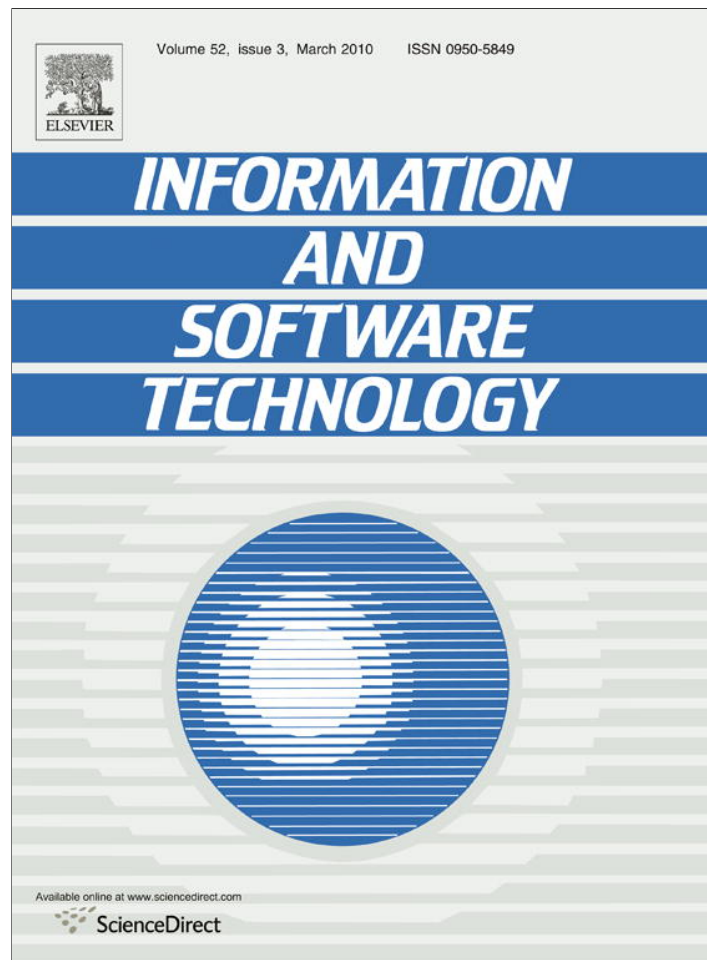


Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

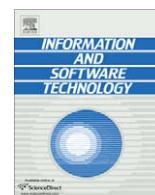
In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

<http://www.elsevier.com/copyright>



Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automated verification of security pattern compositions

Jing Dong*, Tu Peng, Yajing Zhao

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

ARTICLE INFO

Article history:

Received 11 July 2008

Received in revised form 3 October 2009

Accepted 5 October 2009

Available online 14 October 2009

Keywords:

Design pattern

Security

Logics

Process algebra

Model checking

ABSTRACT

Software security becomes a critically important issue for software development when more and more malicious attacks explore the security holes in software systems. To avoid security problems, a large software system design may reuse good security solutions by applying security patterns. Security patterns document expert solutions to common security problems and capture best practices on secure software design and development. Although each security pattern describes a good design guideline, the compositions of these security patterns may be inconsistent and encounter problems and flaws. Therefore, the compositions of security patterns may be even insecure. In this paper, we present an approach to automated verification of the compositions of security patterns by model checking. We formally define the behavioral aspect of security patterns in CCS through their sequence diagrams. We also prove the faithfulness of the transformation from a sequence diagram to its CCS representation. In this way, the properties of the security patterns can be checked by a model checker when they are composed. Composition errors and problems can be discovered early in the design stage. We also use two case studies to illustrate our approach and show its capability to detect composition errors.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

With increasing security attacks, security becomes a critical requirement for successful software system development. Studies [37] have shown that design flaws and errors are commonly the main source of security holes that are explored by attackers. Therefore, secure software architecture and design are at the heart of software security. Security becomes one of the most important factors that affect software quality. Security patterns [33,35] document good practices to solve security problems arising frequently in software development and encourage reusing expert solutions. A security pattern is a recipe of solving a particular security problem. It is a design pattern [19] that generally describes a group of participants as well as their relationships and collaborations, which achieve some security goals. Each participant in the group is defined generically in terms of the role it plays in the security pattern. The benefits of security patterns include the reuse of security design solutions instead of the reuse of just a piece of code, documentation of expert design experience, recording of security design tradeoffs, capturing of security decisions, and improvement of communication.

Multiple security patterns can be used in large software systems to solve many security problems. Combining security patterns may help to reuse expert solutions on different security

problems in the same system. While each security pattern describes expert experience on solving a particular security problem, the composition of these security patterns is not always a good solution. There can be inconsistencies among security patterns such that some critical security properties may no longer hold. The inconsistencies between security patterns may cause problems in the design. Discovering these problems and errors early in the design stage is important because such design errors are very difficult to find and correct when they are transformed to implementation errors. Analysis techniques that help to find such design errors are crucial to the quality of the software systems.

There are several automated verification techniques, such as model checking and theorem proving. Model checking has been initially applied in the hardware community to verify safety and liveness properties [3,5,16]. It has also been used in the software community, e.g., in the verifications of web service composition [20,21,8], in distributed cache coherence analysis [38], in hypermedia applications [11], and in security property analysis [23]. In this paper, we use model checking techniques to analyze the consistency of security pattern compositions. More specifically, we formally specify the behavioral aspect of the security patterns in the Calculus of Communicating Systems (CCS) [28], as well as the properties of each security pattern. We provide a general rule for specifying security pattern behavior modeled by a sequence diagram in CCS. We define the synchronous message, asynchronous message, and alternative flows of a UML sequence diagram and transform them into CCS specifications. We also prove the faithfulness of the CCS specification with respect to the sequence

* Corresponding author.

E-mail addresses: jdong@utdallas.edu (J. Dong), txp051000@utdallas.edu (T. Peng), yxz045100@utdallas.edu (Y. Zhao).

diagrams. A model checker is used to perform the analysis and check whether the characteristics of each security pattern still hold after they are composed. Our analysis results show that our approach is able to find the design errors that may lead to security holes and flaws.

The remainder of this paper is organized as follows: the next section describes our analysis techniques on security pattern compositions. Section 3 presents two case studies to illustrate our approach and show the discovery of several subtle security problems in the design. The last two sections cover related work and conclusions.

2. Our security analysis techniques

Many security patterns have been identified to solve the recurring security problems, such as authentication, authorization, and confidentiality during communication. One of the common security goals is secure communication between two parties. Unsecured communications are often exposed to eavesdropping, spoofing, sniffer, and replay attacks. The replay attacks copy the legitimate transactions and resend them. The sniffer attacks just capture sensitive information for later use. Many of these attacks can be categorized as man-in-the-middle attacks which cannot only harm the unsecured network but also VPN where data is exposed at the end points. This exposed data is still subjected to disclosure, modification, or duplication. Some of these attacks are easy to carry out, even for novices. As a consequence, these attacks may result in huge losses for businesses that need to communicate sensitive data.

In this section, we first describe model checking techniques. We then introduce our approach on modeling the behavior of a security pattern, which may then be checked by a model checker. We describe a general way to specify security pattern behavior in CCS and prove the faithfulness of the transformation from the sequence diagrams of a security pattern to our specification.

2.1. Model checking

In order to analyze the compositions of security design patterns, we apply model checking techniques. Model checking is a method of verifying algorithmically a formula against a logic model [5]. This verification technique can be automated by model checking tool (model checker). In our case, we assume a logic model representing the security patterns and their compositions, as well as a logic formula representing a property of these patterns. In order to determine whether the pattern-based system satisfies the properties, it is checked whether the formula holds in the logic model of security pattern compositions.

