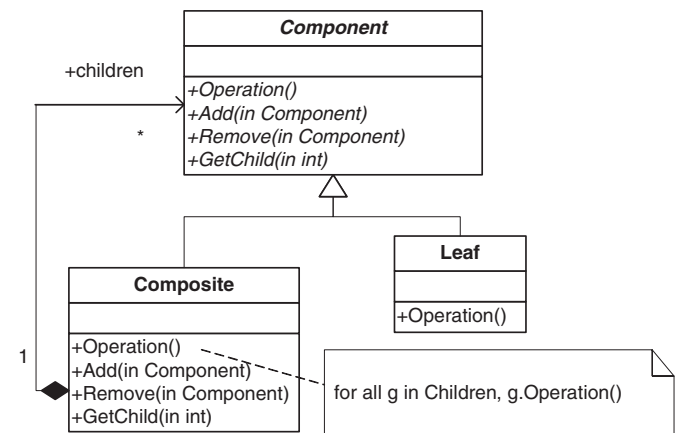


FIGURE 1. Composite pattern.

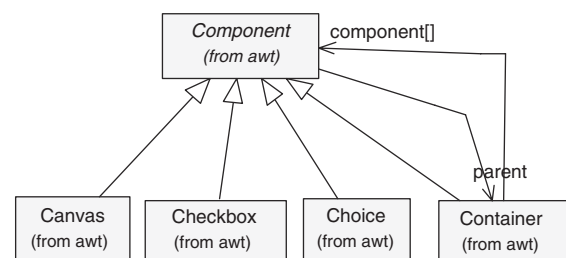


FIGURE 2. An instance of Composite pattern in Java.awt.

The Adapter pattern [1] provides a wrapper to translate an interface of a class that is incompatible with the client into a compatible interface, so as to allow classes to work together without changing the existing classes. As shown in Fig. 3, the client intends to use the SpecificRequest method in the Adaptee class. However, the SpecificRequest method cannot be used directly by the client because the method interface may not be compatible with the client. The client only knows the Target interface, which is the Request operation. The Adapter class can adapt the interface difference between the Target and Adaptee classes by delegating the client’s request to the SpecificRequest method in the Adaptee class. When the Adapter pattern is used in an application, it is instantiated by modifying the names of its modeling elements with application domain information.

The Adapter pattern has also been applied in many real-world applications. For instance, an Adapter instance from the Java.awt package is shown in Fig. 4 where the

AWTEventListener class plays the role of Target, the LightweightDispatcher class plays the role of Adapter and the Container class plays the role of the Adaptee.

In a software design, two or more design patterns may be integrated to solve multiple design problems. There can be several different ways to compose design patterns. For example, an integration of the instances of the Composite and Adapter patterns can be found in the Java.awt package as shown in Fig. 5 where the Container class becomes the overlapping part of the integration. It plays two roles in two different patterns: the Composite role in the Composite pattern and the Adaptee role in the Adapter pattern. This example presents a case when two design patterns are integrated after their instantiations. On the other hand, they can be integrated before instantiation. An interesting research question is whether these two options arrive in the same design result. Are the integration and instantiation of design patterns commutable? The answer of this question is important because it can release the software developers from making difficult design decisions that actually reach the same goal [9]. Thus, it can save design time and reduce possible errors. In addition, it can reduce the complexity of software system designs.

In the following sections, we study the commutability of the integration and instantiation of design patterns based on our formal framework [10–12]. We provide the proofs of commutability theorems. Since each design pattern normally includes both structural and behavioral aspects, we investigate the commutability in both aspects in the following sections.

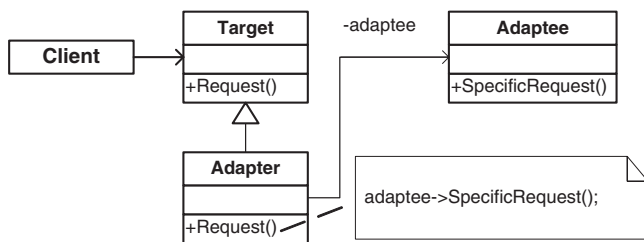


FIGURE 3. Adapter pattern.

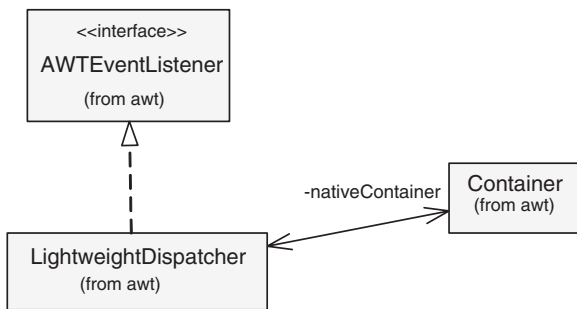


FIGURE 4. An instance of Adapter pattern in Java.awt.

### 3. STRUCTURAL COMMUTABILITY

Structural aspect of design patterns describes the classes and their relationships. It is typically modeled by the UML class diagram informally. This section presents our study on the commutability of the instantiation and integration in the structural aspect of the design patterns. Section 3.1 introduces a definition of the structural aspect of a design pattern, called structural contract. The instantiation and integration of the structural contracts are formally defined in Sections 3.2 and 3.3, respectively. Section 3.4 provides the proofs of the commutability.

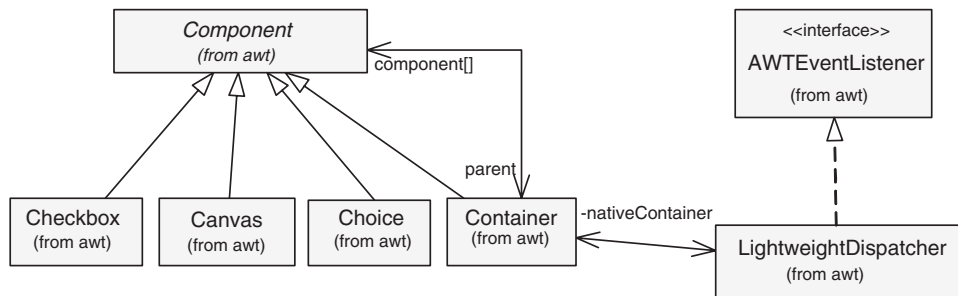


FIGURE 5. An integration of Composite and Adapter pattern instances in Java.awt.

### 3.1. Structural contract

We define the structural aspect of a design pattern as structural contract as follows.

**DEFINITION 3.1** (Structural contract). *Let  $CT$  be the set of all class names of a design pattern. Let  $AVT$  be the set of all attribute variable names of all classes in  $CT$ . Let  $MT$  be the set of all method names of all classes in  $CT$ . Let  $TT$  be the set of all type names used to define attribute variables and methods. Let  $ART$  be the set of access rights. Let  $PST$  be the set of predicates used to specify the relationships of the element in  $CT$ ,  $AVT$ ,  $MT$ ,  $TT$  and  $ART$  in a design pattern. Then, the structural aspect of a design pattern is a tuple  $SC = \langle C, AV, M, T, AR, PS \rangle$ , where  $C \subseteq CT$  is a set of classes in the design pattern;  $AV \subseteq AVT$  is a set of attributes defined in the classes of  $C$ ;  $M \subseteq MT$  is a set of methods defined in the classes of  $C$ ;  $T \subseteq TT$  is a set of types that are used to define the attributes and methods in the classes of  $C$ ;  $AR \subseteq ART$  is a set of access rights that the attributes and methods can have in a class of  $C$ ;  $PS \subseteq PST$  is a set of predicates that specify a structural aspect of a design pattern. These predicates declare the static model of a design pattern.  $PS$  may contain but not limited to the following predicates:*

- *class: If  $class(c)$  is true, then  $c$  plays the role of a class in the pattern.*
- *abstractclass: If  $abstractclass(c)$  is true, then  $c$  plays the role of an abstract class in the pattern.*
- *variable: If  $variable(c, ar, a, t)$  is true, then attribute  $a$  is defined in class  $c$  with type  $t$ . It can be accessed with the right  $ar$ .*
- *method: If  $method(c, ar, m, t)$  is true then  $m$  is a method defined in class  $c$  with return type  $t$ . It can be accessed with the right  $ar$ .*
- *invoke: If  $invoke(c_1, m_1, c_2, m_2)$  is true then method  $m_2$  defined in class  $c_2$  calls method  $m_1$  defined in  $c_1$ .*
- *inherit: If  $inherit(a, b)$  is true then  $b$  is a subclass of  $a$ .*
- *return: If  $return(c, m, o)$  is true then the method  $m$  defined in class  $c$  returns an object  $o$ .*

Based on the above definition, an example of the structural contract of the Composite pattern (Fig. 1) is given as follows.

**EXAMPLE 3.1.** The structural aspect of the Composite pattern contract is  $SC_{Composite} = \langle C, AV, M, T, AR, PS \rangle$  where

- The set of classes in the Composite pattern is  $C = \{Component, Composite, Leaf\}$ .
- The set of attributes defined in classes  $C$  is  $AV = \{children\}$ .
- The set of methods defined in classes  $C$  is  $M = \{Add, Remove, getChild, Operation\}$ .
- The set of types that are used to define the attributes and methods in classes  $C$  is  $T = \{void, Component, Composite, Leaf\}$ .
- The set of access rights that the attributes and methods can have in a class of  $C$  is  $AR = \{public, protected, private\}$ .

- The set of predicate symbols that specify the structural aspect of the Composite pattern is  $PS =$

```
{abstractclass(Component),
method(Component, public, add, void),
method(Component, public, remove, void),
method(Component, public, getChild, Component),
method(Component, public, Operation, void),
class(Composite),
inherit(Component, Composite),
method(Composite, public, add, void),
method(Composite, public, remove, void),
method(Composite, public, getChild, Component),
method(Composite, public, Operation, void),
variable(Composite, private, children),
class(Leaf),
inherit(Component, Leaf),
method(Composite, public, Operation, void) }
```

In this example, the set of predicates specify the structural aspect of the Composite pattern. For instance, the Component class is an abstract class whereas the Composite and Leaf classes are concrete classes. There are also a number of methods defined in each class. For instance, ‘method(Composite, public, remove, void)’ denotes that a public method called ‘remove’ is defined in the Composite class with the return type to be ‘void’.

### 3.2. Structural instantiation

**DEFINITION 3.2** (Instantiation and multiple-instantiation). *Let the structural contract of a design pattern be*

$$SC = \langle C, AV, M, T, AR, PS \rangle,$$

*based on Definition 3.1. Then, an instance, denoted by*

$$SC' = \langle C', AV', M', T', AR', PS' \rangle,$$

*of the structure contract  $SC$  is defined by the instantiation function  $\delta$  which is a function from  $C$  to  $C'$ ,  $AV$  to  $AV'$ ,  $M$  to  $M'$ ,  $T$  to  $T'$ ,  $AR$  to  $AR'$ ,  $PS$  to  $PS'$ , where*

- $C' \subset CT$  and  $C' = \delta(C)$  is a set of new class names that replace the old class names in the design pattern.
- $AV' \subset AVT$  and  $AV' = \delta(AV)$  is a set of new attribute names that replace the old attribute names in the design pattern.
- $M' \subset MT$  and  $M' = \delta(M)$  is a set of new method names that replace the old method names in the design pattern.
- $T' \subset TT$  and  $T' = \delta(T)$  is a set of new type names that replace the old type names in the design pattern. And  $\forall t \in T_i - C_i : \delta(t) = t$ .
- $AR \subset ART$  and  $AR' = \delta(AR)$ . Note that the access rights will not be changed when a design pattern is instantiated.
- $PS' = \{p(y_1, \dots, y_n) \in PST \mid \exists p(x_1, \dots, x_n) \in PS, s.t. y_i = \delta(x_i)\}$  is a set of new predicates that replace the old predicates in the design pattern.

For each class  $c \in C$ , when it has one and only one instance, that is,  $\delta$  is a one–one function from  $C$  to  $C'$ , we call it *instantiation* or *single instantiation*. On the other hand, when  $c$  may have more than one instance, that is,  $\delta$  is a one–many correspondence from  $C$  to  $C'$ , we call it *multiple-instantiation*.

### 3.3. Structural integration

The combination of two or more patterns is called integration. We describe the integration of two structural contracts by the following definition.

**DEFINITION 3.3 (INTEGRATION FUNCTION).** *Let  $SC_i = \langle C_i, AV_i, M_i, T_i, AR_i, PS_i \rangle$ ,  $i = 1, 2$  be two structural contracts. Then, the integration of  $SC_1$  and  $SC_2$  denoted by  $SC = \langle C, AV, M, T, AR, PS \rangle$  is defined by a function  $\sigma$*

$$\begin{aligned} \sigma : C_1 \cup C_2 &\rightarrow C, AV_1 \cup AV_2 \rightarrow AV, M_1 \cup M_2 \rightarrow M. \\ T_1 \cup T_2 &\rightarrow T, AR_1 \cup AR_2 \rightarrow AR, PS_1 \cup PS_2 \rightarrow PS. \end{aligned}$$

And that

$$(1) C = \sigma(C_1) \cup \sigma(C_2), \text{ and } \forall c \in C_i (i = 1, 2)$$

$$\sigma(c) \in C_1 \cup C_2. \quad (\text{condition 1})$$

$$(\sigma(c) \in C_i) \rightarrow (\sigma(c) = c), \quad (\text{condition 2})$$

$$(2) AV = AV_1 \cup AV_2, \text{ here we assume that } AV_1 \cap AV_2 = \emptyset, \text{ where the intersection of } AV_1 \text{ and } AV_2 \text{ is an empty set. This assumption is feasible because if two attributes from } AV_1 \text{ and } AV_2 \text{ have the same name, we can rename them by adding their class names as prefix and hence their names will be different.}$$

$$(3) M = M_1 \cup M_2, \text{ for the same reason as above, we can reasonably assume that } M_1 \cap M_2 = \emptyset.$$

$$(4) T = \sigma(T_1) \cup \sigma(T_2), \text{ and } \forall t \in T_i - C_i : \sigma(t) = t,$$

$$(5) AR = AR_1 \cup AR_2.$$

$$(6) PS = \sigma(PS_1) \cup \sigma(PS_2), \text{ and } \forall p(x_1, \dots, x_n) \in PS_i : \sigma(p(x_1, \dots, x_n)) = p(\sigma(x_1), \dots, \sigma(x_n)).$$

According to the above definition, the integration of two patterns is actually described by the integration function  $\sigma$ . Informally, the integration function of pattern  $SC_1$  and pattern  $SC_2$  maps some classes of  $SC_1$  to those of  $SC_2$  and maps some classes of  $SC_2$  to those of  $SC_1$ , respectively. The rest of class names is unchanged. We will prove this point in the following lemma. We name the classes that are mapped into the other pattern as integration participants, which are defined as follows.

