

# Verifying Behavioral Correctness of Design Pattern Implementation

Tu Peng, Jing Dong, Yajing Zhao

*Department of Computer Science*

*University of Texas at Dallas, Richardson, TX 75083, USA*

*{txp051000, jdong, yxz045100}@utdallas.edu*

## Abstract

*Design pattern describes a recurring problem and its common solution, which usually is in abstract form. The application of design pattern requires coding the generic solution. It is necessary to assure the coding process correctly implements not only the structure but also the desired behavior of the design pattern. This problem is called implementation correctness in this paper. By providing the definition of partial order between sequence diagrams, we formally describe the implementation correctness. We verify the implementation correctness with model checking by using process algebra to specify the source code and temporal logic to specify the behavior of the pattern.*

**KEYWORDS:** Design pattern, model checking, temporal logic, partial order, process algebra

## 1. Introduction

Design pattern [12] is a reusable software development strategy. While design pattern documents expert experience, the correctness of its implementation is critical to assure the quality of the software system. Each design pattern is usually described by its intent, motivation, structural and behavioral solutions, consequences, known uses, and etc. The structural and behavioral solutions are typically modeled by class and sequence diagrams, respectively. In this paper, we focus on analyzing the behavioral correctness of design pattern implementation by exploiting the partial order relationship between the sequence diagram of a general design pattern and that of its implementation. We identify this problem as *implementation correctness*: given a design pattern  $X$  and its implementation program  $P$ , is the program the correct implementation of  $X$ ? For example, consider the source code shown in [21] which is adopted from [19]. This source code is claimed to be an implementation of the Observer pattern [12]. However, how could we justify this? More generally, how do we know whether the implementation of a design pattern satisfies its behavior? To know whether the behavior of a pattern is implemented correctly is critical in assuring software reliability.

Formal methods have been widely used in verification because of two advantages. First, formal verification renders rigorous results. Second, formal verification can be facilitated by tool support, e.g. a model checker, thus less human efforts and human errors are involved. However, there are

three obstacles to formally verify design pattern implementation. First, design pattern is more like guidance rather than any concrete algorithm; hence it is not easy to formally describe what rules to be verified. Second, design pattern implementation may be different from the original description, in terms of class names, method names, class numbers, method numbers, and so on. This makes it difficult to apply a single algorithm to verify the correctness of different implementations. Third, regarding to the correctness of behavioral characteristics of patterns, software program which has a large number of states can cause significant runtime delay and even state explosion when using model checker.

Our approach to tackle these obstacles is based on the following ideas. First, since the behavior of design patterns can be normally modeled by sequence diagrams, we abstract the verification rules from sequence diagrams and use temporal logics, e.g., CTL [10] and its extensions, to specify them. Second, we introduce anonymous specification, so that the implementation with different class/method names can be checked against the original design pattern. Third, we use CCS [15] to specify the sequence diagram recovered from the program, instead of the program itself. This makes the resulted system less complex by involving fewer states.

There have been many researches on the formal specification and verification of design pattern. Some approaches [1][3][11][14][17] focused on the formal specification of design pattern. Other approaches [4][5][6][8] discussed the verification of design pattern applications, supported by model checker. Previous works mainly focus on verifying the properties of design patterns at the design level, e.g., the liveness and safety properties of design pattern [6], the correctness of design pattern composition [4], or the security design patterns [8]. Little effort on verifying design pattern implementation has made, which is very important to assure the reliability of software system [13].

In the next section, we introduce the partial order between sequence diagrams, from which we can formally define the implementation correctness problem. In Section 3, we discuss in detail how to verify the implementation against the design pattern. Related theorems and algorithms are also presented. In Section 4, we present a case study to demonstrate the algorithms for implementation correctness problem, and provide other application which can be reduced to the same problem. We conclude this paper in Section 5.