There are several model checking tools, such as SMV [25], SPIN [22], XMC [30], and CWB-NC [40]. In this paper, we will concentrate on the CWB-NC model checker since we can use it to specify security patterns in process algebra and analyze their composition by model checking. CWB-NC is a software tool which is capable of not only model checking but also behavioral equivalence verification. As a model checker, CWB-NC requires the user to specify the systems in CCS and the temporal properties with GCTL [29,40] (extension of computational tree logic, CTL [17]). It will then check whether the system satisfies the properties. In behavioral equivalence verification mode, CWB-NC requires the user to specify two systems in CCS. It will then check whether the two systems are equivalent under certain constraints. In this paper, we will use CWB-NC as a model checker. The syntax of CCS is as follows:

spec: *binding_list*
binding: “*proc*” *id* “=” *agent* | “*set*” *id* “=” *id_set*

agent: “*nil*” | *id* | *act* “.” *agent* | *agent* “+” *agent* | *agent* “|” *agent*
 | *agent* “\” *restriction* | (“*agent* “) ”
act: *id* | [“ ”] *id*
restriction: *id_set*

where the system specification consists of a list of bindings. Each binding is a formula, which either defines a process by using keyword “*proc*” or defines a set by using keyword “*set*”. 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 used together with parallel composition, to denote that the messages being restricted are internal messages. Let us consider an example specification of the behavior of the Observer pattern [19] in CCS as follows:

```
* observer *
proc OBSERVER=OSUBJECT|OBSERVER\{osetstate,ouupdate}
proc OSUBJECT=osetstate.ochangestate.'notify.
OSUBJECT1
proc OSUBJECT1=notify.'ouupdate.OSUBJECT
proc OOBERVER=ochange.'osetstate.OOBSERVER+ouupdate.
ogetstate.OOBSERVER1
proc OOBERVER1=OOBSERVER
```

The system specification of the Observer pattern consists of two processes, OSUBJECT and OOBERVER. The interaction of these two processes is that the OOBERVER process sends out osetstate message to the OSUBJECT which changes its state and sends out ouupdate to all OOBERVERs. The OOBERVER processes perform action ogetstate to update their states and thus keep it consistent with the OSUBJECT.

Temporal properties are expressed in GCTL, which is an extension of CTL. The syntax of GCTL is

$$S ::= p \mid \neg p \mid S \wedge S \mid S \vee S \mid Ap \mid Ep \mid Gp \mid Fp$$

$$P ::= \theta \mid \neg \theta \mid S \mid P \wedge P \mid P \vee P \mid XP \mid P \cup P \mid PRP$$

where S is state formula, P is path formula. p is atomic proposition, and θ is atomic action proposition. A is a universal quantifier which means that the formula is true in all paths starting from the current state. E is an existential quantifier which means that there exists a path following the current state, such that the formula is true. G is a path universal quantifier which means that the formula is true for all the states along 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. X is a path quantifier which means the formula is true in the next state in the path from the current state. G , F and X are always used together with A and E . We illustrate the use of quantifiers in Fig. 1.

2.2. Overview of our approach

Fig. 2 illustrates the main characteristics of our approach to analyzing security pattern compositions. Initially, each security pattern is formally specified using CCS. The security pattern specifications are generic in the sense that they capture good design practice in a domain-independent way. These declarative representations, which constitute the models of the security patterns, are then instantiated into concrete domain-specific representations. In this way, security design practice can be reused. The instances of security patterns are integrated to form a model Σ of the composition of the security patterns, which is then submitted

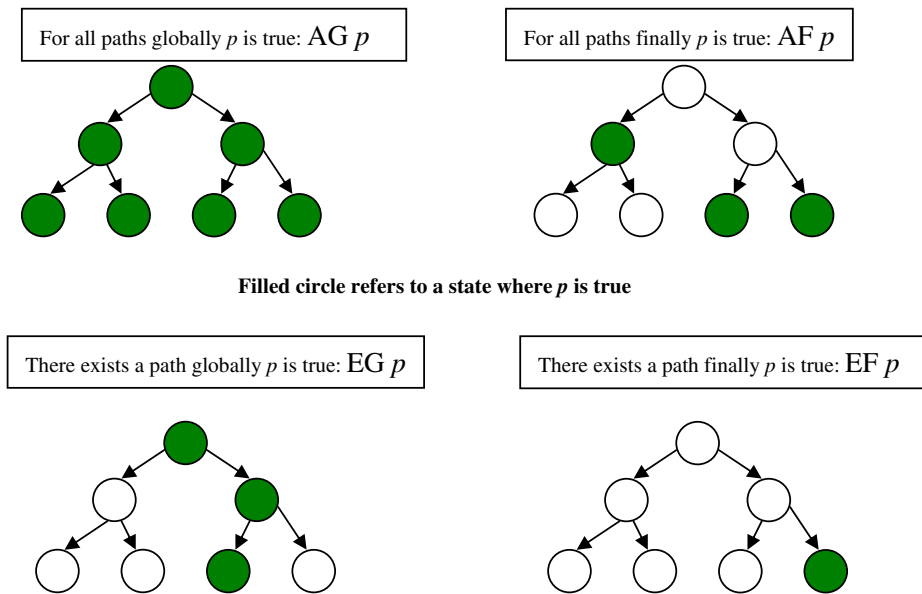


Fig. 1. CTL examples.

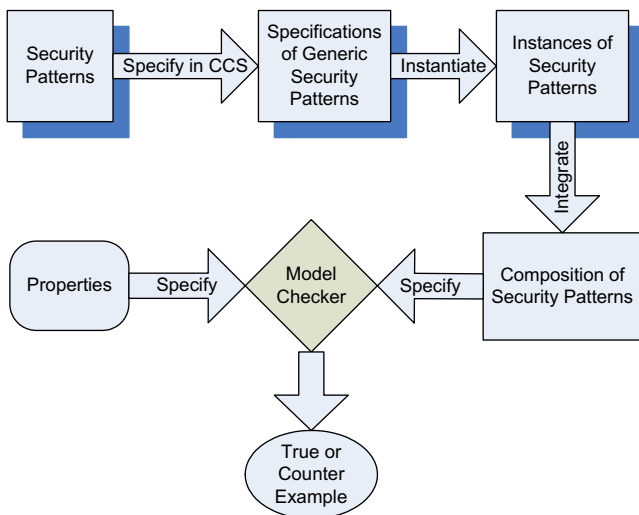


Fig. 2. Overview of our approach.

to a model checker. We use the model checker to check Σ against the property specification Φ . The model checker outputs either true, if Σ satisfies Φ , or a counterexample, if it does not.

More specifically, we provide the following guidance to formalizing security patterns in CCS. Each object in the security pattern is modeled as a CCS process that inputs some messages, performs actions, and then outputs some messages. Thus, each object is normally specified as a CCS process in the following way:

$$P(p) ::= \text{in}().\text{action}().\text{out}()$$

where $\text{in}()$ specifies that the process p receives some message from another process. Since p corresponds to an object, it means some operation of the corresponding object has been invoked by some other object. A process p may have multiple $\text{in}()$ which specifies that some operations of the object corresponding to p have been invoked by other objects. The $\text{action}()$ specifies that the process performs some action after receiving the message. The action typically changes the internal states of the corresponding process. The process can perform multiple actions as well. Model checking is generally applied to these state changes to verify whether a sequence of

state changes is desired. As a consequence, a process may send out some message after performing the actions. The $\text{out}()$ specifies that the corresponding object of P may invoke the operations of some other objects. Similarly, each process may send out multiple messages.

Since the behavior of each security pattern is typically modeled by sequence diagrams, we formalize each object in the sequence diagram as a CCS process. In the rest of this section, we present the steps to specify sequence diagram in CCS and prove theorems to show the faithfulness of CCS specifications. In other words, we show that CCS specifications maintain the information carried by sequence diagrams.