**DEFINITION 3.4 (INTEGRATION PARTICIPANTS).** *All symbols are based on Definition 3.3. Let*

$$X_1 = \{\forall c \in C_1 | \sigma(c) \in C_2\}.$$

$X_1$  is the set of classes in pattern  $SC_1$  that participate in the integration. Let

$$X_2 = \{\forall c \in C_2 | \sigma(c) \in C_1\}.$$

$X_2$  is the set of classes in pattern  $SC_2$  that participate in the integration. We call  $X_1$  and  $X_2$  integration participants of patterns  $SC_1$  and  $SC_2$ , respectively.

**LEMMA 3.1.** *Based on the symbols of Definition 3.4, we have  $\forall z \in C_i - X_i : \sigma(z) = z (i = 1, 2)$ .*

*Proof.* Based on the definition of  $\sigma$  and  $X_i$  in Definitions 3.3 and 3.4, respectively, let  $z \in C_1 - X_1$ . By Definition 3.4 of  $X_1$ , we conclude that

$$\forall c \in C_1 : c \notin X_1 \rightarrow \sigma(c) \notin C_2.$$

Hence,  $\forall z \in C_1 - X_1 : \sigma(z) \notin C_2$ .

Considering condition 1 of Definition 3.3, we have  $\sigma(z) \in C_1$ .

Hence by condition 2 of Definition 3.3,  $\sigma(z) = z$ .

A similar proof can be given for the case of  $i = 2$ .  $\square$

Lemma 3.1 tells us the classes that do not participate in both patterns are not changed by the integration function in the integration of two patterns.

### 3.4. Structural commutability of instantiation and integration

Given the definitions of structural instantiation and integration of design patterns, let us consider their structural commutability.

**THEOREM 3.1.** *Let  $SC_i = \langle C_i, AV_i, M_i, T_i, AR_i, PS_i \rangle$   $i = 1, 2$ . Let  $\delta_1$  be the instantiation functions from  $SC_1$  to  $SC'_1$  and  $\delta_2$  be the instantiation functions from  $SC_2$  to  $SC'_2$ . Let  $\sigma$  be an integration function. Let  $X_i$  and  $Y_i$  be the classes of  $SC_i$  and  $SC'_i$  which participate in the integration, respectively. If the following two conditions are satisfied,*

$$(1) \forall x_1 \in X_1, \delta_2(\sigma(x_1)) = \sigma(\delta_1(x_1))$$

$$\forall x_2 \in X_2, \delta_1(\sigma(x_2)) = \sigma(\delta_2(x_2))$$

$$(2) |X_i| = |Y_i|$$

then we have

$$\forall c \in C_i - X_i : \delta_i(\sigma(c)) = \sigma(\delta_i(c)), i = 1, 2.$$

*Proof.* Let  $P_i = C_i - X_i$ . According to Lemma 3.1, we have  $\forall c \in P_i, \sigma(c) = c$ . Thus  $\delta_i(\sigma(c)) = \delta_i(c)$ .

Let  $c \in P_1$ . We will prove that  $\delta_1(c) \in C'_1 - Y_1$  by contradiction.

Let  $X_1 = \{x_1, \dots, x_n\}$ . By (1), we know  $\delta_2(\sigma(x_i)) = \sigma(\delta_1(x_i))$ . This is to say  $\sigma(\delta_1(x_i)) \in C'_2 (i = 1, \dots, n)$ . By Definition 3.4, we have  $\delta_1(x_i) \in Y_1$ . Hence  $Y_1 \supseteq \{\delta_1(x_1), \dots, \delta_1(x_n)\}$ . Consider (2),  $\exists n, s.t., |X_1| = |Y_1| = n$ . Notice that  $\delta_1$  is a one–one mapping because it is a single instantiation, we have  $\forall i \neq j : \delta_1(x_i) \neq \delta_1(x_j)$ . Hence we have  $Y_1 = \{\delta_1(x_1), \dots, \delta_1(x_n)\}$ . If  $\delta_1(c) \in Y_1$ , that is  $\exists k \in \{1, \dots, n\} : \delta_1(c) = \delta_1(x_k)$ . Since  $\delta_1$  is a one–one mapping (single instantiation), we have  $c = x_k \in X_1$ . This contradicts with

$c \in C_1 - X_1$ . Thus  $\delta_1(c) \notin Y_1$ . That is  $\delta_1(c) \in C'_1 - Y_1$ . And by Lemma 3.1, we know

$$\delta_1(c) = \sigma(\delta_1(c)).$$

Notice that  $\sigma(c) = c$ , we have  $\delta_1(\sigma(c)) = \sigma(\delta_1(c))$ . That is  $\forall c \in C_1 - X_1 : \delta_1(\sigma(c)) = \sigma(\delta_1(c))$ .

Similar result can be applied to  $c \in P_2$ . That is

$$\forall c \in C_2 - X_2 : \delta_2(\sigma(c)) = \sigma(\delta_2(c)). \quad \square$$

Informally, Theorem 3.1 states that if the instantiation and integration are commutable for the integration participants ( $X_1$  and  $X_2$ ) of  $SC_1$  and  $SC_2$ , then they are commutable for all the classes ( $C_1$  and  $C_2$ ) of  $SC_1$  and  $SC_2$ .

**DEFINITION 3.5 (INSTANTIATION FUNCTION OF INTEGRATION).** Let  $\delta_i (i = 1, 2)$  be the instantiation function (one-one) of  $SC_i$ . Let  $SC$  be the integration of  $SC_1$  and  $SC_2$ . Then we derive the instantiation function of  $SC$  as:  $\delta(c) = \delta_i(c)$ , if  $c \in C_1$ .

Note that the value of subscript 'i' is 1 or 2 in the rest of this section.

**COROLLARY 3.1.** Let  $SC_1$  and  $SC_2$  be two structural contracts, and their instantiation function be  $\delta_1$  and  $\delta_2$ , respectively. Let  $\delta$  be the one-one instantiation function on  $SC$  derived from  $\delta_1$  and  $\delta_2$ , which is stated in Definition 3.5. Let  $X_i$  and  $Y_i$  be the classes of  $SC_i$  and  $SC'_i$  which participate in the integration, respectively. Let  $SC$  be the integrated structural contract of  $SC_1$  and  $SC_2$  and  $\sigma$  be the integration function. If the following two conditions are satisfied:

- (1)  $\forall x_i \in X_i, \delta(\sigma(x_i)) = \sigma(\delta(x_i))$ .
- (2)  $|X_i| = |Y_i|$

then we have  $\forall c \in C_i : \delta(\sigma(c)) = \sigma(\delta(c))$ .

*Proof.* Consider Theorem 3.1, we have

$$\forall c \in C_1 - X_1 : \delta_1(\sigma(c)) = \sigma(\delta_1(c)).$$

Since by Definition 3.3,  $\delta(\sigma(c)) = \delta_1(\sigma(c))$ , and  $\delta(c) = \delta_1(c)$ . Hence we have

$$\forall c \in C_1 - X_1 : \delta(\sigma(c)) = \sigma(\delta(c)).$$

Together with condition (1), we have

$$\forall c \in C_1 : \delta(\sigma(c)) = \sigma(\delta(c)).$$

Similarly, we have

$$\forall c \in C_2 : \delta(\sigma(c)) = \sigma(\delta(c)). \quad \square$$

**COROLLARY 3.2.** Under the same conditions provided by Corollary 3.1, we have that  $\forall p(x_1, \dots, x_n) \in PS_i$ :

$$\delta(\sigma(p(x_1, \dots, x_n))) = \sigma(\delta(p(x_1, \dots, x_n))).$$

$\forall atr \in AV_i$ :

$$\delta(\sigma(atr)) = \sigma(\delta(atr)).$$

$\forall mth \in M_i$ :

$$\delta(\sigma(mth)) = \sigma(\delta(mth)).$$

*Proof.* By Definitions 3.2 and 3.3, we have

$$\delta(\sigma(p(x_1, \dots, x_n))) = p(\delta(\sigma(x_1)), \dots, \delta(\sigma(x_n))),$$

and

$$\sigma(\delta(p(x_1, \dots, x_n))) = p(\sigma(\delta(x_1)), \dots, \sigma(\delta(x_n))).$$

For  $1 \leq k \leq n$ , if  $x_k \in C_i$ , according to Corollary 3.1, we have  $\delta(\sigma(x_k)) = \sigma(\delta(x_k))$ .

If  $x_k \in AV_i$  or  $x_k \in M_i$ , according to Definition 3.3,

$$\delta(\sigma(x_k)) = \sigma(\delta(x_k)) = \delta(x_k).$$

If  $x_k \in T_i$ , according to Definition 3.3 and Corollary 3.1, we have  $\delta(\sigma(x_k)) = \sigma(\delta(x_k))$ .

Hence we have for all possible value of  $x_k$ .

$$\forall 1 \leq k \leq n : \delta(\sigma(x_k)) = \sigma(\delta(x_k)).$$

This is to say

$$\delta(\sigma(p(x_1, \dots, x_n))) = \sigma(\delta(p(x_1, \dots, x_n))).$$

For a given attribute  $atr$ , let us suppose  $atr$  belongs to a class  $c$  which does not participate in the integration. Then  $\sigma(c) = c$  and  $\sigma(atr) = atr$ . Since  $\forall c \in C_i : \delta(\sigma(c)) = \sigma(\delta(c))$ , we have  $\delta(c) = \sigma(\delta(c))$ . This is to say, the instance of class  $c$  is not changed by the integration. Naturally, the attribute of  $\delta(c)$  will not be changed by the integration, that is,  $\delta(atr) = \sigma(\delta(atr))$ . As a whole, we obtain  $\delta(\sigma(atr)) = \sigma(\delta(atr))$ . Suppose  $atr$  belongs to a class which participates in the integration, by definition  $atr$  should appear as  $atr$  in the integration. (Note that in Definition 3.3, we have already made the assumption that the attribute names in different class of the integration will be different. This definition is feasible, because we could add the class name as the prefix of an attribute name if there is any name confliction.) That is  $\sigma(atr) = atr$ . Since  $\delta(c)$  must participate in the integration, we have  $\sigma(\delta(atr)) = \delta(atr)$ . As a whole, we obtain  $\forall(\sigma(atr)) = \sigma(\delta(atr))$ . Therefore,  $\forall atr \in AV_1 : \delta(\sigma(atr)) = \sigma(\delta(atr))$ .

Similar reasoning can be applied for any method  $mth$ , and we obtain

$$\forall mth \in M_i : \delta(\sigma(mth)) = \sigma(\delta(mth)).$$

Considering Definition 3.3 ( $\forall ty \in T_i - C_i : \sigma(ty) = ty$ ) and Definition 3.2 ( $\forall ty \in T_i - C_i : \delta(ty) = ty$ ) then it is easy to conclude that  $\forall ty \in T_i - C_i : \sigma(\delta(ty)) = \delta(\sigma(ty)) = ty$ . Since we have already proved that  $\forall ty \in C_i : \sigma(\delta(ty)) = \delta(\sigma(ty)) = ty$ , as a whole we have  $\forall ty \in T_i : \sigma(\delta(ty)) = \delta(\sigma(ty)) = ty$ .

Finally,  $\forall acr \in AR_i : \delta(\sigma(acr)) = \sigma(\delta(acr))$  is true because neither integration nor instantiation change access rights.  $\square$

Corollaries 3.1 and 3.2 actually proved the commutability of single instantiation and integration of classes  $C$  and predicates  $PS$  of structural contract  $SC$ . This point is formally summarized in the following theorem.

**THEOREM 3.2 (STRUCTURAL COMMUTABILITY).** *Under the conditions described in Theorem 3.1, we have*

$$\sigma(\delta(SC_i)) = \delta(\sigma(SC_i)).$$

*Proof.* This is an obvious conclusion from Corollaries 3.1 and 3.2.  $\square$

After we discussed the structural commutability of integration and single instantiation, let us study the commutability between integration and multiple-instantiation.