## 2. Partial Order of Sequence Diagrams

Sequence diagram reflects the order of the method invocations in an object, and the interaction between different objects in a software system. There are three advantages of using sequence diagram to capture the behavior of a design pattern or a program. First, the methods, the order of their occurrence and their interactions are rigorously described, which make it easy for formalizing. Second, there exist tools which are able to recover sequence diagrams from source code. This can release human from analyzing source code directly. Third, sequence diagram is an abstraction of the behavior of objects, which involves less variables and states; hence the system derived from a sequence diagram is more suitable for model checking [2]. Figure 1 displays the sequence diagram that models the behavior of the general Observer pattern. Figure 2 shows the sequence diagram of the source code in [21] which is an implementation of the Observer pattern. Sequence diagrams can be automatically generated from the source code by tools, e.g., Together [20].

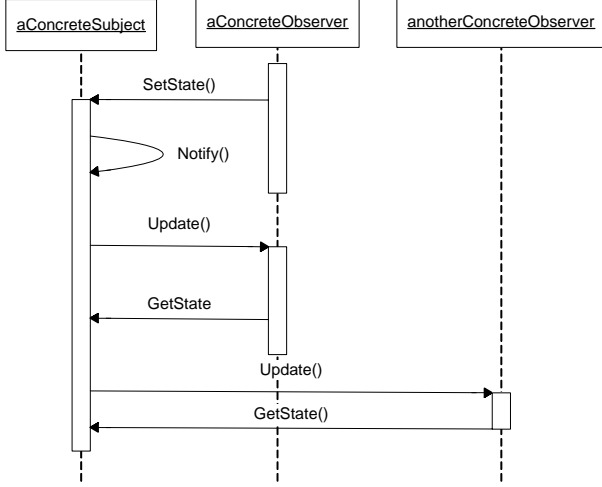


Figure 1 sequence diagram of observer pattern

Our goal can now be reduced to studying the relationship between the sequence diagram in Figure 1 and that in Figure 2. If the sequence diagram of the implementation satisfies that of the design pattern, it is reasonable to believe that the implementation is correct. This relationship between two sequence diagrams is defined in partial order in this paper.

There have been several researches on discovering design pattern from source code [7][9][16], so that the original design decisions of the source code can be recovered. The correctness of the recovery can actually be reduced to checking the partial order relationship between the sequence diagram of the recovered pattern and that of the source code. We will discuss how the partial order is formally defined in the rest of this section, how the partial order can be automatically checked in Section 3, and how the partial order can be applied to solve practical problems in Section 4.

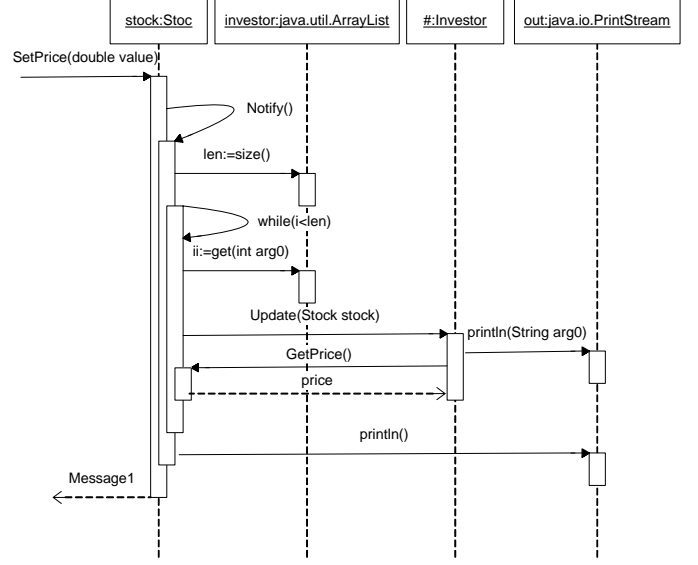


Figure 2 Sequence diagram of the implementation

In our approach, we found the order of the actions in a sequence diagram is the key factor to decide the relationship of two sequence diagrams. Hence we first define a convenient notation to denote the time order between actions.

**Definition 1** Suppose  $a$  and  $b$  are two actions. We use  $a < b$  to denote  $a$  occurs before  $b$ , and  $a > b$  to denote  $a$  occurs after  $b$ .

As the sequence diagrams may come from different sources, e.g., from a design model by a designer or recovered from a piece of source code, the actions usually are named differently although they may define the same sequence of actions. Hence we are interested in the order of two different sets of actions, rather than the action names, from two different sequence diagrams. We define the order of actions as follows.