In order to specify a sequence diagram in CCS, we first define the formal syntax for sequence diagram in Definition 1. Based on the formal syntax of sequence diagram, we provide Rule 1 to introduce the steps to specify sequence diagram by CCS formula. Then we introduce auxiliary definitions and present theorems by which we prove the faithfulness of CCS specification. At last, we summarize the steps to specify sequence diagram by CCS in Theorem 4.

The following definition formalizes a sequence diagram of UML 2.0 with alternative flows.

Definition 1. The formal semantics of a sequence diagram is represented by a tuple $SD = \langle PT, MS, AB, FG \rangle$, where PT is a set of participating objects in the sequence diagram. A participant, $p \in PT$, is an object. MS is a set of messages in the sequence diagram. A message, $\langle c, r, s, t \rangle \in MS$, is a four-dimension vector, where $c, r \in PT$, s is the message signature and t is the message type in the sequence diagram. The message type can be either synchronous or asynchronous. AB is the set of activation bars in the sequence diagram. FG is the set of sequence fragments in the sequence diagram. A fragment $f \in FG$ can be the following value:

- $\langle o, \text{if}, \text{condition}, m, n \rangle$: a sequence of messages, $\{m, \dots, n\}$, are executed by object o , if the *condition* is satisfied.

The following definitions provide some important notations which are used in the theorems in this paper.

Definition 2. Suppose P is a CCS formula and a is an action. The fact that a appears in P is denoted by $a \in P$.

For example, suppose b and c are other two actions, then $P = a \cdot b \cdot c$, $P = a + b + c$, and $P = (a|b|c)\setminus L$ all imply $a \in P$.

Definition 3. Suppose P is a CCS formula and $a, b \in P$. $a <_t b$ denotes that action a happens before action b , that is, there is a sequential action path from a to b :

$$\exists c_1, \dots, c_n \in P, \text{ s.t. } a \cdot c_1 \dots c_n b \in P$$

The following rule introduces steps on how to specify sequence diagram in CCS.

Rule 1. Let $SD(X) = \langle PT, MS, AB, FG \rangle$ be the sequence diagram of security pattern X . The rule to derive $CCS(X)$ from $SD(X)$ is defined as follows:

1. For each object $o \in PT$, o is specified as $P(o)$ in CCS ($P(o) \in CCS(X)$). That is, each object o in sequence diagram is specified by a CCS formula $P(o)$.
2. For each $m = \langle c, r, s, t \rangle \in MS$, $c = r$, m is specified as $action(m) \cdot action(s)$ in $P(c)$ ($action(m) \cdot action(s) \in P(c)$). That is, for an internal message m (message caller and message receiver are both object c), formula $P(c)$ contains $action(m)$ and $action(s)$. Note that $action(s)$ is used to specify the signature of message m . When no information of signature is required in analysis, we can omit $action(s)$.
3. For each $m = \langle c, r, s, t \rangle \in MS$, $c \neq r$, m is specified as $out(m) \cdot out(s)$ in $P(c)$ and $in(m) \cdot in(s)$ in $P(r)$ ($out(m) \cdot out(s) \in P(c)$, $in(m) \cdot in(s) \in P(r)$). That is, for an outgoing message or incoming message m , formula $P(c)$ contains $out(m) \cdot out(s)$, where $out(m)$ denotes m is sent out by object c and $out(s)$ denotes m has signature s . $P(r)$ contains $in(m) \cdot in(s)$, where $in(m)$ denotes m is received by object r and $in(s)$ denotes m has signature s . When no information of signature is required in analysis, we can omit $in(s)$ or $out(s)$.
4. Let $m = \langle c, r_1, s_1, t_1 \rangle \in MS$ and $n = \langle c, r_2, s_2, t_2 \rangle \in MS$ be two messages sent from the same object c . Let m and n be specified in CCS as $P(m)$ and $P(n)$, respectively. Suppose n happens right after m ($m <_t n$) in a sequence diagram. Then $P(m)$ and $P(n)$ are formulated as $P(m) \cdot P(n)$ in $P(c)$.
5. Let $o \in PT$ and $alt_o = \langle o, \text{if}, [val_1 = val_2], m, n \rangle \in FG$ be the fragment denoting alternative flows of o (see Fig. 5), which means that if val_1 equals val_2 then a sequence of messages starting from m and ending with n will be executed. The alternative flows with condition $val_1 = val_2$ can be modeled by CCS formula $P(alt.o) = (in(val_1) \cdot P(mn) \cdot Q \mid out(val_2) \cdot in(val_2) \cdot Q) \setminus \{val_1\}$.
6. Suppose each object $o \in PT$ has been specified as $P(o)$ following the above steps. The CCS specification of sequence diagram $SD(X)$ is

$$CCS(X) = \left(\prod_{o \in PT} G(o) \right) [f] \setminus L$$

where $G(o) = P(o) \cdot G(o)$ formulizes the recursive occurrence of $P(o)$. L contains all the messages sent between objects. f is an optional operator used to rename messages.

Rule 1 provides the steps to specify a sequence diagram in CCS. We then show that the synchronous messages, asynchronous messages, and alternative flows of a sequence diagram are faithfully transformed into CCS formula by Rule 1.

2.3. Synchronous message

The synchronous messages in a sequence diagram are transformed into the synchronous actions in CCS formula. Fig. 3 shows synchronous messages used in sequence diagram of UML2.0. The Message Caller (object a) sends out a message and waits for the re-

turn before continuing. In this case, the message sent by the Caller is a synchronous message. We use Definition 4 to describe the characteristic of Synchronous Messages.

Definition 4. Let $SD = \langle PT, MS, AB, FG \rangle$ be the sequence diagram containing a message caller, object a , and a message receiver, object b . Let $m = \langle a, b, sig1, synchronous \rangle$ be a synchronous message from object a to object b , with signature $sig1$. Let $r = \langle b, a, sig2, asynchronous \rangle$ be the return message from object b to object a , with signature $sig2$. Let $P(a) = P_1 \cdot out(m) \cdot in(r) \cdot P_2$ be the CCS formula specifying object a , where $out(m)$ is the action sending out message m , $in(r)$ is the action receiving message r , $P_1 <_t out(m)$, and $in(r) <_t P_2$. Let $P(b) = in(m) \cdot Q \cdot out(r)$ be the CCS formula specifying object b , where $in(m)$ is the action receiving message m , $out(r)$ is the action sending out message r , and $in(m) <_t Q <_t out(r)$. The synchronous property w.r.t. synchronous message m is defined as $Q <_t P_2$, that is, action P_2 will not happen until action Q is completed.

We illustrate the synchronous message in Fig. 3. We use two messages, m and r , to represent the synchronous message and the corresponding return message, respectively. We use message $p1$ to represent any action happens before the synchronous message and message $p2$ to represent any action happens after the return message. Message q represents any action performed by object b before returning message r . While this synchronous property can be observed intuitively in the sequence diagram, it is not clear whether the derived CCS formula from Rule 1 preserves it. That is, whether the derived action of message $p2$ (P_2) happens after the derived action of message q (Q) in CCS formula ($Q <_t P_2$). We use the following theorem to prove that CCS formula preserves the synchronous property of sequence diagram.

Theorem 1. Let $SD = \langle PT, MS, AB, FG \rangle$ be the sequence diagram containing a message caller, object a , and a message receiver, object b , and $CCS(SD)$ be the CCS formula specifying SD . Let $m = \langle a, b, sig1, synchronous \rangle$ be a synchronous message from object a to object b , with signature $sig1$. Let $r = \langle b, a, sig2, asynchronous \rangle$ be the return message from object b to object a , with signature $sig2$. According to Rule 1, $P(a) = P_1 \cdot out(m) \cdot in(r) \cdot P_2$ is the CCS formula specifying object a , where $out(m)$ is the action sending out message m , $in(r)$ is the action receiving message r , P_1 is the action happens before $out(m)$ ($P_1 <_t out(m)$) and P_2 is the action happens after $in(r)$ ($in(r) <_t P_2$). According to Rule 1, $P(b) = in(m) \cdot Q \cdot out(r)$ is the CCS formula specifying object b , where $in(m)$ is the action receiving message m , $out(r)$ is the action sending out message r , and Q is the action happening after $in(m)$ and before $out(r)$ ($in(m) <_t Q <_t out(r)$). Then $CCS(SD) = P_1 \cdot Q \cdot P_2$.