**THEOREM 3.3.** *Let  $SC_i = \langle C_i, AV_i, M_i, T_i, AR_i, PS_i \rangle$   $i = 1, 2$ . Let  $\delta_1$  be the multiple-instantiation function from  $SC_1$  to  $SC'_2$ . Let  $\delta_2$  be the multiple-instantiation function from  $SC_2$  to  $SC'_1$ . Let  $\delta$  be the unified instantiation function. Let  $\sigma$  be an integration function. Let  $X_i$  and  $Y_i$  be the classes of  $SC_i$  and  $SC'_i$  which participate in the integration, respectively. If the following three conditions are satisfied:*

- (1)  $\forall x \in X_i, \delta(\sigma(x)) = \sigma(\delta(x))$ , where  $\delta(\sigma(x))$  and  $\sigma(\delta(x))$  are two sets of class names. This equivalence between them means they have identical elements.
- (2)  $|\bigcup_{x \in X_i} \delta(x)| = |Y_i|$ . This means the number of integration participants of  $SC'_i$  equals to the number of the instances of  $X_i$ .
- (3)  $\forall x_1, x_2 \in C_1 \cup C_2, \delta(x_1) \cap \delta(x_2) = \emptyset$ . This means that if two classes are different, their instances must be different from each other.

then we have:

$$\forall c \in C_i (i = 1, 2) : \delta(\sigma(c)) = \sigma(\delta(c)).$$

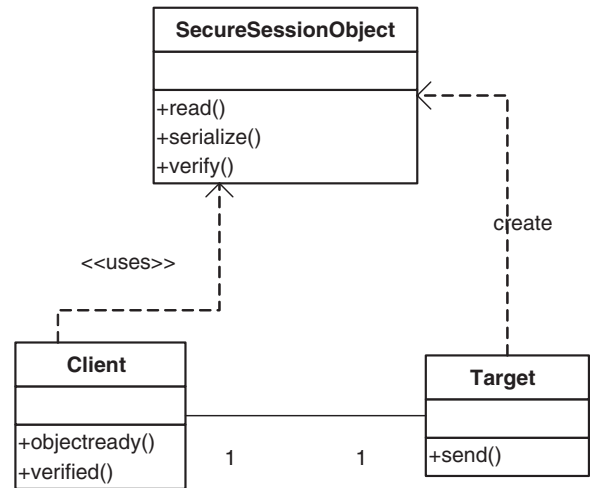
*Proof.* First we prove that  $\bigcup_{x \in X_i} \delta(x) = Y_i$ . For  $\forall x \in X_i$ , by definition,  $x \neq \sigma(x)$ . Then by (3) we know  $\delta(x) \neq \delta(\sigma(x))$ . Given (1),  $\delta(\sigma(x)) = \sigma(\delta(x))$ , we have  $\delta(x) \neq \sigma(\delta(x))$ , that is  $\delta(x) \in Y_i$ . Hence,  $\bigcup_{x \in X_i} \delta(x) \subseteq Y_i$ . Considering (2), we have  $\bigcup_{x \in X_i} \delta(x) = Y_i$ .

Considering that  $\forall c \in C_i - X_i$ , then  $\sigma(c) = c$ . Thus  $\delta(\sigma(c)) = \delta(c)$ . Now let us prove  $\sigma(\delta(c)) = \delta(c)$ . To do this, we need to prove that  $\delta(c) \cap Y_i = \emptyset$ , that is, the instances of class  $c$  do not take part in the integration of  $SC_i$ . By way of contradiction, let us assume that  $\delta(c) \cap Y_i \neq \emptyset$ , that is, there must be some  $x \in X_i$  such that  $\delta(c) \cap \delta(x) \neq \emptyset$ . Since  $c \neq x$ , by (3) we know  $\delta(c) \cap \delta(x) = \emptyset$ , which is a contradiction. Thus, the assumption is incorrect and we must have  $\delta(c) \cap Y_i = \emptyset$ , and, by definition, this means that  $\delta(c) \cap Y_i = \emptyset$ . Hence  $\sigma(\delta(c)) = \delta(c) = \delta(\sigma(c))$ . Given (1), we have  $\forall c \in C_i : \delta(\sigma(c)) = \sigma(\delta(c))$ .  $\square$

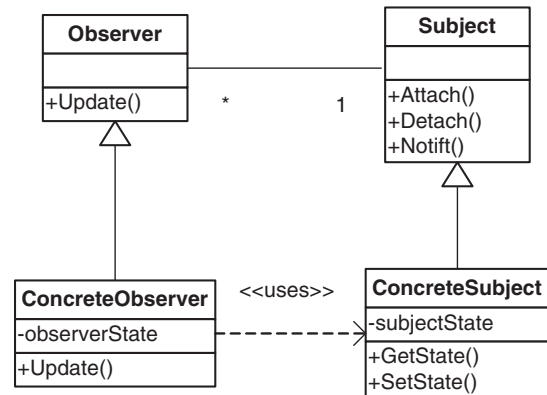
**EXAMPLE 3.2.** Consider an example of the structural commutability in the integration and instantiation of the Secure Session Object (SSO) pattern [13] and the Observer pattern [1]. The class diagrams of the SSO and Observer patterns are shown in Figs 6 and 7, respectively.

The SSO pattern is normally used to build trust between Client and Target. The Target creates the SecureSessionObject that stores the identity of the Client. Before each communication, the Client only needs to call the public operation ‘send’ of the Target, to let the Target retrieve its information from the SSO and verify the Client’s identity. Therefore, the Target would maintain trustworthiness with the Client in their later conversation. The Observer pattern provides a flexible way to maintain the consistency of different copies of data displayed in different views. The Subject stores data and maintains its consistency, while the Observers update and display data in their own ways. Let us consider the integration of the SSO pattern and the Observer pattern. Our goal is to assure that the true identity of the Observer is exposed to the Subject. This could prevent the Observer passing malicious data to the Subject that will broadcast to other good Observers and cause the wide spread of malicious data. The integration of the SSO pattern and the Observer pattern can achieve this goal.

The structure of the integrated patterns is shown in Fig. 8.



**FIGURE 6.** Class diagram of the Secure Session Object pattern.



**FIGURE 7.** Class diagram of the Observer pattern.

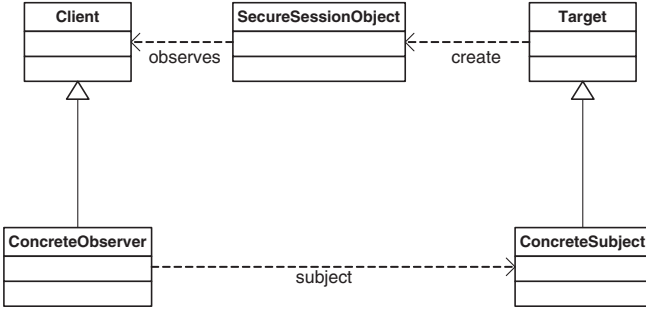


FIGURE 8. Class diagram of the integration.

To formally specify the integration, we introduce integration function  $\sigma$ , and apply it to the classes of those two patterns, respectively. Let  $C_{ob}$  and  $C_{ss}$  be the classes of the Observer pattern and the SSO pattern, respectively. Let  $C_1$  be the class of their integration.

$C_{ob} = \{\text{Observer, Subject, ConcreteSubject, Concrete Observer}\}$

$C_{ss} = \{\text{Target, Client, SecureSessionObject}\}$ .

The application of function  $\sigma$ , defined by Definition 3.3, is  $\sigma(\text{Observer}) = \text{Client}$ ,  $\sigma(\text{Subject}) = \text{Target}$ .  $\sigma$  is identical mapping on other classes of  $C_{ob}$  and  $C_{ss}$ . That is, function  $\sigma$  maps classes Observer and Subject into the SSO pattern, when these two patterns are integrated. By Definition 3.3, we have  $C_1 = \sigma(C_{ob}) \cup \sigma(C_{ss}) = \{\text{Client, ConcreteObserver, SecureSessionObject, Target, ConcreteSubject}\}$ .

According to Corollary 3.1, as long as the instantiation, characterized by instantiation function  $\delta$ , satisfies  $\sigma(\delta(\text{Observer})) = \delta(\sigma(\text{Observer}))$ , and  $\sigma(\delta(\text{Subject})) = \delta(\sigma(\text{Subject}))$ , the structural integration and instantiation is commutable. More specifically, suppose the Client and Observer will be renamed to myClient and myObserver, respectively, after instantiation. Consider the fact that  $\sigma(\text{Observer}) = \text{Client}$ , we would have  $\delta(\sigma(\text{Observer})) = \text{myClient}$ . Therefore the condition  $\delta(\sigma(\text{Observer})) = \delta(\sigma(\text{Observer}))$  requires  $\sigma(\text{myObserver}) = \text{myClient}$ . Informally, this is to require the topology of the integration of the instances remains the same as that of the patterns.

The structural commutability only completes half of the task of commutability analysis. Our next step is to analyze whether the behavioral integration and instantiation are commutable, which is presented in next section.

#### 4. BEHAVIORAL COMMUTABILITY

In contrast to the structural aspect of a design pattern contract, the behavioral contract specifies the dynamic information, such as the collaboration among the objects participating in the pattern and the creation of new objects. The behavioral contract is modeled by the collaborations of societies of objects that play different roles and work together to carry out some behavior

that is bigger than the sum of the individual behaviors. The behavioral contract is essential because the structural contract only captures the static information. However, patterns are also characterized by the interactions among the objects and operations. Unlike structural aspect of design patterns, we use the Calculus of Communication System (CCS) [14] to specify and prove the behavioral commutability due to the following reasons. Although first-order logic (FOL) is used to specify the structural aspect, it is not suitable for behavioral aspect since it cannot represent event sequence and time order. CCS has been successfully used in specifying behaviors in concurrent systems [15] and for model checking approximations [16]. Other process algebras or temporal logics may also be used to model the behavior of design patterns, which is out of the scope of this paper. In this section, we first formally define the behavioral aspect of design patterns as behavioral contract. We then define the instantiation and integration of these contracts. Finally, we present a theorem that proves the commutability of the integration and instantiation of behavioral contracts.

We present the syntax of CCS as follows:

```

formula : id '=' agent
agent   : 'nil' | id | act '.' agent | agent '+' agent | agent '|'
         agent | agent '\' restriction | '(' agent ')'
act     : id | "'" id
restriction : {id }

```

Each formula is a binding of a name and an agent. A name is a user defined id. Each agent defines an expression of actions and operators. An action name is a user defined id, which represents a system activity. The prime symbol ( $'$ ), which is placed in front of an action name, denotes that this action is an output of the system. The operations between agents include sequential composition  $'.'$ , non-deterministic choice  $'+'$ , parallel composition  $'|'$  and restriction  $'\.'$ . Restriction is defined as a set of action ids, and is used together with parallel composition, to denote that the messages being restricted are internal messages. Let us consider an example formula describing the interaction between the forward, the guard and the center in a basketball game. The center performs getPosition and then waits passBall. At the same time, the forward catches the rebound and passes it to the guard. The guard then passes the defensive players and passes the ball to the center, who completes the attack with slamDunk. The CCS expressions that model this application are shown as follows.

```

Basketball=center|guard|forward \ {passBall,passRebound}
Center=getPosition.passBall.slamDunk.Center
Forward=catchRebound.'passRebound.guardOpponent.
Forward
Guard=passRebound.passDefence.'passBall.Guard

```

##### 4.1. Behavioral contract

DEFINITION 4.1 (BEHAVIORAL CONTRACT). *All objects in the behavior of a design pattern are defined as a group of*

processes in CCS. Let  $PRT$  be the set of all process names of a design pattern. Let  $MET$  be the set of all message names. Let  $AT$  be the set of all action names in all design patterns. The behavioral aspect of a design pattern is a tuple  $BC = \langle P, IM, OM, IM_1, OM_1, A \rangle$  where

- $P \subseteq PRT$  is a finite set of process names. Each process in  $P$  corresponds to a CCS expression describing the behavioral aspect of the process. Let  $SC = \langle C, AV, M, T, AR, PS \rangle$  be the corresponding structural contract, then  $C$  is a subset of  $P$ , i.e. each class corresponds to a process in behavioral contracts. There are process names which do not correspond to any class names in the structural contract.
- $IM \subseteq MET$  is a finite set of input messages sent to the processes in  $P$ .
- $OM \subseteq MET$  is a finite set of output messages sent from the processes in  $P$ .
- $IM_1 \subseteq MET$  is the finite set of input messages sent from outside the design pattern to the processes in  $P$ .
- $OM_1 \subseteq MET$  is the finite set of output messages sent outside the design pattern from the processes in  $P$ .
- $A \subseteq AT$  is a finite set of actions that can be performed by the processes in  $P$ .

Definition 4.1 describes the elements in the behavior of a design pattern. Based on these elements, we define the CCS expression of the behavioral contract which describes the behavior of the design pattern.

**DEFINITION 4.2 (CCS EXPRESSION OF BEHAVIORAL CONTRACT).** Let  $BC = \langle P, IM, OM, IM_1, OM_1, A \rangle$  be the behavioral contract. Let  $CCSEXP$  be the set of all CCS formulas. We define a function  $G$ , which is from  $PATTERNNAMES \times PRT$  to  $CCSEXP$ , i.e. it maps a pair (pattern name, process name) into a CCS expression. Let  $X$  be the pattern name and  $p$  be a process name. The definition of  $G(X, p)$  is given below:

$$G(X, p) = \left( \sum_{i \in IM(p)} \text{in}(i) \cdot A(i) \cdot \overline{\text{out}}(o_i) \right) \cdot G(X, p),$$

where

- (1)  $IM(p)$  consists of all the input messages sent to process  $p$ ,
- (2)  $\text{in}(i)$  is the input port which takes messages as parameter and  $i$  is the input messages to the process,
- (3)  $\overline{\text{out}}(o_i)$  is the output port takes messages as parameter and  $o_i$  is the output messages from the process. In this paper,  $\overline{\text{out}}(o_i)$  is sometimes simplified as  $\text{out}(o_i)$ .
- (4)  $A(i)$  represents any action in the action set  $A$  after input message  $i$ .