**Definition 2** Suppose  $A$  is a set of actions and  $f$  is a one-one function with domain and range over  $A$ . Suppose  $t$  is a set of actions, which are mapped into another set of actions  $f(t)$  by function  $f$ . We say actions in  $t$  and actions in  $f(t)$  have the same order, denoted by  $t \approx f(t)$  when the following formula applies,

$$\forall t_1, t_2 \in t:$$

$$(t_1 > t_2 \rightarrow f(t_1) > f(t_2)) \wedge (t_1 < t_2 \rightarrow f(t_1) < f(t_2)).$$

This definition specifies that the renaming function  $f$  reserves the order of the actions in  $t$ .

**Lemma 1** If  $t \approx f(t)$  and  $s \subseteq t$ , then  $s \approx f(s)$ .

The proof is obvious.

**Definition 3** Suppose  $D_1$  and  $D_2$  are two sequence diagrams and  $t$  is a set of actions. We say  $D_1$  satisfies  $D_2$  with

respect to  $t$ , expressed by the formula  $D_1 \succ_t D_2$ , if and only if the following conditions are observed:

- 1)  $D_2$  contains all the actions of  $t$ , and  $D_1$  contains a set of actions  $t_1$ , which can be mapped one by one onto  $t$ . That is, there exists a one-one function  $f$  from  $t$  to  $t_1$ .
- 2)  $t \approx f(t)$ .

This definition states that one sequence diagram satisfies another sequence diagram.

**Theorem 1**  $\succ_t$  is a partial order.

*Proof:* We need to prove the transitive, asymmetric, and reflexive properties of  $\succ_t$ .

*Transitive:* suppose there are three sequence diagrams,  $D_0$ ,  $D_1$  and  $D_2$ , with  $D_1 \succ_t D_2$  and  $D_0 \succ_{t_1} D_1$ . Since  $D_1 \succ_t D_2$ , there is a one-one function  $f$  that maps the actions from  $t$  to  $t_1$ . Since  $D_0 \succ_{t_1} D_1$ , there is a one-one function  $f_1$  that maps the actions from  $t_1$  to  $t_2$ . Then it is clear that  $f_1 \circ f$  is the one-one function from  $t$  to  $t_2$ . On the other hand, for any two actions  $a$  and  $b$  from  $t$ , if  $a$  occurs before  $b$  in  $D_2$ , then  $f(a)$  occurs before  $f(b)$  in  $D_1$  because  $D_1 \succ_t D_2$ . Then  $f_1(f(a))$  occurs before  $f_1(f(b))$  because  $D_0 \succ_{t_1} D_1$ . Hence  $D_0 \succ_t D_2$ .

*Asymmetric:* it is clear that  $D_1 \succ_t D_2$  and  $D_2 \succ_t D_1$  cannot be true at the same time. Otherwise,  $t$  may contains actions not in  $D_1$ .

*Reflexive:* it is obvious that  $D_1 \succ_t D_1$ .

Hence we complete the proof that  $\succ_t$  is a partial order.

Informally, partial order describes to what extend two sequence diagrams are similar to each other. If a partial order exists between two sequence diagrams, one could say the behavior of one sequence diagram is captured or included in another sequence diagram.

**Definition 4** Suppose  $t_1, t_2, \dots, t_n$  are  $n$  sets of actions, we write  $D_1 \succ_{t_1, \dots, t_n} D_2$  to mean that  $D_1 \succ_{t_1} D_2, \dots, D_1 \succ_{t_n} D_2$  hold.

### 3. Verify Design Pattern Implementation

In this section, we will discuss how to verify if the implementation of a given design pattern is correct. We start by formally defining the implementation correctness.

**Definition 5** Given a design pattern  $X$ , whose sequence diagram is  $D_X$ . Given a program  $P$ , whose sequence diagram is  $D_P$ . Given  $n$  sets of actions  $t_1, t_2, \dots, t_n$  which occur in  $D_X$ . Then we say  $P$  is a correct implementation of  $X$  with respect to  $t_1, t_2, \dots, t_n$ , if and only if  $D_P \succ_{t_1, \dots, t_n} D_X$ .