Proof. According to Rule 1, we know $CCS(SD) = P(a)|P(b)\setminus\{m,r\}$. That is

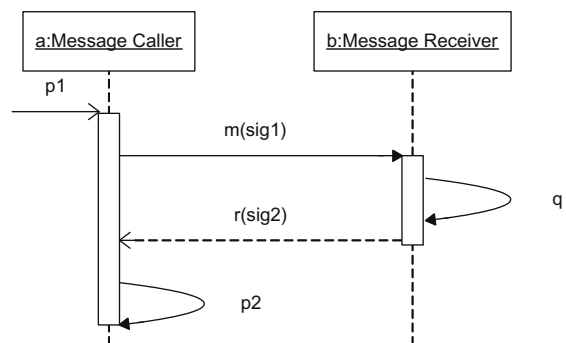


Fig. 3. Synchronous message.

$$CCS(SD) = (P_1 \cdot out(m) \cdot in(r) \cdot P_2 \mid in(m) \cdot Q \cdot out(r)) \setminus \{m, r\}$$

Since P_1 is not restricted by $\{m, r\}$, based on CCS Expansion Law in [28], it can be taken out of the parallel operation. Hence we have

$$\begin{aligned} & (P_1 \cdot out(m) \cdot in(r) \cdot P_2 \mid in(m) \cdot Q \cdot out(r)) \setminus \{m, r\} \\ &= P_1 \cdot (out(m) \cdot in(r) \cdot P_2 \mid in(m) \cdot Q \cdot out(r)) \setminus \{m, r\} \end{aligned}$$

Based on CCS Expansion law, $out(m)$ and $in(m)$ complement each other and can be merged to nil , we have

$$\begin{aligned} & (P_1 \cdot out(m) \cdot in(r) \cdot P_2 \mid in(m) \cdot Q \cdot out(r)) \setminus \{m, r\} \\ &= P_1 \cdot nil \cdot (in(r) \cdot P_2 \mid Q \cdot out(r)) \setminus \{m, r\} \end{aligned}$$

Since internal message nil can be omitted, we have

$$\begin{aligned} & P_1 \cdot nil \cdot (in(r) \cdot P_2 \mid Q \cdot out(r)) \setminus \{m, r\} \\ &= P_1 \cdot (in(r) \cdot P_2 \mid Q \cdot out(r)) \setminus \{m, r\} \end{aligned}$$

Since Q is not restricted by $\{m, r\}$, it can be taken out of the parallel operation, we have

$$P_1 \cdot (in(r) \cdot P_2 \mid Q \cdot out(r)) \setminus \{m, r\} = P_1 \cdot Q \cdot (in(r) \cdot P_2 \mid out(r)) \setminus \{m, r\}$$

Based on CCS Expansion Law, $out(r)$ and $in(r)$ complement each other and can be merged to nil , we have

$$P_1 \cdot Q \cdot (in(r) \cdot P_2 \mid Q \cdot out(r)) \setminus \{m, r\} = P_1 \cdot Q \cdot nil \cdot (P_2 \mid nil) \setminus \{m, r\}$$

Omitting nil and unnecessary restriction set, we have

$$P_1 \cdot Q \cdot nil \cdot (P_2 \mid nil) \setminus \{m, r\} = P_1 \cdot Q \cdot P_2$$

This is to say

$$\begin{aligned} CCS(SD) &= (P_1 \cdot out(m) \cdot in(r) \cdot P_2 \mid in(m) \cdot Q \cdot out(r)) \setminus \{m, r\} \\ &= P_1 \cdot Q \cdot P_2 \quad \square \end{aligned}$$

This theorem shows that synchronous messages in a sequence diagram are faithfully transformed into CCS because it shows that Q always happens before P_2 , i.e., $P_1 <_t Q < P_2$.

2.4. Asynchronous message

Different from synchronous message, a message is asynchronous when the Message Caller can send or receive other messages without waiting for the return message. Fig. 4 shows an example of asynchronous message in sequence diagram of UML 2.0. In this example, object a can send message $p2$ right after sending out message m . In this case, message q and message $p2$ may happen in any order. It may be possible for $p2$ to happen before q , and also possible for q to happen before $p2$. In other words, the design allows either order of their occurrences.

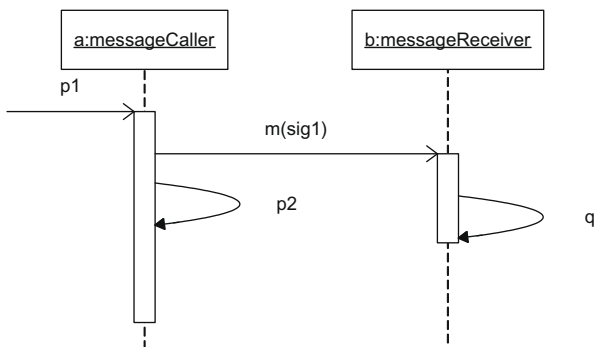


Fig. 4. Asynchronous message.

Definition 5. Let $SD = \langle PT, MS, AB, FG \rangle$ be the sequence diagram containing a message caller, object a , and a message receiver, object b , and $CCS(SD)$ be the CCS formula specifying SD . Let $m = \langle a, b, sig1, asynchronous \rangle$ be an asynchronous message from object a to object b , with signature $sig1$. Let $P(a) = P_1 \cdot out(m) \cdot P_2$ be the CCS formula specifying object a , where $out(m)$ is the action sending out message m , $P_1 <_t out(m)$ and $out(m) <_t P_2$. Let $P(b) = in(m) \cdot Q$ be the CCS formula specifying object b , where $in(m)$ is the action receiving message m , and $in(m) <_t Q$. The asynchronous property w.r.t. asynchronous message m is defined as $P_2 <_t Q \vee Q <_t P_2$. That is, action P_2 may happen before or after action Q .

We illustrate the Asynchronous Property in Fig. 4. The following theorem proves that the Asynchronous Property of sequence diagram is preserved by CCS formula.

Theorem 2. Let $SD = \langle PT, MS, AB, FG \rangle$ be the sequence diagram containing a message caller, object a , and a message receiver, object b , and $CCS(SD)$ be the CCS formula specifying SD . Let $m = \langle a, b, sig1, asynchronous \rangle$ be an asynchronous message from object a to object b , with signature $sig1$. According to Rule 1, $P(a) = P_1 \cdot out(m) \cdot P_2$ is the CCS formula specifying object a , where $out(m)$ is the action sending out message m , P_1 is the action happening before $out(m)$ ($P_1 <_t out(m)$) and P_2 is the action happening after $out(m)$ ($out(m) <_t P_2$). According to Rule 1, $P(b) = in(m) \cdot Q$ is the CCS formula specifying object b , where $in(m)$ is the action receiving message m , and Q is the action happening after $in(m)$ ($in(m) <_t Q$). Then $CCS(SD) = P_1 \cdot P_2 \cdot Q + P_1 \cdot Q \cdot P_2$.