Then, the CCS-process  $CCS(BC)$  of the behavioral contract  $BC$  is defined by introducing

$$CCS(BC) = \left( \prod_{p \in P} G(X, p) \right) [f] \setminus L$$

where

- (1)  $f : IM \cup OM \rightarrow IM \cup OM$  is a message relabel function that replaces one message by another message. If  $f(m_1) = m_2$ , the expression  $p[f]$  means all occurrences of message  $m_1$  in the expression of process  $p$  are replaced by message  $m_2$ .
- (2)  $L$  is the set of restricted messages. When a message  $i \in L$  is both sent out by  $\overline{\text{out}}$  of a process and received by  $\text{in}$  of another process in the same system, the message  $i$  is restricted such that it is not visible from outside. For example, let  $p = \text{in}(i) \cdot \overline{\text{out}}(j)$ ,  $q = \text{in}(j) \cdot \overline{\text{out}}(k)$ , and the set of restricted messages is  $\{j\}$ , then  $(p|q) \{j\} = \text{in}(i) \cdot \overline{\text{out}}(k)$ . This formula is based on Milner's theory in the CCS [14].

Informally, Definition 4.2 specifies that the message in  $L$  links one process with another process, when the message is the input message of the former process and the output message of the latter process.  $f$  and  $L$  work together to link one process with other process and thus formulate the CCS expression of the behavioral contract.

Let us consider an example of behavior contract specification of the SSO pattern [13] that stores the authentication and authorization information of the client in an object, so that the client will not have to carry authentication or authorization procedure repeatedly each time it wants to communicate with the Target. The sequence diagram of the SSO pattern is shown in Fig. 9. Note that our behavioral model is not restricted by sequence diagrams although this example shows our formal model is based on a sequence diagram.

**EXAMPLE 4.2 (BEHAVIORAL CONTRACT OF SSO PATTERN).** Let the behavioral contract of the SSO pattern be  $BC_{SSO} = \langle P, IM, OM, IM_1, OM_1, A \rangle$  where

$P = \{\text{Client, CreateObject, SecureSessionObject, Target}\}$ .

$IM = \{\text{access, objectready, serialize, send, verified, create, read, verify}\}$

$OM = \{\text{access, objectready, serialize, send, verified, create, read, verify}\}$

$IM_1 = \emptyset$ .

$OM_1 = \emptyset$ .

$A = \emptyset$ .

The CCS expression of each process is as follows:

$G(SSO, \text{Client}) = \text{out}(\text{access}) + \text{in}(\text{objectready}) \cdot \text{out}(\text{serialize}) \cdot \text{out}(\text{send}) + \text{in}(\text{verified})$ .

$G(SSO, \text{CreateObject}) = \text{in}(\text{create}) \cdot \text{out}(\text{read})$

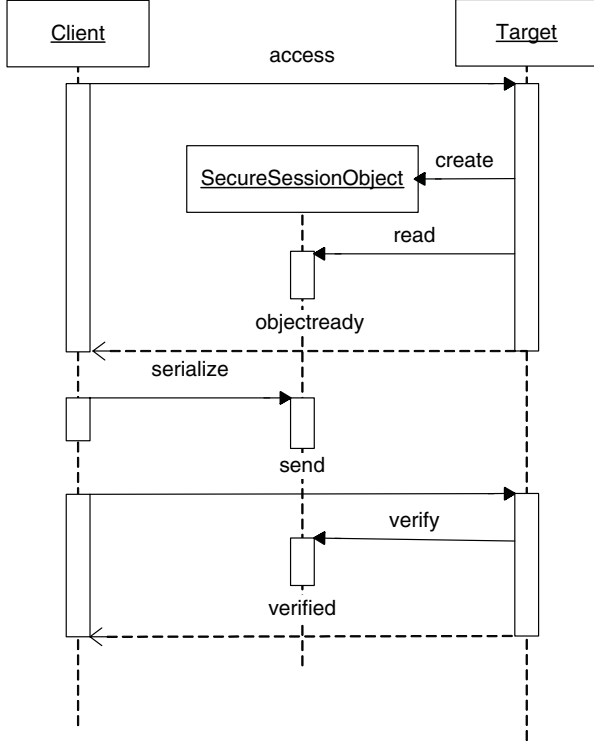


FIGURE 9. Sequence diagram of Secure Session Object.

$G(SSO, SecureSessionObject) = \text{in}(\text{read}) + \text{in}(\text{serialize})$   
 $+ \text{in}(\text{verify}).$   
 $G(SSO, Target) = \text{in}(\text{access}).\text{out}(\text{create}).\text{out}(\text{objectready})$   
 $+ \text{in}(\text{send}).\text{out}(\text{verify}).\text{out}(\text{verified}).$

The CCS expression  $BC_{SSO}$  is as follows:

$CCS(BC) =$   
 $(G(SSO, Client)|$   
 $G(SSO, CreateObject)|$   
 $G(SSO, SecureSessionObject)|$   
 $G(SSO, Target)) \setminus$   
 $\{\text{verify, verified, serialize, read,}$   
 $\text{access, create, send, objectready}\}$

## 4.2. Behavioral instantiation

In this section, we define behavioral instantiation of design patterns. In particular, we distinguish single and multiple instantiations as two different kinds of instantiation, so that we can prove the commutability separately. A participating class of a design pattern may be instantiated into one or more objects in an application.

**DEFINITION 4.3 (INSTANTIATION OF BEHAVIORAL CONTRACT).**

Let  $SC$  be a structural contract and  $\delta$  be its structural instantiation function. Let  $BC = \langle P, IM, OM, IM_I, OM_I, A \rangle$  be the behavioral contract of  $SC$ , and  $CCS(BC)$  be the activity

expression. The instance of  $BC$ ,  $BC' = \langle P', IM', OM', IM'_I, OM'_I, A' \rangle$ , is derived by extending  $\delta$  to  $BC$  and  $CCS(BC)$ .  $\delta$  is defined as a one-one correspondence in the following domains:  $IM, OM, IM_I, OM_I, A$ , and that  $IM' = \delta(IM)$ ,  $OM' = \delta(OM)$ ,  $IM'_I = \delta(IM_I)$ ,  $OM'_I = \delta(OM_I)$ ,  $A' = \delta(A)$ .

We extend  $\delta$  from class  $C$  of  $SC$  to  $P$  by adding the definition: Let  $P_1 = P - C$ , then  $p \in P_1$ ,  $\delta(p) = p$ . That is,  $\delta(P) = \delta(C) \cup P_1$ , which can be written briefly as  $P' = C' \cup P_1$ . Note that each class defined in the structural contract must have a process with the same name, which specifies its behavior. Hence the set of process names,  $P$ , is the superset of the set of class names,  $C$ . We call these process essential processes. Furthermore, there may be other processes whose names cannot be found in  $C$ . They are called intermediate processes.

We extend the domain of  $\delta$  and apply it to function  $G$ . That is,  $G$  is transformed by  $\delta$  into a new function  $\delta(G)$ , which still maps from  $PATTERNNAMES \times PRT$  to  $CCSEXP$ . However,  $\delta(G)$  is the CCS expression of the instance of a process, instead of the process. The definition is given as follows by using the definition of  $G$ . Let  $\delta(G)$  be simplified as  $G'$  and  $p$  be any process from  $P$ .

- (1) Single instantiation. Suppose  $\delta(p)$  is a single-value function, that is,  $p$  has only one instance, and let  $G(X, p) = \left( \sum_{i \in IM(p)} \text{in}(i) \cdot A(i) \cdot \overline{\text{out}}(o_i) \right) \cdot G(X, p)$ . Let us define the CCS expression of  $p'$  as  $G(X, p') = \left( \sum_{i \in IM'(p')} \text{in}(i) \cdot A(i) \cdot \overline{\text{out}}(o_i) \right) \cdot G(X, p')$ . Here the difference between  $G(X, p)$  and  $G(X, p')$  is that only the message in  $G(X, p)$  is replaced by a new message name. For example, if the input message for process  $p$  is  $i$  in the generic pattern  $X$ , whose instance  $p'$  of pattern  $X$  maps the input message  $i$  to  $j$ , then each occurrence of  $i$  in  $G(X, p)$  is replaced by  $j$ . The instance of  $CCS(BC)$  is  $\delta(CC(BC)) = \left( \prod_{p \in P} G(X, p') \right) [\delta(f)] \delta(L)$ .
- (2) Multiple-instantiation. Suppose  $\delta(p)$  is a multiple-value function, that is,  $p$  has more than one instances. Let  $\delta(p) = p'$ , where  $p'$  is a set of multiple instances of  $p$ . If the interaction among these instances is parallel, i.e. all instances behave simultaneously, then the definition of  $G'$  is  $G'(X, p) = \prod_{i \in p'} G(X, i)$ . If these instances  $p'$  are exclusive and non-deterministic choices, then the definition of  $G'$  is  $G'(X, p) = \sum_{i \in p'} G(X, i)$ . If these instances are ordered sequentially, then the definition of  $G'$  is  $G'(X, p) = \bullet_{i \in p'} G(X, i)$ . Multiple instances represent the cases when the instantiation of a design pattern maps one process to multiple processes that interact with each other in different ways. For example, there can be multiple traversals on the same aggregate at the same time in the Iterator pattern. The process instances of the ConcreteStrategy in the Strategy pattern are non-deterministic choices because only one algorithm is used at a given time. In contrast, the process

instances of the ConcreteState in the State pattern are sequentially composed which represents an order of states.

Based on the definition of  $G'$ , we provide the important definition of the extension of function  $\delta$  on  $CCS(BC)$ , i.e. the instance of  $CCS(BC)$ , as follows

$$\delta(CCS(BC)) = \left( \prod_{p \in P} G'(X, p) \right) [\delta(f)] \setminus \delta(L).$$

No matter a single instance or multiple instances a process has, the following definition provides a way to formulize the instantiation in a uniform way,

$$\delta(CCS(BC)) = \left( \prod_{p \in P} G'(X, p) \right) [\delta(f)] \setminus \delta(L),$$

where

$$G'(X, p) = G(X, p')$$

if  $p'$  is the single instance of  $p$ . Here,  $\delta(f)$  is a new message relabel function,

$$\delta(f) : IM' \cup OM' \rightarrow IM' \cup OM'$$

To be more specific,  $\forall m \in IM \cup OM$ ,  $\delta(f)$  is applied on the instances of  $m$ ,  $\delta(m)$ , with the following definition:  $\delta(f)(\delta(m)) = \delta(f(m))$ .

**EXAMPLE 4.3.** Consider the  $SSO$  pattern described in Example 4.2. The structural contract includes a set of class names,  $C = \{\text{Client}, \text{SecureSessionObject}, \text{Target}\}$ . The behavioral contract includes a set of processes,  $P = \{\text{Client}, \text{CreateObject}, \text{SecureSessionObject}, \text{Target}\}$ . Therefore, the intermediate processes are  $P_1 = \{\text{CreateObject}\}$  based on Definition 4.3. Suppose that the structural instantiation is defined as follows:

$$\begin{aligned} \delta(\text{Client}) &= \text{myData}, \\ \delta(\text{Target}) &= \{\text{myView1}, \text{myView2}\}, \\ \delta(\text{SecureSessionObject}) &= \text{SecureSessionObject}. \end{aligned}$$

Then, the instances of the processes of behavioral contract is

$$\begin{aligned} \delta(P) &= \delta(C) \cup P_1 \\ &= \{\text{myData}, \text{myView1}, \text{myView2}, \\ &\quad \text{SecureSessionObject}, \text{CreateObject}\}. \end{aligned}$$

We should then decide what  $\delta(G)$  would be. That is, what are the instances of the  $CCS$  expressions of the behavioral contract of the  $SSO$  pattern? Let us assume the instance of  $IM$  to be the

same as  $IM$ . That is  $IM' = IM$ . Let  $\delta(G)$  be simplified as  $G'$ . Consider the fact that

$$\begin{aligned} \delta(\text{Client}) &= \text{myData}, \\ \delta(\text{SecureSessionObject}) &= \text{SecureSessionObject}. \end{aligned}$$

are single instantiation, hence we have

$$G'(SSO, \text{Client}) = G(SSO, \text{myData}).$$

Consider the assumption that  $IM' = IM$ , hence we have

$$G(SSO, \text{myData}) = G(SSO, \text{Client}).$$

That is, eventually we have

$$G'(SSO, \text{Client}) = G(SSO, \text{Client}).$$

Similarly we have

$$\begin{aligned} G'(SSO, \text{SecureSessionObject}) \\ &= G(SSO, \text{SecureSessionObject}). \end{aligned}$$

Consider that  $\delta(\text{Target}) = \{\text{myView1}, \text{myView2}\}$  is multiple-instantiation, and we assume  $\text{myView1}$  and  $\text{myView2}$  are two processes that happened by non-deterministic choice, then we have

$$G'(SSO, \text{Target}) = \sum_{i \in \{\text{myView1}, \text{myView2}\}} G'(SSO, i).$$

That is to say,

$$G'(SSO, \text{Target}) = G(SSO, \text{myView1}) + G(SSO, \text{myView2}).$$

Eventually, we can derive the  $CCS$  expression of the instance of  $CCS(BC)$  as

$$\begin{aligned} \delta(CCS(BC^{\text{SSO}})) &= (G'(SSO, \text{Target}) | \\ &\quad G'(SSO, \text{Client}) | \\ &\quad G'(SSO, \text{CreateObject}) | \\ &\quad G'(\text{SecureSessionObject})) \setminus \delta(L). \end{aligned}$$

Again, consider the fact that  $IM' = IM$ , which means actually the name of message is not changed in the instantiation. Thus, the restriction set will be the same as in  $CCS(BC)$ , that is  $\delta(L) = \{\text{verify}, \text{verified}, \text{serialize}, \text{read}, \text{access}, \text{create}, \text{send}, \text{objectready}\}$ .