This definition formally specifies the implementation correctness problem. That is, by comparing the order of the actions in  $t_1, t_2, \dots, t_n$  (from the design pattern) and their correspondent actions (from the implementation), one could know whether the design pattern is correctly implemented. We will then propose an approach to verify whether  $D_P \succ_{t_1, \dots, t_n} D_X$ .

Given design pattern  $X$  and program  $P$ . The following is the outline of our approach.

- 1) For a given design pattern  $X$ ,  $D_X$  is known. That is, the behaviour of a given pattern is known and is used as the standard to be verified against.
- 2) For  $X$ 's implementation  $P$ ,  $D_P$  can be obtained automatically by using a software tool which can recover the sequence diagram from a program.
- 3) For all actions in  $t_1, t_2, \dots, t_n$ , we use temporal logic to specify their existence and the order of their occurrence. This specification consists of a set of temporal logic properties, which is denoted by  $PROP_X$ .
- 4) Formally specify  $D_P$ , using CCS that is a process calculus, and obtain the formal expression of  $D_P$  denoted by  $CCS_P$ , which is a set of processes.
- 5) Verify if  $CCS_P \models PROP_X$  is true, which will be defined in Definition 6.

We use the CCS (calculus for communicating system) [15] as the specification language. CCS is a process algebra which describes labelled transition system where several subsystems communicate with each other. The syntax of CCS is shown as follows

$$A ::= a.A \mid a'.A \mid A \mid A \mid A + A \mid A \setminus L \mid A[f].$$

where  $A$  is a CCS process.  $a$  is an incoming message and  $a'$  is an outgoing message.  $a.A$  means after accepting message  $a$ , process  $A$  happens, where “.” is sequential operator;  $A/A$  means process  $A$  and  $A$  are concurrent, where “|” is parallel composition operator;  $A + A$  means either one of the two processes can happen, where “+” is summarization operator;  $A \setminus L$  means all the messages of  $A$  which are included in set  $L$  are restricted as internal message, and “\” is restriction operator;  $A[f]$  means that the messages of  $A$  are renamed by the rule provided by  $f$ , and  $[\ ]$  is referred as relabel operator.

**Definition 6**  $CCS_P \models PROP_X$  holds when every property in  $PROP_X$  is satisfied by  $CCS_P$ . More specifically, for every property  $p_i$  in  $PROP_X$ , there exists a process in  $CCS_P$  which satisfies  $p_i$ .

The following definition provides a way to compute  $PROP_X$ . We will specify the order of the actions in set  $t_1, t_2, \dots, t_n$ , in terms of temporal logic. These temporal logic specifications are usually called properties.  $PROP_X$  is actually a set of these properties.

**Definition 7**  $PROP_X$  is a set of temporal logic properties, which specify the existence and order of the actions in  $t_i$ . Namely,  $PROP_X = \{1 \leq i \leq n \mid p_i\}$ . For each action  $a$  in  $t_i$ , suppose  $f$  is the function defined on  $t_i$ . Suppose  $m \subseteq t_i$  such that  $f(m) = m$ :

- 1) Let  $p_i$  specify the existence of all actions in  $t_i$ .
- 2) Let  $p_i$  specify the order of occurrence of all actions in  $m$  by referring their names.
- 3) Let  $p_i$  specify the order of occurrence of all actions in  $t_i - m$  without referring their names.

There are a few points about this definition that need to be mentioned. First, it is not fully automatable and human efforts are needed in every step. Second, the actions in  $t_i - m$  have different names in the implementation from those in the design pattern, which need to be specified anonymously, so that the name difference between design pattern and its implementation can be tolerated in model checking.

In the rest of this section, we present two algorithms for our approach. Algorithm 1 provides the steps to compute  $CCS_p$ , which is an abstract model of program  $P$ .  $CCS_p$  is a set of processes specifying the actions of objects in  $D_p$ . Algorithm 2 provides steps to check  $CCS_p \models PROP_X$ ,

#### Algorithm 1

For each class  $C$  appearing in  $D_p$ , the specification of  $C$ 's actions,  $spec_C$ , is obtained by the following steps:

- 1) For each activation bar in a sequence diagram, specify its actions with sequential order. For all the outgoing actions, specify them with a prime symbol before their names. For all the other messages, just specify them with their names.
- 2) Connect the specification of each activation bar with summarization operation.
- 3) Add a recursion and we complete the specification of the actions performed by  $C$ .