Proof. According to Rule 1, we know $CCS(SD) = P(a) \mid P(b) \setminus \{m\}$. That is

$$(P_1 \cdot out(m) \cdot P_2 \mid in(m) \cdot Q) \setminus \{m\} = CCS(SD).$$

Since P_1 is not restricted by $\{m\}$, based on CCS Expansion Law in [28], it can be taken out of the parallel operation, hence we have

$$(P_1 \cdot out(m) \cdot P_2 \mid in(m) \cdot Q) \setminus \{m\} = P_1 \cdot (out(m) \cdot P_2 \mid in(m) \cdot Q) \setminus \{m\}$$

Based on CCS Expansion law, $out(m)$ and $in(m)$ complement each other and can be merged to nil , we have

$$P_1 \cdot (out(m) \cdot P_2 \mid in(m) \cdot Q) \setminus \{m\} = P_1 \cdot nil \cdot (P_2 \mid Q) \setminus \{m\}$$

Since internal message nil can be omitted, we have

$$P_1 \cdot nil \cdot (P_2 \mid Q) \setminus \{m\} = P_1 \cdot (P_2 \mid Q) \setminus \{m\}$$

Since Q and P_2 are not restricted by $\{m\}$, by CCS Expansion Law, we have

$$(P_2 \mid Q) \setminus \{m\} = P_2 \cdot Q + Q \cdot P_2$$

This is to say

$$\begin{aligned} CCS(SD) &= P_1 \cdot (P_2 \mid Q) \setminus \{m\} = P_1 \cdot (P_2 \cdot Q + Q \cdot P_2) \\ &= P_1 \cdot P_2 \cdot Q + P_1 \cdot Q \cdot P_2 \quad \square \end{aligned}$$

Theorem 2 shows that CCS formula truthfully preserves the asynchronous property carried by asynchronous message in sequence diagram. Since $CCS(SD) = P_1 \cdot P_2 \cdot Q + P_1 \cdot Q \cdot P_2$ action Q and action P_2 (in CCS) can happen in any order, which demonstrates that message $p2$ and message q (in Fig. 4) can be sent in any order. That is to say message m is asynchronous message.

2.5. Alternative flows

We introduced steps to specify simple sequence diagrams in CCS formula in Rule 1. After proving the synchronous/asynchronous messages can be faithfully transformed into CCS, we next show the transformation of alternative flows (see Fig. 5) into CCS

and prove its faithfulness in the following theorem. Note that the transformation of loop in a sequence diagram into CCS is out of the scope of this paper since no security patterns and properties with loops are considered in our case.

Theorem 3. Let $SD = \langle PT, MS, AB, FG \rangle$ be a sequence diagram. Let $o \in PT$ and $alt_o = \langle o, if, [val_1 = val_2], m, n \rangle \in FG$ be the fragment denoting alternative flows of o . Thus, if val_1 equals val_2 then a sequence of messages starting from m and ending with n ($\{m, \dots, n\}$) will be executed. If val_1 and val_2 are not equal then message q will be executed. According to Rule 1, the alternative flows with condition $val_1 = val_2$ can be modeled by CCS formula. Let $P(mn)$ be the action specifying messages $\{m, \dots, n\}$ (guarded by condition of alt_o); Q be the action specifying message q ; $in(val_1)$ and $out(val_2)$ be the auxiliary actions used to model the conditions. Then

$$P(alt_o) = (in(val_1) \cdot P(mn) \cdot Q \mid out(val_2) \cdot in(val_2) \cdot Q) \setminus \{val_1\}$$

Proof. We will construct a CCS formula which represents the alternative flows corresponding to alt_o . Let $P(o)$, $P(m)$ and $P(n)$ be the CCS formula specifying object o , message m , and message n , respectively. Let $Q \in P(o)$ be the process satisfying

$$\forall X \in P(o) : (P(n) \prec_t X \rightarrow Q \prec_t X) \wedge (P(n) \prec_t Q)$$

That is, Q is the process that follows $P(n)$. Let $P(mn) \in P(o)$ be the process satisfying

$$P(m) \in P(mn) \wedge (\forall X \in P(mn) : P(m) \prec_t X)$$

$$P(n) \in P(mn) \wedge (\forall X \in P(mn) : X \prec_t P(n))$$

That is, $P(mn)$ models the scope that is guarded by the condition of alt_o . Then the CCS formula specifying the alternative flows of object o can be constructed as

$$P(alt_o) = (in(val_1) \cdot P(mn) \cdot Q \mid out(val_2) \cdot in(val_2) \cdot Q) \setminus \{val_1\}$$

We show next that $P(alt_o)$, truthfully covers two branches: If $val_1 = val_2$ that is

$$\begin{aligned} (in(val_1) \cdot P(mn) \cdot Q \mid out(val_2) \cdot in(val_2) \cdot Q) \setminus \{val_1\} \\ = (in(val_1) \cdot P(mn) \cdot Q \\ \mid out(val_1) \cdot in(val_1) \cdot Q) \setminus \{val_1\} \end{aligned}$$

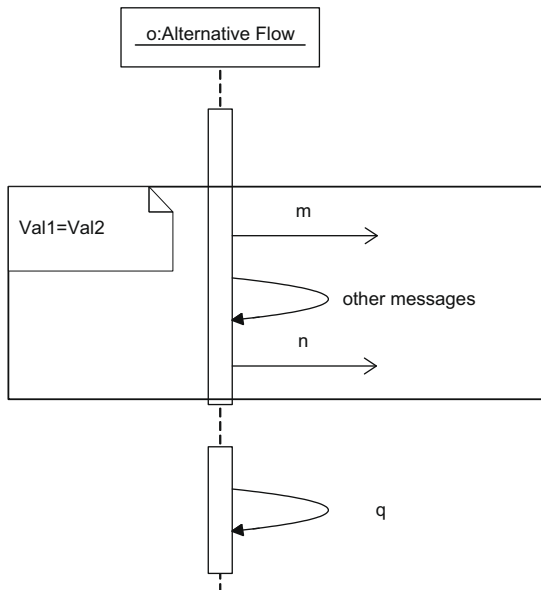


Fig. 5. Alternative flows in sequence diagram of UML 2.0.

By using CCS expansion law, $in(val_1)$ and $out(val_1)$ complement each other, hence we have

$$\begin{aligned} (in(val_1) \cdot P(mn) \cdot Q \mid out(val_1) \cdot in(val_1) \cdot Q) \setminus \{val_1\} \\ = nil \cdot (P(mn) \cdot Q \mid in(val_1) \cdot Q) \setminus \{val_1\} \\ = (P(mn) \cdot Q \mid in(val_1) \cdot Q) \setminus \{val_1\} \end{aligned}$$

Consider $P(mn)$ and Q are not restricted by $\{val_1\}$, they are taken out of the parallel operation. Thus we have

$$\begin{aligned} (P(mn) \cdot Q \mid in(val_1) \cdot Q) \setminus \{val_1\} \\ = P(mn) \cdot Q (nil \mid in(val_1) \cdot Q) \setminus \{val_1\} \end{aligned}$$

That is, when $val_1 = val_2$, we have

$$P(alt_o) = P(mn) \cdot Q (nil \mid in(val_1) \cdot Q) \setminus \{val_1\}$$

If $val_1 \neq val_2$, then according to CCS expansion law, val_2 is not a restricted message. Hence Q , $out(val_2)$ and $in(val_2)$ can be taken out of the parallel operation, thus we have

$$\begin{aligned} (in(val_1) \cdot P(mn) \cdot Q \mid out(val_2) \cdot in(val_2) \cdot Q) \setminus \{val_1\} \\ = out(val_2) \cdot in(val_2) \cdot Q \cdot (in(val_1) \cdot P(mn) \cdot Q \mid nil) \setminus \{val_1\} \end{aligned}$$

That is, when $val_1 \neq val_2$, we have

$$\begin{aligned} P(alt_o) = out(val_2) \cdot in(val_2) \cdot Q \cdot (in(val_1) \cdot P(mn) \cdot Q \\ \mid nil) \setminus \{val_1\} \end{aligned}$$

Finally, let $T_1 = (nil \mid in(val_1) \cdot Q) \setminus \{val_1\}$, and let $T_2 = (in(val_1) \cdot P(mn) \cdot Q \mid nil) \setminus \{val_1\}$. We obtain

$$P(alt_o) = \begin{cases} P(mn) \cdot Q \cdot T_1 & val_1 = val_2 \\ out(val_2) \cdot in(val_2) \cdot Q \cdot T_2 & val_1 \neq val_2 \end{cases}$$