### 4.3. Behavioral integration

**DEFINITION 4.4 (INTEGRATION OF BEHAVIORAL CONTRACTS).** Let  $SC^i = \langle C^i, AV^i, M^i, T^i, AR^i, PS^i \rangle$ ,  $1 \leq i \leq n$ . be the structural contracts of  $n$  design patterns. The integration of  $SC^i$  is denoted by  $SC = \langle C, AV, M, T, AR, PS \rangle$ . Let  $\sigma$  be the integration function mapping  $SC^i$  to  $SC$ . Let

$BC^i = \langle P^i, IM^i, OM^i, IM_1^i, OM_1^i, A^i \rangle$  and  $CCS(BC^i)$  be their behavioral contracts. Then the behavioral contract of the integrated pattern denoted by  $BC = \langle P, IM, OM, IM_1, OM_1, A \rangle$  is defined by extending the domain of  $\sigma$  and applying it to  $BC^i$  and  $CCS(BC^i)$ . We consider two different kinds of behavioral integration, message integration and process integration.

- (1) *Message integration.* When two patterns are integrated only by sending and receiving messages, they are composed by message integration. In this kind of integration, there is no overlapping on the processes from different patterns. The processes of one pattern just interact with those of another pattern by exchanging messages. All internal processes of each pattern behave exactly the same before and after the integration. The  $CCS$  expression of message integration is expressed as the parallel composition of the  $CCS$  processes of each individual pattern as follows:

$$CCS(BC) = \left( \prod_{i=1}^n CCS(BC^i) \right) [f] \setminus L,$$

where the property of the integration is fully specified by  $f$  and  $L$ .  $f$  renames the outgoing message of one pattern and makes it the same as the incoming message of another pattern, thus integrating the two patterns.

- (2) *Process integration.* A more complex kind of integrations is that one or more processes of one pattern overlap with those of another pattern, which is called process integration. Based on Definition 3.3, the structural integration function  $\sigma$  may rename (map) one class name in a pattern to the same name of one class in another pattern. Therefore, some class names disappear during the integration. The corresponding processes of the disappeared classes are called the intermediate processes in the integration. That is, the intermediate processes exist but their corresponding class names are not among the class names of the integration. Here we have

$$P = \bigcup_{i=1}^n \bigcup_{p \in P^i} (i, p), \quad IM = \bigcup_{i=1}^n IM^i,$$

$$OM = \bigcup_{i=1}^n OM^i, \quad IM_1 = \left( \bigcup_{i=1}^n IM_1^i \right) - L,$$

$$OM_1 = \left( \bigcup_{i=1}^n OM_1^i \right) - L.$$

Suppose  $CCS(BC^i) = \prod_{p \in P^i} G(i, p)[f_i] \setminus L_i$ .  $CCS(BC)$  is defined as  $CCS(BC) = \prod_{(i, p) \in P} G_\sigma(i, p)[f] \setminus L$ , where  $G_\sigma$  is a function that takes in a pattern name and a process name and outputs the  $CCS$  expression of the process in the integration.  $G_\sigma$  is different from  $G$  in case that a process

may have different behavior in the integration from its behavior in its generic pattern.  $f = \sum_{i=1}^n f_i$  is the combination of the relabel functions of each pattern.  $f$  renames a message to a different name by calling relabel function  $f_i$ .  $L \supseteq \bigcup_{i=1}^n L_i$  is the restriction set of the integration, which contains as subset the union of all the restriction sets of each individual pattern.

Let us consider an example of the behavioral integration of the  $SSO$  pattern and the Observer pattern. A common application is that the server deploys the data that is accessed and displayed by the client in different ways. However, the access to the data has to be limited to a certain group of persons, who are granted with different level of rights. The advantage of using access control is to eliminate random access which can usually destruct the data or disclose sensitive information. Suppose the clients want to switch freely between multiple data display layouts while they do not want to re-confirm their access right to the server every time when they make a switch. An integration of the Observer pattern and the  $SSO$  pattern may solve these problems. The behavior of the  $SSO$  pattern has been presented in Example 4.2. The sequence diagram of the Observer pattern is shown in Fig. 10. In the following, we first specify the behavior contract of the Observer pattern. We then show the behavioral integration of the Observer pattern and the Secure Object pattern.

EXAMPLE 4.4 (PROCESS INTEGRATION). Consider the behavioral contract of the Observer pattern,  $BC^{\text{observer}}$ , whose class name and process name sets are

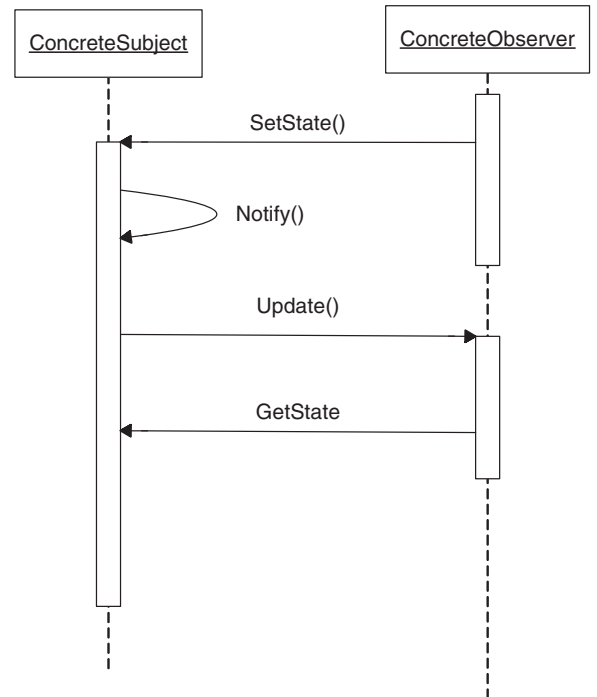


FIGURE 10. Sequence diagram of Observer pattern.

$C_{observer} = \{Subject, Observer, ConcreteSubject, Concrete Observer\}$

$P_{observer} = \{Subject, Observer, ConcreteSubject, Concrete Observer\}$

The CCS process of each process is as follows:

$G(Observer, ConcreteSubject) = (in(SetState).action(Notify).out(Update)). G(Observer, ConcreteSubject).$

$G(Observer, ConcreteObserver) = (in(Update).out(GetState) + out(SetState)). G(Observer, ConcreteObserver).$

Then,  $CCS(BC^{observer}) = G(Observer, ConcreteObserver) | G(Observer, ConcreteSubject) \setminus \{SetState, Update, GetState\}.$

Let us consider a behavioral integration of the Observer pattern and the Secure Session Object pattern, whose sequence diagram is shown in Fig. 11.

The CCS expression of the processes in the integration is defined as follows:

$G_{\sigma}(SSO, Client) = in(createC). out(access) + in(request). out(serialize). out(send) +$

$in(verified). out(requestOk) + in(objectready). out(createOk).$

$G_{\sigma}(SSO, CreateObject) = G(SSO, CreateObject).$

$G_{\sigma}(SSO, SecureSessionObject) = G(SSO, SecureSessionObject).$

$G_{\sigma}(SSO, Target) = in(createT). in(access).$

$out(create). out(objectready) + out(verified). out(verify).$

$G_{\sigma}(Observer, ConcreteSubject) = out(createT). in(SetState). action(Notify). out(Update).$

$G_{\sigma}(Observer, ConcreteObserver) = in(update). out(request) + in(requestOk). out(getState) + in(createOk). out(setState).$

Let  $P = \{(SSO, Client), (SSO, SecureSessionObject), (SSO, Target), (SSO, CreateObject), (Observer, ConcreteObserver), (Observer, ConcreteSubject)\}.$

Let  $L_{observer} = \{setState, update, getState\},$

$L_{SSO} = \{verify, verified, serialize, read, access, create, send, objectready\}$

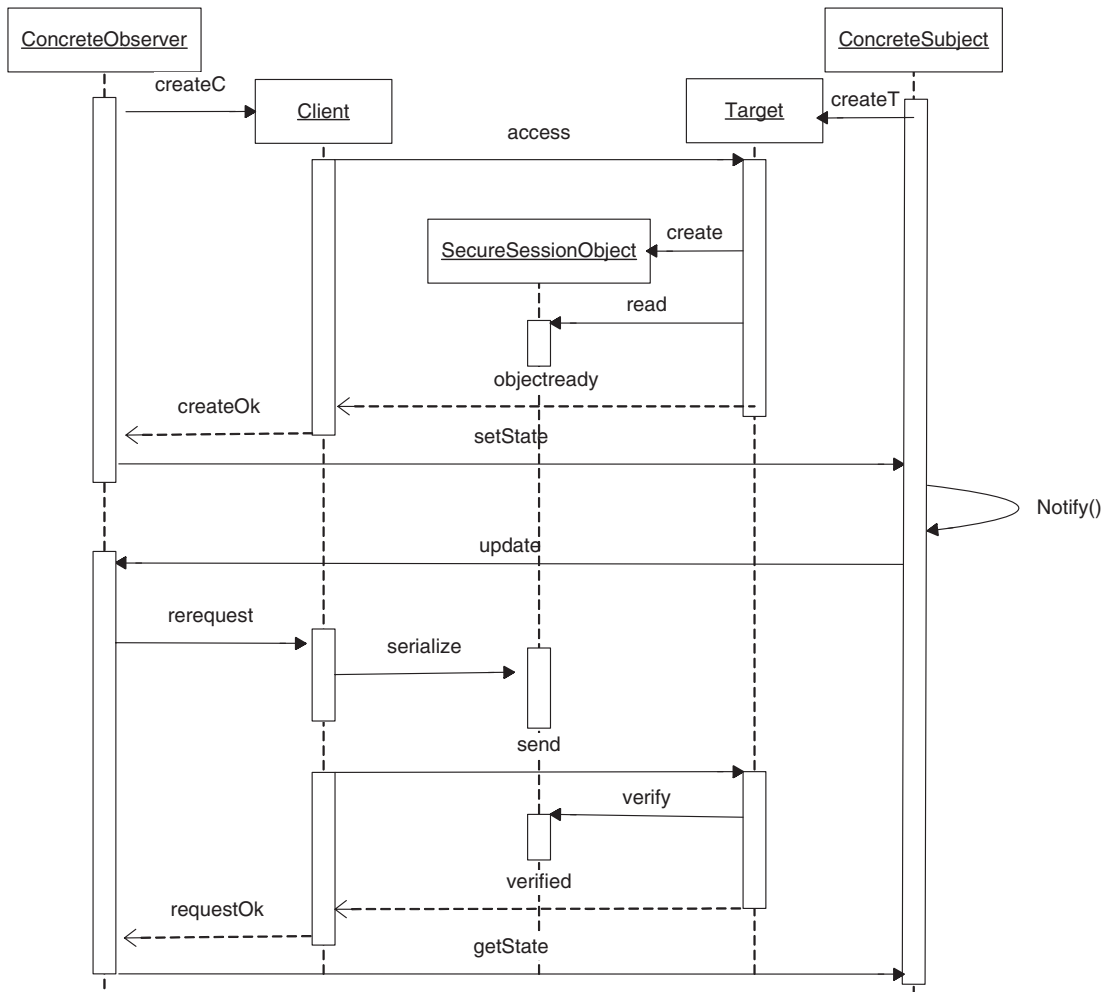


FIGURE 11. Sequence diagram of the integration.

and we have  $L = L_{SSO} \cup L_{\text{observer}} \cup \{\text{create}C, \text{create}T, \text{create}Ok, \text{request}, \text{request}Ok\}$ .

Let  $I$  denote the name of the integration. The behavior of the integration is specified by the formula as follows.

$$CCS(BC^I) = \prod_{(i,p) \in P} G_\sigma(i,p) \setminus L.$$

Note that we do not have relabel function in this specific example. At this moment, the message integration and process integration have quite different form. Through the following theorem, we can unify their expressions into one form.

**THEOREM 4.1 (EXPANSION OF MESSAGE INTEGRATION).** *Suppose that we have pattern  $1, \dots, n$  with CCS expression as*

$$CCS(BC^i) = \left( \prod_{p \in P^i} G(i,p) \right) [f_i] \setminus L_i,$$

where  $i$  refers to the pattern name.

The message integration defined in Definition 4.4 is  $CCS(BC) = \left( \prod_{i=1}^n CCS(BC^i) \right) [f] \setminus L$ ,

Then, we have the following important conclusion:

$$CCS(BC) = \left( \prod_{(i,p) \in P} G(i,p) \right) [F] \setminus L^*,$$

where

$$P = \bigcup_{i=1}^n \bigcup_{p \in P^i} (i,p), \quad F = f \circ f_i, \quad L^* = L \cup \left( f \left( \bigcup_{i=1}^n L_i \right) \right).$$

*Proof.* See the Appendix for summarized proof and Theorem 4.4 in [12] for detailed proof.  $\square$

In the above formula,  $L^*$  is the restriction set which contains all the restricted messages in  $L_i$ . By use of  $L^*$  together with  $F$  and  $P$ , we are able to formulate the integration and the individual design pattern in a uniform way. In a sense that they are both formulated as the parallel composition of process formulas.

In case of message integration, Theorem 4.1 provides a simplified form of the CCS expression of the integrated patterns. As a result, message integration and process integration can be defined by a uniform formula.

Based on this theorem, we can define the instantiation of the integrated design pattern in the following section.