$$CCS_p = \{ C \text{ from } D_p \mid spec_C \}$$

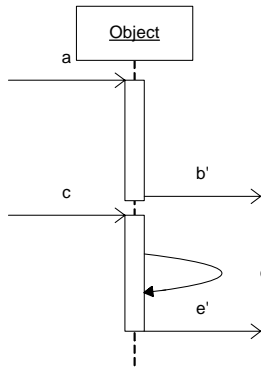


Figure 3 A Simple Sequence Diagram

Let us use an example to illustrate how Algorithm 1 is used to specify a sequence diagram in CCS. Figure 3 shows a simple sequence diagram. Its CCS process is  $\{a.b'+c.d.e'\}$ , where  $a.b'$  is obtained through the first activation bar and  $c.d.e'$  is obtained from the second one.

#### Algorithm 2

For every property  $r$  in  $PROP_t$ , if there exist a process  $s$  in  $CCS_p$ , such that  $s$  satisfies  $r$ , then  $r$  is checked by model checker CWB-NC [18].

**Theorem 2** Specification of  $D_p$  is automatable.

*Proof:* This comes naturally from the existence of the Algorithm 1 to specify  $D_p$ .

**Theorem 3**  $CCS_p \models PROP_X$  holds in Definition 7 if and only if  $D_p \succ_{t_1, \dots, t_n} D_X$  holds.

*Proof:* On one hand, if  $D_p \succ_{t_1, \dots, t_n} D_X$  holds, then  $D_p \succ_{t_i} D_X$  holds for  $1 \leq i \leq n$ . That is,  $t_i \approx f(t_i)$ . Let  $p_i \in PROP_X$  be the temporal logic property specifying  $t_i$ . Since  $t_i \approx f(t_i)$ , which means the actions in  $t_i$  and their correspondence in  $f(t_i)$  happen in the same order. It follows that the process containing actions in  $f(t_i)$  must satisfy  $p_i$ . Hence  $CCS_p \models PROP_X$  holds.

On the other hand, if  $CCS_p \models PROP_X$  holds, then all the properties in  $PROP_X$  are satisfied by  $P$ . Hence the actions specified by the properties must happen in the same order as their correspondence in  $P$ . This would mean that  $D_p \succ_{t_i} D_X$  holds for  $1 \leq i \leq n$ . Thus we complete the proof.

This theorem demonstrates that it is reasonable for us to use model checking to check  $CCS_p \models PROP_X$ , so as to determine whether  $D_p \succ_{t_1, \dots, t_n} D_X$  holds.

## 4. Case Study

In this section, we apply our approach presented in Section 3 to study the case in Section 2, that is, whether the source code is the correct implementation of the Observer pattern. As described in [12], the behaviour of the Observer pattern can be specified by the sequence diagram shown in Figure 1. The basic property of the Observer pattern can be summarized: 1) whenever the setstate in Subject is invoked, 2) the notify action must be performed, 3) the update in every Observer must be invoked, followed by the getstate. Hence the problem is to verify whether  $D_p \succ_t D_{observer}$ , with  $t = \{\text{notify, update, setstate, getstate}\}$ , where we assume the action names are case insensitive. In particular, we specify the following properties of the Observer pattern.

- The first property states that there must be an action that happens before “notify”. This action can be “set-

state” or named differently. Since “setstate” can have different name in real implementation, this consideration is essential.

```
prop subject_order1=
(E F{setstate}-> (E F{notify}))\ / (E F{-
setstate}-> (E F{notify}))
```

- The second property states that actions “notify” and “update” must exist in the Subject. Symbol “/” is applied before update to denote this message is outgoing.

```
prop subject_exist=
E F{notify}\ / E F{'update'}
```

- The third property states that action “update” must happen after the “notify” action.

```
prop subject_order2=
E F{notify}-> (E F{'update'})
```

- The fourth property states that there must be an action happens after the “update” action. Usually this is “getstate”, but can have different name.

```
prop subject_order3=
(E F{'update'}-> (E F{getstate}))\ / (E
F{'update'}-> (E F{-getstate}))
```