According to CCS Expansion Law, $in(val_1)$ is a restricted action in T_1 and can never happen. Thus $T_1 = (nil \mid in(val_1) \cdot Q) \setminus \{val_1\}$ is a CCS process which yields no action, i.e., $T_1 = nil$. Therefore we obtain

$$P(mn) \cdot Q \cdot T_1 = P(mn) \cdot Q$$

According to CCS Expansion Law, $in(val_1)$ is a restricted action in T_2 and can never happen. Thus $T_2 = (in(val_1) \cdot P(mn) \cdot Q \mid nil) \setminus \{val_1\}$ is a CCS process which yields no action, i.e., $T_2 = nil$. Therefore we obtain

$$out(val_2) \cdot in(val_2) \cdot Q \cdot T_2 = out(val_2) \cdot in(val_2) \cdot Q$$

That is

$$P(alt_o) = \begin{cases} P(mn) \cdot Q & val_1 = val_2 \\ out(val_2) \cdot in(val_2) \cdot Q & val_1 \neq val_2 \end{cases}$$

Hence, in case $val_1 = val_2$, $P(alt_o)$ yields $P(mn) \cdot Q$. In case $val_1 \neq val_2$ there exists two extra actions ($out(val_2) \cdot in(val_2)$) before Q . However, $out(val_2) \cdot in(val_2)$ will not influence the order of $P(mn) \cdot Q$ and other actions occurred in the sequence diagram, because val_2 is not a message name (it is a condition name) and cannot interact with other actions (impossible to have $out(val_2)$ or $in(val_2)$ in other actions). Therefore $P(alt_o)$ also models precisely the transitions of the sequence diagram in case $val_1 \neq val_2$. \square

Informally speaking, in Theorem 3 we first construct a CCS formula $P(alt_o)$ representing alternative flow alt_o based on Rule 1. We then prove $P(alt_o)$ faithfully represents the alternative flows. $P(alt_o)$ is specially constructed to meet two criteria. First, it must include the execution of $P(mn) \cdot Q$ while excluding the execution of Q or $P(mn)$ if $val_1 = val_2$. Second, it must include the execution of Q while excluding the execution of $P(mn)$ if $val_1 \neq val_2$. To prove these two criteria, we need to show that T_1 and T_2 will terminate

Table 1
CCS transformation of sequence diagram.

Sequence diagram	CCS formula
$a \in PT$	$P(a)$ is a CCS formula
$m = \langle c, r, s, t \rangle \in MS, c = r$	$action(m) \cdot action(s) \in P(c)$ [1]
$m = \langle c, r, s, t \rangle \in MS, c \neq r$	$out(m) \cdot out(s) \in P(c) \cdot in(m) \cdot in(s) \in P(r)$ [2]
$alt_o = \langle o, if, [val_1 = val_2], m, n \rangle \in FG$	$P(alt_o) = (in(val_1) \cdot P(mn) \cdot Q)out(val_2) \cdot in(val_2) \cdot Q \setminus \{val_1\}$

the execution (equal to *nil*). To be more specific, we know each CCS formula is actually a state transition system, and each execution of the formula is a system transition from one state to another. In case $val_1 \neq val_2$, $P(alt_o)$ will transit to $Q \cdot T_2$ (by executing $out(val_2) \cdot in(val_2)$) and then to T_2 (by executing Q). Since T_2 has no transition, the system execution is terminated. Hence we proved that Q is executed while $P(mn)$ is not executed.

Theorem 3 shows that the guard condition is assumed to be $val_1 = val_2$, where val_1 and val_2 are two CCS variables which can take actual value when being executed. In general $val_1 = val_2$, is the guard condition for the case when val_1 is equal or not equal to val_2 . In the case that the guard condition is $val_1 > val_2$ or $val_1 < val_2$ we can make the following transformation:

Let $x = val_1 - val_2$ and $y = abs(val_1 - val_2)$ (abs denotes the absolute value of $val_1 - val_2$), then the guard condition $val_1 > val_2$ is equivalent to $x = y$ whereas $val_1 < val_2$ is equivalent to $x \neq y$. Therefore a general guard condition (including $=, \neq, >, <$) and the alternative flows led by it can all be specified by **Theorem 3**.

Table 1 summarizes how each element of a sequence diagram can be modeled by CCS formula.

- [1] $c = r$ implies that the caller and receiver are the same object, therefore m is an action of object c , which is modeled as CCS formula $action(m)$. Signature s is modeled as $action(s)$.
- [2] t is not modeled in CCS formula because synchronous/asynchronous information will be automatically preserved by CCS formula, which was proved by **Theorem 1**. Signature s is modeled as $out(s)$ and $in(s)$ in $P(c)$ and $P(r)$, respectively, where $out(s)$ and $in(s)$ can be omitted as stated in **Rule 1**.

Since we have proved that synchronous message, asynchronous message and alternative flows can each be truthfully transformed into CCS formula, we will further prove that any sequence diagram composed of synchronous messages, asynchronous messages and alternative flows can be truthfully transformed into CCS formula.

Theorem 4. Suppose $SD = \langle PT, MS, AB, FG \rangle$ is a sequence diagram containing synchronous messages m , and return message r . Suppose SD contains asynchronous message n which is sent from object a to object b . Suppose SD contains alternative flows alt . Suppose SD is transformed into CCS formula, $CCS(SD)$, by **Rule 1**. Then $CCS(SD)$ truthfully preserves synchronous property, asynchronous property and alternative flows.

Proof. Without losing generality, we assume that synchronous messages m and its return message r are sent before asynchronous message n . According to **Theorem 1**, $CCS(SD) = P_1 \cdot Q \cdot P_2$, where $P_1 <_t out(m)$, $in(r) <_t P_2$ and $in(m) <_t Q <_t out(r)$. Considering P_2 is defined as the actions after $in(r)$. Thus, P_2 contains the action sending out asynchronous message n ($out(n)$) and the actions after $out(n)$. According to **Theorem 2**, $P_2 = P_3 \cdot Q_1 + Q_1 \cdot P_3$, where $out(n) <_t P_3$ and $in(n) <_t Q_1$. That is,

$$CCS(SD) = P_1 \cdot Q \cdot P_2 = P_1 \cdot Q \cdot (P_3 \cdot Q_1 + Q_1 \cdot P_3) \\ = P_1 \cdot Q \cdot P_3 \cdot Q_1 + P_1 \cdot Q \cdot Q_1 \cdot P_3$$

Hence it is easy to see the synchronous property and asynchronous property are preserved by $CCS(SD)$. Suppose $P(alt)$ is the CCS formula specifying alt and $P(guarded)$ is the CCS formula specifying actions to be executed if the condition ($val_1 = val_2$) is satisfied. To maintain generality, we consider four situations: $P(alt) = P_1$, $P(alt) = Q_1$, $P(alt) = Q$, and $P(alt) = P_3$ respectively (that is, the alternative flows can appear as any action).