#### 4.4. Behavioral commutability of instantiation and integration

We now consider whether the integration and instantiation of behavioral contracts are commutable. The behavior of each design pattern is formalized by its behavioral contract  $BC$  and its CCS processes  $CCS(BC)$ . From Definition 4.4, we know

that the integration of  $BC^i = \langle P^i, IM^i, OM^i, IM_I^i, OM_I^i, A^i \rangle$  is simply the union of sets. Hence the order of instantiation and integration of  $BC^i$  does not matter. Whether the behavioral instantiation and integration are commutable is only decided by the CCS processes of each design pattern. Therefore, we only need to consider the commutability of the CCS expressions.

We first consider that the instantiation happens after integration. This is described in the following definition.

**DEFINITION 4.5 (Instantiation of integrated behavioral contract).** *Based on the symbols used in Theorem 4.1, the instantiation of the integrated CCS expressions is defined by the following equation.*

For message integration:

$$\delta(CCS(BC)) = \left( \prod_{(i,p) \in P} G'(i,p) \right) [\delta(F)] \setminus \delta(L^*).$$

For process integration:

$$\delta(CCS(BC)) = \left( \prod_{(i,p) \in P} G'_\sigma(i,p) \right) [\delta(f)] \setminus \delta(L).$$

**EXAMPLE 4.5. (THE INSTANCE OF THE INTEGRATION DESCRIBED IN EXAMPLE 4.4).** In order to obtain  $\delta(CCS(BC^I))$ , we need to derive through the following steps. This instantiation is multiple-instantiation considering the fact that process ConcreteObserver has two instances, i.e. TableView and PieView. By considering  $\delta(\text{ConcreteObserver})$  as a multiple-instantiation, we have

$$\begin{aligned} G'_\sigma(\text{Observer}, \text{ConcreteObserver}) \\ = G_\sigma(\text{SSO}, \text{TableView}) | G_\sigma(\text{SSO}, \text{PieView}). \end{aligned}$$

Consider  $\delta$  as a single instantiation as follows, then we have

$$\begin{aligned} G'_\sigma(\text{SSO}, \text{Client}) &= G_\sigma(\text{SSO}, \text{Client}). \\ G'_\sigma(\text{SSO}, \text{CreateObject}) &= G_\sigma(\text{SSO}, \text{CreateObject}). \\ G'_\sigma(\text{SSO}, \text{SecureSessionObject}) &= G_\sigma(\text{SSO}, \\ &\quad \text{SecureSessionObject}). \\ G'_\sigma(\text{SSO}, \text{Target}) &= G_\sigma(\text{SSO}, \text{Target}). \end{aligned}$$

Figure 12 presents the sequence diagram of the instantiation of the integration. The CCS process of this instantiation of the integration is formulized as follows:

$$\delta(CCS(BC^I)) = \left( \prod_{p \in P} G'_\sigma(i,p) \right) \setminus \delta(L).$$

Since no message is changed by  $\delta$ , we have  $\delta(L) = L$ . We have derived  $L$  and  $P$  in Example 4.4.

**THEOREM 4.2.** *Instantiation and message integration are commutable. That is, let the CCS process of design pattern  $i$  be*

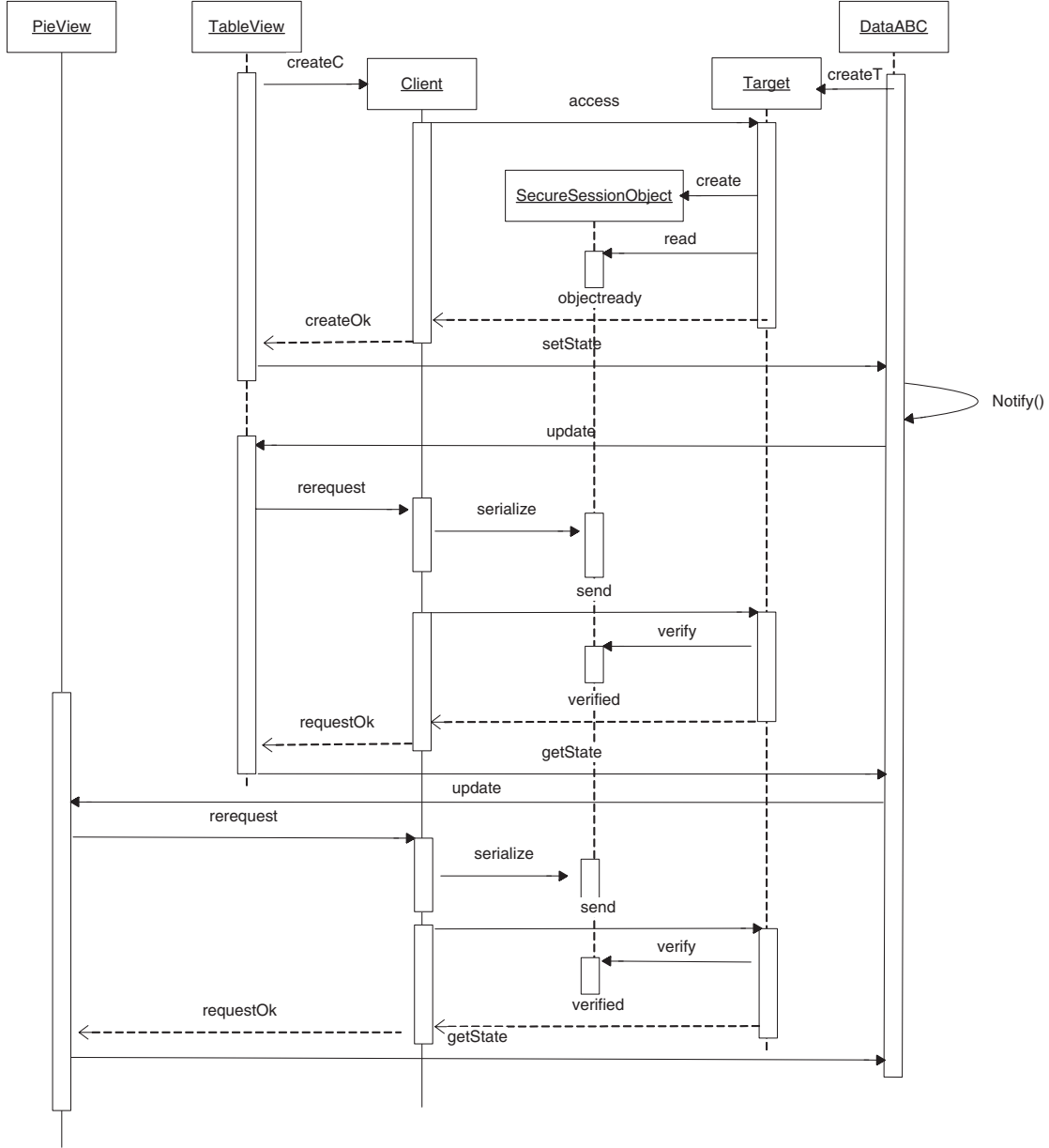


FIGURE 12. Instance of the integration of the Observer and SSO patterns.

$CCS(BC^i) = \left( \prod_{p \in P^i} G(i, p) \right) [f_i] \setminus L_i$  and the CCS process of their integration be

$$CCS(BC) = \left( \prod_{i=1}^n CCS(BC^i) \right) [f] \setminus L.$$

Let the instantiation relation  $\delta$  be given by Definition 4.3. Then we have the following conclusion

$$\delta(CCS(BC)) = \left( \prod_{i=1}^n \delta(CCS(BC^i)) \right) [\delta(f)] \setminus \delta(L).$$

The left side of the equation is the instance of an integration, whereas its right side is the integration of instances. Hence this equation actually denotes the commutability of instantiation and integration.

*Proof.* Consider we first instantiate patterns  $1, \dots, n$ , then integrate their instances.  $\square$

By Definition 4.3, we obtain the instances of  $CCS(BC^i)$  as

$$\delta(CCS(BC^i)) = \left( \prod_{p \in P^i} G'(i, p) \right) [\delta(f_i)] \setminus \delta(L_i).$$

Then the integration of  $\delta(CCS(BC^i))$  is expressed by the left part of the following equation. By Theorem 4.1,

$$\begin{aligned} & \left( \prod_{i=1}^n \delta(CCS(BC^i)) \right) [\delta(f)] \backslash \delta(L) \\ &= \left( \prod_{(j,p) \in P_1} G'(j,p) \right) [F_1] \backslash L_1^*, \quad \text{with} \end{aligned}$$

- (1)  $P_1 = \bigcup_{i=1}^n \bigcup_{p \in P^i} (i, p)$ , (2)  $F_1 = \delta(f) \circ \delta(f_i)$ , and  
(3)  $L_1^* = \delta(L) \cup (\delta(f)(\bigcup_{i=1}^n \delta(L_i)))$ .

Thus, if we first instantiate  $CCS(BC^i)$  then integrate them, the resulted CCS expression will be

$$\left( \prod_{(j,p) \in P_1} G'(j,p) \right) [F_1] \backslash L_1^*.$$

Next commute the integration and instantiation by first integrating  $CCS(BC^i)$  then instantiating the resulted integration.

The message integration of  $CCS(BC^i)$  is described by  $CCS(BC) = (\prod_{i=1}^n CCS(BC^i)) [f] \backslash L$ .

By Definition 4.5, we have  $\delta(CCS(BC)) = (\prod_{(i,p) \in P} G'(i,p)) [\delta(F)] \backslash \delta(L^*)$  where  $P = \bigcup_{i=1}^n (i, p)$ ,  $p \in P^i$ .

To prove  $(\prod_{(i,p) \in P_1} G'(i,p)) [F_1] \backslash L_1^* = (\prod_{(i,p) \in P} G'(i,p)) [\delta(F)] \backslash \delta(L^*)$ , we will first prove that  $F_1 = \delta(F)$ . Since  $F_1 = \delta(f) \circ \delta(f_i)$  and  $\delta(F) = \delta(f \circ f_i)$ , we only need to show  $\delta(f \circ f_i) = \delta(f) \circ \delta(f_i)$ . For any  $m \in IM \cup OM$ , let  $n = f_i(m)$  and  $l = f(n)$ . That is to say,  $(f \circ f_i)(m) = l$ . In Definition 4.3, for any relabel function  $f$ , we define  $\delta(f)(\delta(m)) = \delta(f(m))$ , hence we have  $\delta(f \circ f_i)(\delta(m)) = \delta(l)$ . Also from Definition 4.3, we have  $\delta(f_i)(\delta(m)) = \delta(f_i(m)) = \delta(n)$ . Hence,

$$(\delta(f) \circ \delta(f_i))(\delta(m)) = \delta(f)(\delta(n)) = \delta(f(n)) = \delta(l).$$

Hence,  $\delta(f \circ f_i) = \delta(f) \circ \delta(f_i)$ .

Then, we will prove  $L_1^* = \delta(L^*)$ . Since  $L^* = L \cup (f(\bigcup_{i=1}^n L_i))$ , we have

$$\delta(L^*) = \delta(L) \cup \delta(f(\bigcup_{i=1}^n L_i)) = \delta(L) \cup \delta(f)(\delta(\bigcup_{i=1}^n L_i)) = L_1^*.$$

Finally, it is straightforward to reach  $P = P_1 = \bigcup_{i=1}^n \bigcup_{p \in P^i} (i, p)$ .

Hence  $\delta(CCS(BC)) = (\prod_{i=1}^n \delta(CCS(BC^i))) [\delta(f)] \backslash \delta(L)$ .

Theorem 4.2 states that instantiation and message integration are commutable for behavioral contracts.

**THEOREM 4.3.** *The instantiation and process integration are commutable, provided*

- (1)  $G'_\sigma(i, p) = G'_\sigma(i, p)$ . As we know,  $G'(i, p)$  is the CCS expression of the instance of process  $i$ . Here,  $G'_\sigma(i, p)$  is what the instance becomes in the integration. As we know,  $G_\sigma(i, p)$  is what process  $i$  become in the integration. Here,  $G'_\sigma(i, p)$  is the instance of  $G_\sigma(i, p)$ .

- (2) Given  $CCS(BC) = \prod_{(i,p) \in P} G_\sigma(i, p) [f] \backslash L$  and its instance  $(\prod_{(i,p) \in P} G'_\sigma(i, p)) [\delta(f)] \backslash \delta(L)$ .

Given  $\delta(CCS(BC^i)) = (\prod_{p \in P^i} G'(i, p)) [\delta(f_i)] \backslash \delta(L_i)$  and their integration  $\prod_{(i,p) \in Q} G'_\sigma(i, p) [g] \backslash M$ .

We require  $M = \delta(L)$ . Actually, this means the instantiation of the restriction set in the integration is the same as the restriction set in the integration of the instantiation of the patterns.

*Proof.* For process integration, we have  $CCS(BC) = \prod_{(i,p) \in P} G_\sigma(i, p) [f] \backslash L$ . Suppose that we integrate first, and then instantiate. According to Definition 4.5, we have  $\delta(CCS(BC)) = (\prod_{(i,p) \in P} G'_\sigma(i, p)) [\delta(f)] \backslash \delta(L)$ .

On the other hand, if we instantiate first, then we have  $\delta(CCS(BC^i)) = (\prod_{p \in P^i} G'(i, p)) [\delta(f_i)] \backslash \delta(L_i)$ .