- The final property of process subject is the conjunction of the above properties.

```
prop subject=
subject_order1 /\ subject_order2 /\
subject_order3 /\ subject_exist
```

All these properties are specified in GCTL, which is an extension of traditional temporal logic CTL\* [10] that is tailored for reasoning about systems whose transitions are labelled by actions. The syntax of GCTL is

$$S ::= p \mid \neg p \mid S \wedge S \mid S \vee S \mid A p \mid E p \mid G p \mid F p$$

$$P ::= \theta \mid \neg \theta \mid S \mid P \wedge P \mid P \vee P \mid X P \mid P \cup P \mid P R P$$

where  $S$  is state formula,  $P$  is path formula.  $p$  is atomic proposition, and  $\theta$  is atomic action proposition.  $A$  is a universal quantifier which means the formula after  $A$  is true in every state starts from the current state.  $E$  is an existential quantifier which means that there exists a state following the current state, where the formula after  $E$  is true.  $G$  is a path universal quantifier which means the formula is true along all the states in the path from the current state.  $F$  is a path existential quantifier which means the formula is true in some state in the path from the current state.  $G, F$  are always used together with  $A$  and  $E$ . More detailed specification of the semantics of GCTL is in [18].

A practical implementation of the Observer Pattern is shown in [21], whose sequence diagram is recovered from its source code in Figure 2. Note that this discovery is automatically done by Together [20].

Then, we need to specify the behaviour of the Observer pattern formally in CCS as follows

---

```
proc observer=
(stock|investors|investor|printer)\{
setprice,get,update,getprice,println}
```

```
proc stock=setprice.notify.'size.'get
.update.stock1
proc stock1=getprice.'println.stock
```

```
proc investors=
size.investors1+get.investor1
proc investors1=investors
```

```
proc investor=update.'println.investor1
proc investor1='getprice.investor
```

```
proc printer=println.printer1
proc printer1=println.printer
```

---

Once we obtain the CCS specification of the source code, we could simply verify it against the properties previously defined, with a model checker [18].

We save the properties of the Observer pattern,  $PROP_i$ , in the file “observer.gctl”. We save the system specification of the sequence diagram of the program in the file “observer.ccs”. Then we can model-check it with the following commands and results:

---

```
cwb-nc> chk -L gctl stock subject
Generating ABTA from GCTL* formula...done
Initial ABTA has 56 states.
Simplifying ABTA:
Minimizing sets of accepting states...done
Performing constant propagation...done
Joining operations...done
Shrinking automaton...done
Computing bisimulation...
Done computing bisimulation.
Simplification completed.
Simplified ABTA has 30 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 29 times.
Stored 0 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):
(0.047,0.000,0.000,0.047)
```

---

Comparing to a conventional approach of model checking, that is, to specify the system from source code, our approach is more efficient and fast, because the system model is concisely built and thus involves far less states than the model built from source code directly. Hence our approach reduces the possibility of state explosion and increases model checking speed. As shown in the runtime scripts above, our system model contains as few as 56 states in total and is simplified to only 30 states, due to the specification from sequence diagram instead of the source code. The running time is only 0.047 seconds.

## 5. Conclusions and Future Work

Design patterns have been widely adopted in software industry to reuse expert design experience. Use of design patterns generally involves implementing them in different forms. So far, there has been lack of methods to ensure the correctness of the design pattern implementation.

In this paper, we provide a method to formally verify design pattern implementation. We formally define the implementation correctness problem in terms of a partial order of two sequence diagrams. We obtain the system specification ( $CCS_p$ ) by recovering the sequence diagram from the source code, and specifying it with CCS. We generate the properties ( $PROP_X$ ) of the design pattern by using temporal logic to specify its behaviour, which is usually modelled by a sequence diagram ( $D_X$ ). We verify  $CCS_p$  against  $PROP_X$  with CWB-NC model checker. Our approach provides a formal standard and concrete method to solve the implementation correctness assurance problem. Moreover, since  $CCS_p$  is built concisely, it is easy and fast to use model checking in our approach.