If $P(alt) = P_1$, according to **Theorem 3**, we obtain

$$P_1 \cdot Q \cdot P_3 \cdot Q_1 = P(alt) \cdot Q \cdot P_3 \cdot Q_1 \\ = \begin{cases} P(guarded) \cdot Q \cdot P_3 \cdot Q_1 \cdot T_1 & val_1 = val_2 \\ out(val_2) \cdot in(val_2) \cdot Q \cdot P_3 \cdot Q_1 \cdot T_2 & val_1 \neq val_2 \end{cases}$$

and

$$P_1 \cdot Q \cdot Q_1 \cdot P_3 = P(alt) \cdot Q \cdot Q_1 \cdot P_3 \\ = \begin{cases} P(guarded) \cdot Q \cdot Q_1 \cdot P_3 \cdot T_1 & val_1 = val_2 \\ out(val_2) \cdot in(val_2) \cdot Q \cdot Q_1 \cdot P_3 \cdot T_2 & val_1 \neq val_2 \end{cases}$$

Hence,

$$CCS(SD) = \begin{cases} P(guarded) \cdot Q \cdot P_3 \cdot Q_1 \cdot T_1 + P(guarded) \cdot Q \cdot Q_1 \cdot P_3 \cdot T_1 & val_1 = val_2 \\ out(val_2) \cdot in(val_2) \cdot Q \cdot P_3 \cdot Q_1 \cdot T_2 + out(val_2) \cdot in(val_2) \cdot Q \cdot Q_1 \cdot P_3 \cdot T_2 & val_1 \neq val_2 \end{cases}$$

Thus,

$$CCS(SD) = \begin{cases} P(guarded) \cdot Q \cdot (P_3 \cdot Q_1 \cdot T_1 + Q_1 \cdot P_3 \cdot T_1) & val_1 = val_2 \\ out(val_2) \cdot in(val_2) \cdot Q \cdot (P_3 \cdot Q_1 \cdot T_2 + Q_1 \cdot P_3 \cdot T_2) & val_1 \neq val_2 \end{cases}$$

It is easy to observe here $Q <_t P_3 \cdot Q_1 + Q_1 \cdot P_3$, that is $Q <_t P_2$. Hence the synchronous property is preserved. Similarly, we could prove asynchronous property is preserved. Moreover, by **Theorem 3**, we could prove alternative flow is preserved by $CCS(SD)$.

By similar way of reasoning, those properties are preserved in case that $P(alt) = Q_1$, $P(alt) = Q$, and $P(alt) = P_3$. \square

We illustrate this proof in **Fig. 6**. The CCS formula (bold) for each message is added beside the message.

3. Case studies

In this section, we present two case studies to illustrate our approach. In particular, we show the detection of several subtle design errors by using our analysis techniques. More specifically, **Section 3.1** describes the Secure Pipe and Authentication Enforcer patterns. **Section 3.2** formally specifies these patterns in CCS. **Section 3.3** introduces an application of secure observer by composing the Observer pattern with security patterns. **Section 3.4** presents

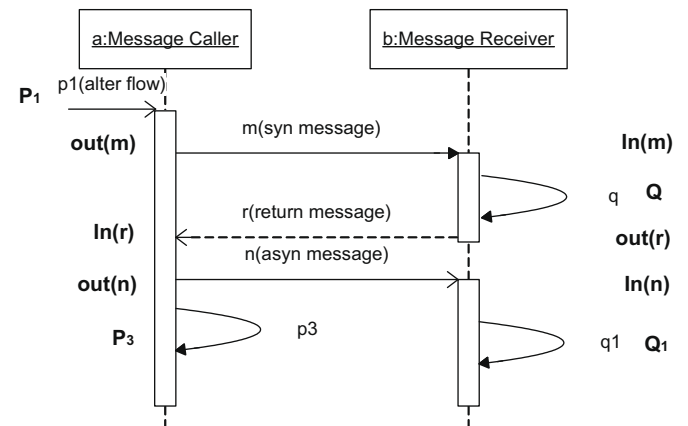


Fig. 6. Synchronous/asynchronous messages and alternative flows.

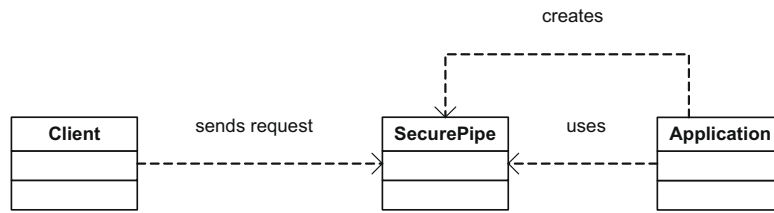


Fig. 7. Secure Pipe pattern class diagram.

our analysis of these security pattern compositions. Section 3.5 introduces another case study on the composition of web service security patterns.

3.1. Security patterns

The Secure Pipe pattern provides a solution to guarantee the integrity and privacy of data sent over the network. The secure pipe does not require application-layer logic and provides a simple and standard way to protect data. The task of securing a pipe is moved to the hardware platform to reduce the complexity of implementation. In some case, it is even moved out of the hardware platform.

Fig. 7 shows a class diagram of the Security Pipe pattern related to an application. Initially, the client may login to the application. The application then creates a Secure Pipe at the system level. The Secure Pipe is an encrypted communication channel over which the client communicates with the application. The channel is secure, which provides data privacy and integrity between two end-

points. When the client logs out, the application destroys the Secure Pipe.

Fig. 8 shows a sequence diagram of the Security Pipe pattern which depicts the activities of each object in an application. When the client wants to send information to the application in a secured channel, it will inform the application by calling the “login” method of the application that creates a secure pipe. The client will then call the “request” method to pass the information to the secure pipe which encrypts the information and sends it to the application.

Another common security problem is authentication. Only valid users can access certain application resources. These users must be properly authenticated. There are several different ways to authenticate users, such as HTTP basic authentication, form-based authentication, certificate-based authentication, and custom authentication via JAAS. These authentication mechanisms can be used jointly in the same application. Due to changes of business requirements, application-specific characteristics, and underlying security infrastructure, users may change the choice of these

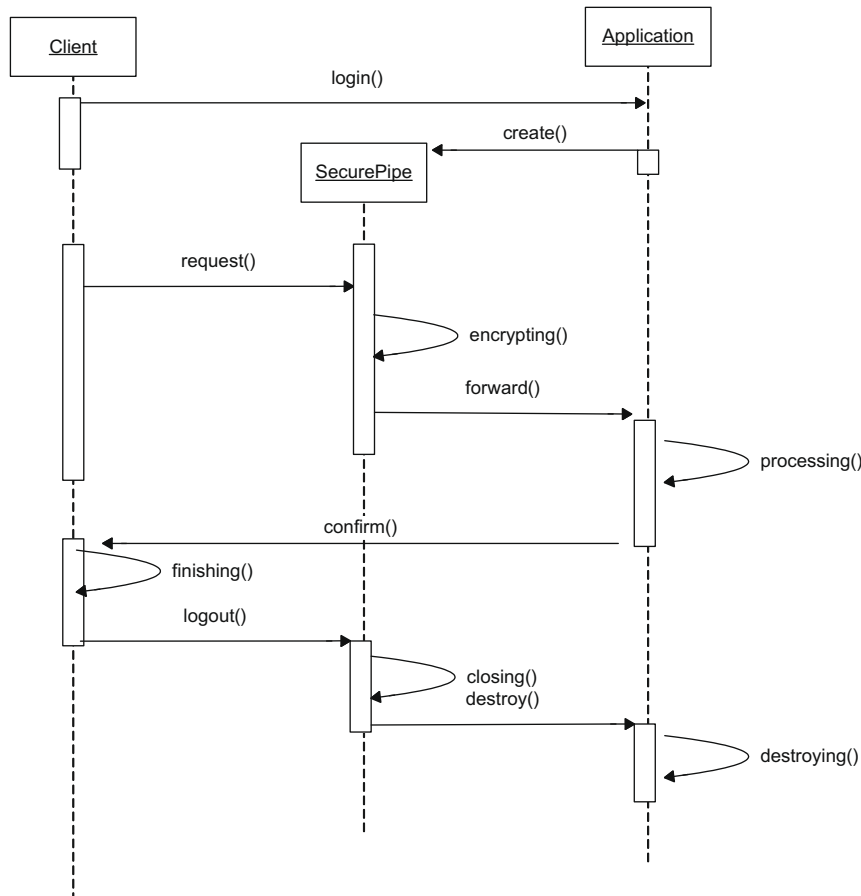


Fig. 8. Secure Pipe pattern sequence diagram.