The integration of  $\delta(CCS(BC^i))$ , by Definition 4.4, is actually  $\prod_{(i,p) \in Q} G'_\sigma(i, p) [g] \backslash M$ , where  $Q = \bigcup_{i=1}^n \bigcup_{p \in P^i} (i, p)$ ,  $g = \sum_{i=1}^n \delta(f_i)$ , and  $M \supset \bigcup_{i=1}^n \delta(L_i)$ . Given the conditions (1) and (2) and the obvious fact that  $g = \delta(f)$ , we conclude that

$$\delta(CCS(BC)) = \left( \prod_{(i,p) \in Q} G'_\sigma(i, p) \right) [g] \backslash M.$$

□

**EXAMPLE 4.6.** In Example 4.5, we show the instance of  $CCS(BC^l)$  as

$$\delta(CCS(BC^l)) = \left( \prod_{p \in P} G'_\sigma(i, p) \right) \backslash \delta(L).$$

What would happen if we first instantiate the Observer and SSO patterns, then integrate the instances? Using the result from Theorem 4.3, we could know that

$$\delta(CCS(BC^l)) = (\delta(CCS(BC^{\text{observer}})) | \delta(CCS(BC^{\text{SSO}}))) \backslash \delta(L).$$

## 5. COMMUTABILITY APPLICATIONS

In this section, we discuss the benefits and applications of the commutability analysis to software design and analysis. We also provide guidelines to judge whether a software design satisfies commutability conditions, which may help the designers to apply our commutability theorems in their practical applications.

### 5.1. Benefits of commutability applications

In the previous sections, we have proved the structural and behavioral commutability of the instantiation and integration of design patterns in certain conditions. Our theorems allow

the designers to decide the commutability by checking the satisfactory of the conditions in practical applications. There are several benefits of applying commutability theorems in software design and development.

First, the commutability analysis can help the designers to detect defects in early stages of software development. More specifically, the designer can detect integration problems early by verifying the integration of the general design patterns before any instantiation when the integration and instantiation are commutable. When applying design patterns, it is typically hard and error prone to integrate different patterns. Thus, integrating design patterns before instantiation allows finding the integration error early in the development process.

Second, the instance of a design pattern is normally more complex than the original pattern because it includes more details. Thus, design analysis is more scalable before the instantiation than after that. When an application satisfies the commutability conditions so that the instantiation and integration are commutable, the designers can delay the instantiation after the integration. This is especially important when using automatic verification tools, such as model checker, which are sensitive to the size of the system design.

As an example, we have studied a system including the Observer pattern [1], the Authentication Enforcer pattern [13] and the Security Pipe pattern [13] in [17, 18]. Figure 13 depicts the Authentication Enforcer pattern where the AuthenticationEnforcer class centralizes and encapsulates the authentication logic. When two clients need to authenticate each other, they can achieve that through the AuthenticationEnforcer class. Figure 14 shows the class diagram of the Security Pipe pattern related to an application. Initially, the client may login the application. The application then creates a SecurePipe at the system level. The SecurePipe is an encrypted communication channel over which the client communicates with the application. The channel is secure and provides data privacy and integrity between two endpoints. When the client logs out, the application destroys the SecurePipe.

In the normal applications of the Observer pattern as shown in the previous sections, the subject and the observers are not

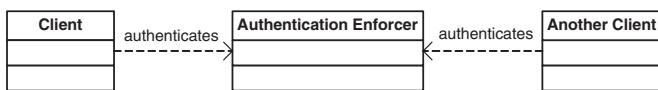


FIGURE 13. Authentication enforcer pattern.

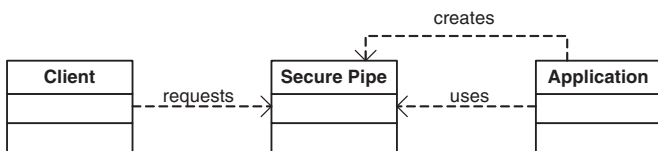


FIGURE 14. Secure Pipe pattern.

required to authenticate each other as shown in Fig. 7. The communications between them are not required to be secure either. Let us consider the example of secure observer where the subject and the observers need to be authenticated and their communications should be secure. The system must ensure the confidentiality of the data passed from the DataABC (an instance of Subject) to the TableView, PieView and BarView (instances of Observer in Fig. 15). The system must also ensure the DataABC and its different views to be authenticated. As a solution, the integration of the instances of the Observer pattern, the Secure Pipe pattern and the Authentication Enforcer pattern is shown in Fig. 16. The system ensures the confidentiality of the data passed from the Subject to the Observer by authenticating every Observer attached to the Subject. In this example, the commutability conditions are satisfied. Therefore, the designers can always integrate these patterns before instantiation, which allows them to find integration errors early.

Discovering integration errors by human observation is tedious and error-prone. Automated verification techniques, such as model checking, can help to check whether the confidentiality and authentication properties have been truthfully satisfied in this design. They can reduce human

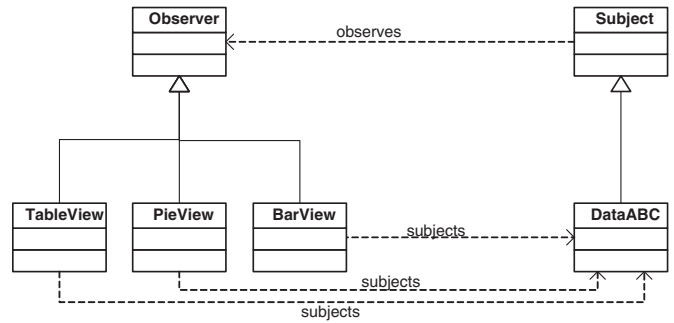


FIGURE 15. Observer pattern instance.

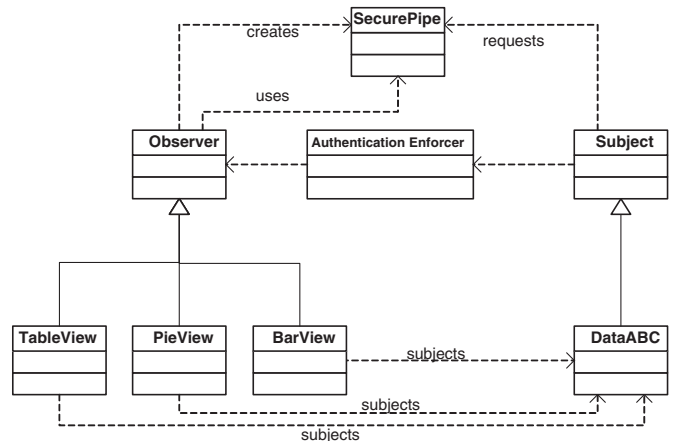


FIGURE 16. Secure and authenticated Observer (instantiation first).

errors and improve performance. However, model checking techniques have a common problem on the scalability of large systems due to the potential state space explosion. Reducing state space is a major consideration when using model checking techniques. Our study on the commutability of the instantiation and integration of design patterns can help on this task. When the instantiation and integration commute, integrating before instantiating renders smaller system designs, as for example shown in Fig. 17, which are more amendable to model checking. We are able to use a model checker to expose several security problems in the design of a secured Observer when we delay the instantiation as shown in [17, 18]. If we instantiate the

patterns before integrate them, nevertheless, the system would become too complex to resort to a model checker. More specifically, the instantiation of the Observer pattern involves introducing multiple instances of Observer. As a result, the system may contain too many states for a model checker to explore.

We should also notice that not all design satisfies the commutability conditions. Hence, we may not always be able to integrate design patterns before instantiating them. In the next section, we provide some guidelines to help the designers use our commutability theorems.

### 5.2. Guidelines for commutability applications

In this paper, we classify the instantiation of design patterns into single and multiple instantiations. This is key to determine whether a design preserve commutability.

For single instantiation, the commutability conditions can be always satisfied in the original design or in adjusted design by renaming classes. Let us review Example 3.2 which is an example of the integration of the Observer pattern and the SSO pattern. As shown in Fig. 18(b), if the instantiation happens before the integration, then the integration merges classes myObserver and myClient, which are instance of Observer and Client respectively, into myClient. As shown in Fig. 18(a), if the integration happens before the instantiation, it requires merging the Observer and the Client. If they are merged to the Observer, then myObserver is the instance of the integration of the Observer and the Client. Hence they are not

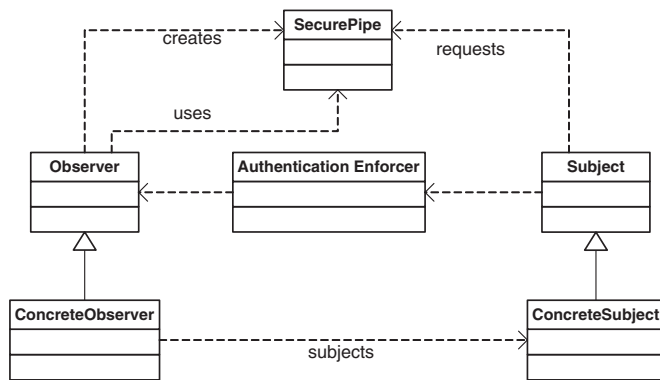


FIGURE 17. Secure and authenticated Observer (integration first).

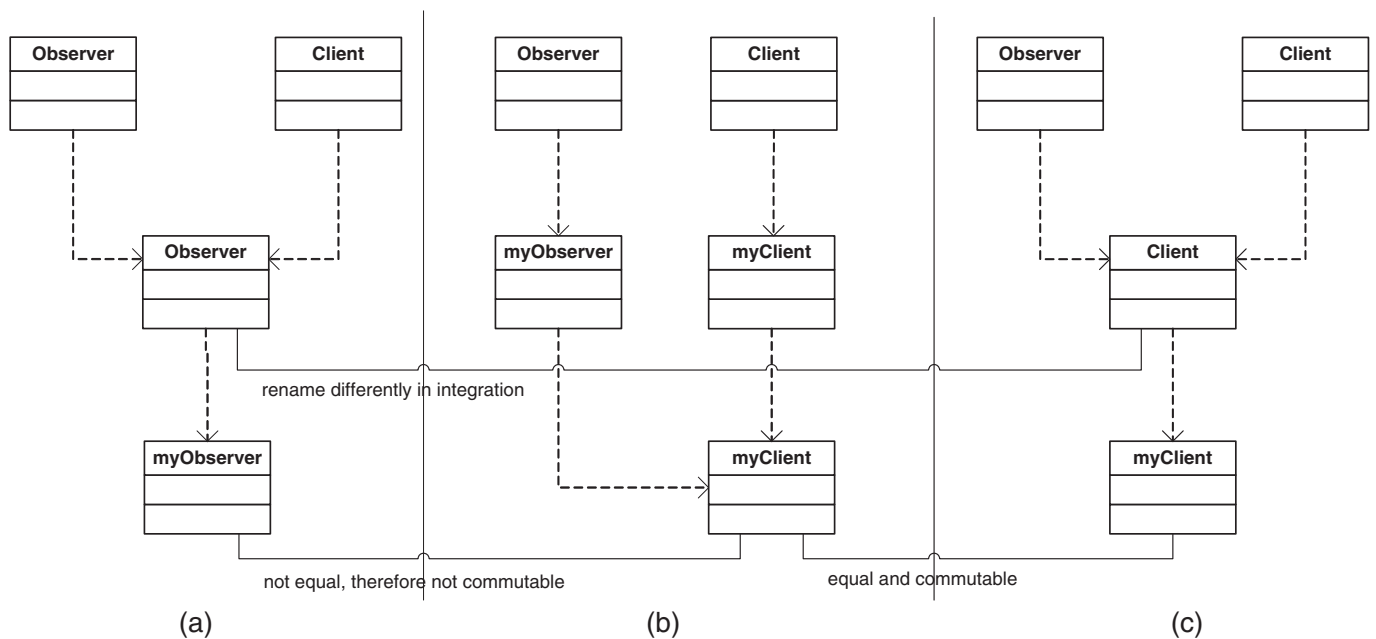


FIGURE 18. Rename example of single instantiation.

**TABLE 1.** Difficulty levels of satisfying commutability conditions.

	Message integration	Process integration
Single instantiation	Very easy	Easy
Multiple instantiation	Hard	Very hard

commutable. If the Observer and the Client are merged into the Client as shown in Fig. 18(c), however, myClient is the instance of the integration. Hence they are commutable. This example shows that the design can be adjusted by renaming the merged classes to satisfy the commutability conditions in single instantiation.

For multiple instantiations, whether the commutability conditions can be satisfied is highly dependent on the nature of multiple instantiations. In multiple instantiations, a class in the original design pattern may be instantiated into many classes. When all these instance classes participate in the integration with other patterns, the instantiation and integration are generally commutable. On the other hand, if some of the instance classes do not participate in the integration, the instantiation and integration may not be commutable. Let us consider an example of the Observer pattern with two instances of Observer, RedShow and GreenShow. The RedShow class displays the data in red color and suggests the data is transferred from an encrypted channel from the Subject. The GreenShow class shows the data in green color and suggests the data is transferred from an open channel. In practical design, the RedShow class may participate in the integration with other patterns, e.g. the Secure Pipe pattern which provides encrypted transfer. The integration may not include the GreenShow class. In this scenario, the system does not preserve commutability and it cannot lead to commutability by any adjustment. It cannot obtain commutability by any adjustment. The reason is that if integration happens first, all instances of Observer would have to either join in or be out of integration. Therefore, we would have both RedShow and GreenShow display either safe data or unsafe data, which is a problem because the example design requires RedShow and GreenShow showing only the safe and unsafe data, respectively.

In summary, we provide the evaluation of the difficulty of satisfying the commutability conditions for different types of instantiations and integrations in Table 1.

## 6. RELATED WORK

Several formal approaches on the specification and verification of design patterns have been presented in [7, 10, 11, 19–25]. To the best of our knowledge, there is no research work investigating the commutability of design pattern instantiation and integration so far. We are the first to study this problem and

provided proofs of our results based on our formal specification and verification framework [11, 12]. In this section, we discuss several other formal specification and verification approaches.