Future work includes verifying the structural correctness of design pattern implementation and exploring other methods of verification. One possible way of verifying the structural correctness is to exploit the current work on design pattern recovery, which is able to extract structural information such as classes, methods, and their associations from source codes and compare such information with general design pattern to determine whether the system correctly implement the pattern from structural point of view. One alternative way of verifying the behavioral correctness is to specify design pattern and its implementation as two systems respectively, and check the behavioral relationship of those systems. Since specifying design pattern directly is a better way to capture design pattern behavior than specifying the summarized properties of the pattern, we could expect more accurate result in this approach.

There are other issues that can be addressed by the implementation correctness. For example, to recover design pattern from source code is an important task in reverse engineering [7][9][16]. However, there lacks research on how to formally specify and verify the correctness of design pattern recovery results. This problem may be addressed as the implementation correctness problem because program  $P$  correctly implements design pattern  $X$ , if and only if design pattern  $X$  is correctly recovered from program  $P$ .

## References

- [1] Paulo Alencar, Donald Cowan, and Carlos Lucena. A Formal Approach to Architectural Design Patterns. *Proceedings of the Third International Symposium of Formal Methods Europe (FME)*, pages 576–594, 1996.
- [2] E.M Clarke, E.A. Emerson, A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2), April 1986.
- [3] S. Chinnasamy, R. R. Rajee, and Z. Liu. Specification of design patterns: An analysis. *Proceedings of the 7th International Conference on Advanced Computing and Communications*, pages 300–304, 1999.
- [4] Jing Dong, Paulo Alencar, and Donald Cowan, Ensuring Structure and Behavior Correctness in Design Composition, *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pp279-287, Edinburgh UK, 2000.
- [5] Jing Dong, Paulo Alencar, and Donald Cowan. A Behavioral Analysis and Verification Approach to Pattern-Based Design Composition. *International Journal of Software and Systems Modeling, Springer Verlag*, 3(4):262–272, December 2004.
- [6] Jing Dong, Paulo Alencar, and Donald Cowan. Automating the Analysis of Design Component Contracts, *Software - Practice and Experience*, Wiley, 36(1), pp. 27-71, Jan. 2006.
- [7] Jing Dong, Dushyant S. Lad and Yajing Zhao, DP-Miner: Design Pattern Discovery Using Matrix, the Proceedings of the Fourteenth Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS), pages 371-380, Arizona, USA, March 2007.
- [8] Jing Dong, Tu Peng, Yajing Zhao, Model Checking Security Pattern Compositions, the Proceedings of the Seventh International Conference on Quality Software (QSIC), pages 80-89, USA, October 2007, IEEE CS Press.
- [9] Jing Dong, Yajing Zhao, and Tu Peng, Architecture and Design Pattern Discovery Techniques – A Review, Proceedings of International Conference on Software Engineering Research and Practice (SERP), pages 621-627, USA, June 2007.
- [10] E.A. Emerson and J.Y. Halpern. ‘Sometime’ and ‘not never’ revisited: On branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151-178, January 1986.
- [11] A.H. Eden and Y. Hirshfeld. Principles in formal specification of object-oriented architectures. *Proceedings of the 11th CASCON, Toronto, Canada*, November 2001.
- [12] Gamma E., Helm R., Johnson R. and Vlissides J. (1995). *Design Patterns-Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley.
- [13] Peter C. Mehltitz, John Penix. Design for verification using design pattern to build reliable system. *Proceeding of 6<sup>th</sup> workshop on component-based software engineering, 2003*
- [14] Tommi Mikkonen. Formalizing Design Pattern. *Proceedings of the 20th International Conference on Software Engineering*, pages 115–124, 1998.
- [15] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [16] Nija Shi and Ron Olsson. Reverse Engineering of Design Patterns from Java Source Code. *The International Conference on Automated Software Engineering*, Japan, Sept. 2006.
- [17] Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and Rewards: Specifying Design Patterns. *Proceedings of the 26th International Conference on Software Engineering*, pages 666–675, May 2004.
- [18] CWB-NC User's Manual. <http://www.cs.sunysb.edu/~cwb/>
- [19] <http://www.dofactory.com/Patterns/Patterns.aspx>
- [20] <http://www.borland.com/us/products/together/index.html>
- [21] <http://www.utdallas.edu/~jdong/papers/Observer.java>