The structural and behavioral aspects of design patterns in terms of responsibilities and rewards are formally specified in [25]. Following the ideas of the design by contract approach in [26], the structural and behavioral specifications are captured as responsibilities, whereas the rewards capture the benefits of applying the pattern with the expected behavior in a system.

The composition of two design patterns based on a specification language (DisCo) has been discussed in [20]. 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. Formal specification of design patterns and their composition based on the Language of Temporal Ordering Specification is proposed in [23]. In particular, the behavioral aspect of the Command and Composite patterns and their combination is specified. While both works investigated the composition of design patterns, there is no study on its relationship to the instantiation of design patterns.

Law-governed support for realizing design patterns has been investigated in [27]. Some rules and constraints of design patterns have been defined. However, the property checking is performed at implementation level.

Taibi and Ngo [24] propose specifying the structural aspect of design patterns in the FOL and the behavioral aspect in the temporal logic of action. Similar to our ideas presented in [7], their approach concentrated on the specifications of design patterns.

Several research works [28–30] have investigated the usage of the commutativity property for concurrency control, instead of proving whether two operations are commutative. Commutativity in user interfaces of human–computer interaction has also been studied in [31]. Dennis *et al.* [32] applied model checking techniques to explore the commutativity of software operations in a therapy control system. The software operations can be considered as a set of elements that are sequentially composed. Thus, their commutativity analysis is under the sequential composition operation. Let  $x, y$  represent two software operations and  $\bullet$  represents sequential composition operation. Their work is to check whether  $x \bullet y = y \bullet x$ . In contrast, our work studies whether two software design processes commute. The instantiation and integration of design patterns are two software design processes describing the activities conducted by software designers and developers. They are human design activities, instead of software system operations at run time. Let  $x, y$  be two design patterns and  $f, g$  be the instantiation and integration, respectively. Our work proves whether  $f(g(x, y)) = g(f(x), f(y))$ . Furthermore, their work only checks the commutativity on a case-by-case manner whereas we prove the general conditions that two design processes commute.

## 7. CONCLUSIONS

Design patterns have been widely adopted in software industry to reuse expert design experience. Use of design patterns generally involves the instantiation and integration of them. So far, it is unclear whether different orders of design process render different design results. The answer of this question is important to software developers since it can save their time and reduce errors in software development.

In this paper, we investigated the issue of the commutability of the instantiation and integration of design patterns. We provided detailed proofs of our results on structural and behavioral aspects of design patterns. Our results showed that the instantiation and integration processes are commutable under certain conditions. More specifically, we define single and multiple instantiations for both structural and behavioral aspects of design patterns. We also distinguish message and process integrations for behavioral contracts. Our proof results show that single instantiation and integration of the structure of design patterns are commutable if the corresponding integration participants are commutable under single instantiation. Multiple-instantiation and integration are commutable when the corresponding integration participants are commutable under multiple-instantiation, the number of integration participants after instantiation is the same as that of the instances of the integration participants, and all classes are mapped to different instances. In the behavioral aspect of design patterns, we have proved that instantiation and message integration are commutable. In contrast, the instantiation and process integration are commutable under two conditions: what the instance of a process becomes in the integration is the same as the instance of what the process becomes in the integration and the instantiation of the restriction set of the integration is the same as the restriction set of the integration of the pattern instantiations.

## REFERENCES

- [1] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- [2] Alvarez, J.M., Diaz, M., Llopis, L., Pimentel, E. and Troya, J.M. (2003) An object-oriented methodology for embedded real-time systems. *Comp. J.*, **46**, 123–145.
- [3] Koriem, S.M. (2006) Development, analysis and evaluation of performance models for mobile multi-agent networks. *Comp. J.*, **49**, 685–709.
- [4] Almendros-Jiménez, J.M. and Iribarne, L. (2009) UML modeling of user and database interaction. *Comp. J.*, **52**, 348–367.
- [5] Goumopoulos, C. and Kameas, A. (2009) Ambient ecologies in smart homes. *Comp. J.*, **52**, 922–937.
- [6] Yacoub, S.M. and Ammar, H.H. (2004) *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison-Wesley Professional.
- [7] Dong, J., Alencar, P.S.C. and Cowan, D.D. (2000) Ensuring Structure and Behavior Correctness in Design Composition. *Proc. 7th Annual IEEE Int. Conf. and Workshop on Engineering of Computer Based Systems (ECBS)*, Edinburgh, UK, pp. 279–287.
- [8] *Java.awt resource information*, September 2006. <http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html>
- [9] Dong, J., Peng, T. and Qiu, Z. (2007) Commutability of Design Pattern Instantiation and Integration. *Proc. First IEEE & IFIP Int. Symposium on Theoretical Aspects of Software Engineering (TASE)*, China, pp. 283–292.
- [10] Dong, J., Alencar, P.S.C. and Cowan, D.D. (2004) A behavioral analysis and verification approach to pattern-based design composition. *Softw. Syst. Model.*, **3**, 262–272.
- [11] Dong, J., Alencar, P.S.C. and Cowan, D.D. (2006) Automating the analysis of design component contracts. *Softw. Pract. Exp.*, **36**, 27–71.
- [12] Dong, J. and Peng, T. (2007) A Formal Framework for Modeling and Analysis of Pattern-Based Design. Technical Report #UTDCS-07-07, Computer Science Department, University of Texas at Dallas.
- [13] Steel, C., Nagappan, R. and Lai, R. (2005) *Core Security Patterns*. Prentice Hall.
- [14] Milner, R. (1989) *Communication and Concurrency*, International Series in Computer Science. Prentice Hall.
- [15] Stirling, C. (1991) *An Introduction to Modal and Temporal Logics for CCS*, Lecture Notes in Computer Science, Vol. 491. Springer, pp. 1–20.
- [16] De Francesco, N., Fantechi, A., Gnesi, S. and Inverardi, P. (2001) Finite approximations for model checking non-finite-state processes. *Comp. J.*, **44**, 109–123.
- [17] Dong, J., Peng, T. and Zhao, Y. (2007) Model Checking Security Pattern Compositions. *Proc. Seventh Int. Conf. on Quality Software (QSIC)*, Portland, Oregon, USA, IEEE CS Press, pp. 80–89.
- [18] Dong, J., Peng, T. and Zhao, Y. (2010) Automated verification of security pattern compositions. *Inf. Softw. Technol.*
- [19] Alencar, P.S.C., Cowan, D.D. and Lucena, C.J.P. (1996) A Formal Approach to Architectural Design Patterns. *Proc. Third Int. Symposium of Formal Methods Europe (FME)*, Barcelona, Spain, October, pp. 576–594.
- [20] Mikkonen, T. (1998) Formalizing Design Pattern. *Proc. 20th Int. Conf. on Software Engineering*, Kyoto, Japan, April, pp. 115–124.
- [21] Alencar, P.S.C., Cowan, D.D., Dong, J. and Lucena, C.J.P. (1999) A Pattern-Based Approach to Structural Design Composition, *Proc. 23rd Annual Int. Computer Software & Applications Conf.*, Phoenix, USA, October, pp. 160–165.
- [22] Chinnasamy, S., Raje, R.R. and Liu, Z. (1999) Specification of Design Patterns: An Analysis. *Proc. 7th Int. Conf. on Advanced Computing and Communications*, Roorkee, India, December, pp. 300–304.
- [23] Saeki, M. (2000) Behavioral specification of GoF design patterns with LOTOS. *Proc. Seventh Asia-Pacific Software Engineering Conf. (APSEC)*, Singapore, December, pp. 408–415.
- [24] Taibi, T. and Ngo, D.C.L. (2003) Formal specification of design pattern combination using BPSL. *Inf. Softw. Technol.*, **45**, 157–170.

- [25] Soundarajan, N. and Hallstrom, J.O. (2004) Responsibilities and Rewards: Specifying Design Patterns. *Proc. 26th Int. Conf. on Software Engineering*, Edinburgh, Scotland, May, pp. 666–675.
- [26] Meyer, B. (1992) Applying ‘design by contract’. *IEEE Comp.*, **25**, 40–51.
- [27] Pal, P. (1995) Law-Governed Support for Realizing Design Patterns. *Technology of Object-Oriented Languages and Systems*, Santa Barbara, USA, pp. 25–34.
- [28] Fekete, A., Lynch, N. and Weihl, W. (1990) Commutativity-based locking for nested transactions. *J. Comput. Syst. Sci.*, **41**, 65–156.
- [29] Weihl, W.E. (1988) Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, **37**, 1488–1505.
- [30] Wu, P. and Fekete, A. (2003) An Empirical Study of Commutativity in Application Code. *Proc. Int. Database Engineering and Applications Symposium*, Hong Kong.
- [31] Cairns, P. and Thimbleby, H. (2008) Affordance and symmetry in user interfaces. *Comp. J.*, **51**, 650–661.
- [32] Dennis, G., Seater, R., Rayside D. and Jackson, D. (2004) Automating Commutativity Analysis at the Design Level. *Int. Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, USA.

## APPENDIX

THEOREM A.1 (EXPANSION OF MESSAGE INTEGRATION).

Suppose we have pattern  $1, \dots, n$  with CCS expression as

$$CCS(BC^i) = \left( \prod_{p \in P^i} G(i, p) \right) [f_i] \setminus L_i,$$

where  $i$  refers to the pattern name.

The message integration defined in Definition 4.4 is  $CCS(BC) = \left( \prod_{i=1}^n CCS(BC^i) \right) [f] \setminus L$ .

Then we have the following important conclusion:

$$CCS(BC) = \left( \prod_{(i,p) \in P} G(i, p) \right) [F] \setminus L^*,$$

where

$$P = \bigcup_{i=1}^n \bigcup_{p \in P^i} (i, p), \quad F = f \circ f_i, \quad L^* = L \cup \left( f \left( \bigcup_{i=1}^n L_i \right) \right)$$

*Proof.* From the definition of behavioral contract, we know  $CCS(BC^i) = \left( \prod_{p \in P^i} G(i, p) \right) [f_i] \setminus L_i$ .

Then we have  $CCS(BC) = \left( \prod_{i=1}^n CCS(BC^i) \right) [f] \setminus L = \left( \prod_{i=1}^n \left( \prod_{p \in P^i} G(i, p) \right) [f_i] \setminus L_i \right) [f] \setminus L$ .

It’s not difficult to know that  $\prod_{i=1}^n \left( \left( \prod_{p \in P^i} G(i, p) \right) [f_i] \setminus L_i \right) = \left( \prod_{i=1}^n \left( \prod_{p \in P^i} G(i, p) \right) [f_i] \right) \setminus \bigcup_{i=1}^n L_i$ .

Therefore,  $\left( \prod_{i=1}^n \left( \prod_{p \in P^i} G(i, p) \right) [f_i] \setminus L_i \right) [f] \setminus L = \left( \left( \prod_{i=1}^n \left( \prod_{p \in P^i} G(i, p) \right) [f_i] \right) \setminus \bigcup_{i=1}^n L_i \right) [f] \setminus L$ .

Then, we can exchange the order of relabel function  $f$  and restriction set  $\bigcup_{i=1}^n L_i$ , and have

$$\begin{aligned} & \left( \left( \prod_{i=1}^n \left( \prod_{p \in P^i} G(i, p) \right) [f_i] \right) \setminus \bigcup_{i=1}^n L_i \right) [f] \setminus L \\ &= \left( \prod_{i=1}^n \left( \left( \prod_{p \in P^i} G(i, p) \right) [f_i][f] \right) \right) \setminus f \left( \bigcup_{i=1}^n L_i \right) \setminus L. \end{aligned}$$

Since the application of  $[f_i][f]$  is renamed first by using  $f_i$  then using  $f$ , it can be written as  $[f \circ f_i]$ . As a result we have

$$\begin{aligned} & \left( \prod_{i=1}^n \left( \left( \prod_{p \in P^i} G(i, p) \right) [f_i][f] \right) \right) \setminus f \left( \bigcup_{i=1}^n L_i \right) \setminus L \\ &= \left( \prod_{i=1}^n \left( \left( \prod_{p \in P^i} G(i, p) \right) [f \circ f_i] \right) \right) \setminus f \left( \bigcup_{i=1}^n L_i \right) \setminus L. \end{aligned}$$

Let the  $F$  be defined as, for any message  $m$ , if  $m$  is a message from contract  $i$ , then  $F(m) = f(f_i(m))$ . Hence we have

$$\begin{aligned} & \left( \prod_{i=1}^n \left( \left( \prod_{p \in P^i} G(i, p) \right) [f \circ f_i] \right) \right) \setminus f \left( \bigcup_{i=1}^n L_i \right) \setminus L \\ &= \left( \prod_{i=1}^n \prod_{p \in P^i} G(i, p) \right) [F] \setminus f \left( \bigcup_{i=1}^n L_i \right) \setminus L. \end{aligned}$$

Let  $P = \bigcup_{i=1}^n \bigcup_{p \in P^i} (i, p)$ , then

$$\begin{aligned} CCS(BC) &= \left( \prod_{i=1}^n \prod_{p \in P^i} G(i, p) \right) [F] \setminus f \left( \bigcup_{i=1}^n L_i \right) \setminus L \\ &= \left( \prod_{(i,p) \in P} G(i, p) \right) [F] \setminus \left( L \cup f \left( \bigcup_{i=1}^n L_i \right) \right). \end{aligned}$$

Hence, we reach

$$CCS(BC) = \left( \prod_{(i,p) \in P} G(i, p) \right) [F] \setminus L^*.$$

□