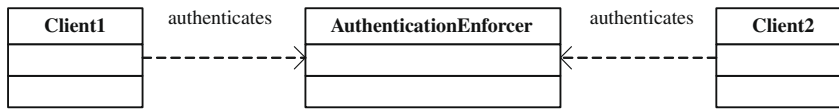


Fig. 9. Authentication Enforcer pattern class diagram.

authentication mechanisms. In addition, there may be multiple entry points into an application, each requiring user authentication. Therefore, the code related to authentication is duplicated in many places making the system difficult to develop and easy to make mistakes.

The Authentication Enforcer pattern provides a solution to centralize the user authentication processes and encapsulate the details of the authentication mechanism. The authentication logic for verifying user identity is delegated to a helper class that interacts with the security providers. This centralized mechanism applies to all different kinds of authentications, such as password-based and client certificate-based authentications. The benefits of centralizing and encapsulating authentication mechanisms behind a common interface include easing authentication requirement evolution and facilitating reuse. The generic interface allows the authentication to be independent from the protocols.

Fig. 9 depicts the Authentication Enforcer pattern where the Authentication Enforcer centralizes and encapsulates the authentication logic. When two clients need to authenticate one another, they can achieve that through the Authentication Enforcer class. More specifically, the Authentication Enforcer class authenticates a user by using the credential passed in the Request Context that contains the user's credentials. The Authentication Enforcer creates a subject instance that represents the authenticated user.

Fig. 10 shows a sequence diagram of the Authentication Enforcer pattern. When client2 wants to build a trust relationship with client1, it calls the "start" method of client1 that sends its privacy information to Authentication Enforcer. The identity of client1 is verified by the Authentication Enforcer. After client1 is identified, it calls the "start" method of client2 that will proceed the same

way to get itself identified. Since both client1 and client2 are identified by a trustworthy third party, Authentication Enforcer, they can then trust each other.

3.2. Formal specifications

In this section, we present the formal specification of the Secure Pipe and Authentication Enforcer patterns based on our general specification template described in Section 2. Since the model specification language of CWB-NC has some minor syntactical difference from our CCS representation, we do not write "in", "out", "action" as the label of each message. Instead, we write the action name directly for any "in", "out", or "action" message. In addition, a prime (') symbol is prefixed before a message name for any outgoing message. In the following example, "spstart" is an incoming message, which is denoted as *in(spstart)* in our CCS representation, and "splogin" is an outgoing message, which is denoted as *out(splogin)* in our CCS representation.

In particular, the behavior of each security pattern is formalized as a group of processes and their communications based on our approach in Section 2. The behavioral specification of the Secure Pipe pattern is presented as follows¹:

```

* sp is used as prefix to all process and action names
of the Secure Pipe pattern *
proc SECUREPIPE = (SPCLIENT|SPPIPE|SPAPP)\
  {splogin,spcreate,sprequest,
  spforward,spconfirm,splogout,spdestroy}
proc SPCLIENT=
  spstart.'splogin.SPCLIENT1+
  spnegotiate.'sprequest.SPCLIENT1+
  spconfirm.spfinishing.'splogout.SPCLIENT1
proc SPCLIENT1 = SPCLIENT
proc SPPIPE = spcreate.SPPIPE0
proc SPPIPE0=
  sprequest.spencrypt.'spforward.SPPIPE1+
  splogout.spclosing.'spdestroy.SPPIPE1
proc SPPIPE1 = SPPIPE0
proc SPAPP=
  splogin.'spcreat.SPAPP1+
  spforward.spprocessing.'spconfirm.SPAPP1+
  spdestroy.spdestroying.SPAPP1
proc SPAPP1 = SPAPP
    
```

where the Secure Pipe pattern is specified by the parallel composition of three processes SPCLIENT, SPPIPE, and SPAPP, which refer to client, securepipe, and application processes, respectively. The SPCLIENT process sends out the 'splogin' when it starts. When it gets the 'spnegotiate' message, it will send out 'sprequest' to the SPPIPE process. When the SPCLIENT gets the 'spconfirm', it will log-out. When the SPPIPE process gets the 'sprequest' from the SPCLIENT, it will perform an encryption action and forward the request to the SPAPP. When the SPPIPE process gets the message

¹ Note that we simplify the process expressions in(X, X) and out(X, X) to in(X) and out(X), respectively, when the port name and message name are the same. The sequential composition (':') has higher priority than the non-deterministic choice ('+') in CCS.

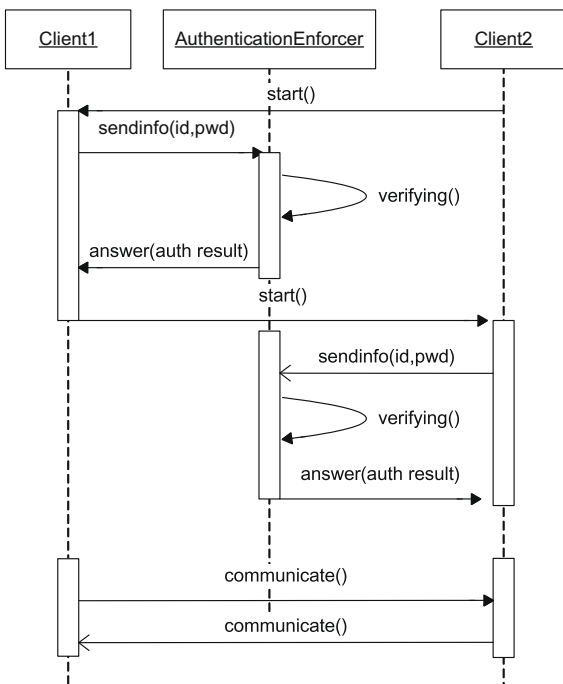


Fig. 10. Authentication Enforcer pattern sequence diagram.

for creation, it will prepare. The SPPIPE process will create a SPPIPE when it gets 'spcreate'. When the SPAPP gets the request forwarded by the SPPIPE, it will process the 'sprequest' and send out the 'sconfirm' to the SPCLIENT. When the SPAPP receives the request to close, it will destroy the connection.

The behavior of the Authentication Enforcer pattern is specified in CCS as follows:

```
* ae is used as prefix to all process and action names
of the authentication enforcer *
proc AUTH_ENFORCER = AECLIENT1|AEMAIN|AECLIENT2\
{aeidentity, aepassword, aeauthresult}
proc AECLIENT1 = aestartup. 'aeidentity. 'aepassword +
aeauthresult. 'aelfinish+ae2finish.aelcommunicate
proc AEMAIN = aeidentity.aepassword.aeverify.
'aeauthresult
proc AECLIENT2 = aestartup. 'aeidentity. 'aepassword +
aeauthresult. 'ae2finish + aelfinish.ae2communicate
```

where the AECLIENT1 and AECLIENT2 processes send their identities and passwords to the AEMAIN process when they want to communicate with each other. The AEMAIN receives and authenticates their information before replying with message aeauthresult. When the clients get the authentication results, they will start the communication. When the AEMAIN (Authentication Enforcer) process receives the identity and password of a client, it will verify and then send out the authentication result.

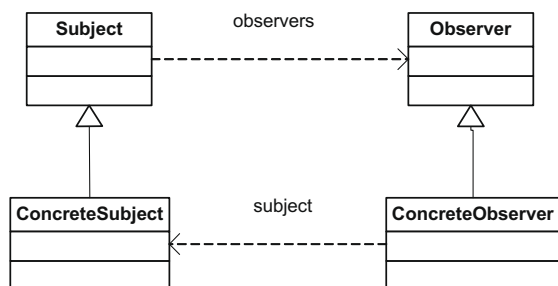


Fig. 11. Observer pattern class diagram.

3.3. Secure observer

Let us consider a case of secure observer. The Observer pattern has been introduced in [19], which addresses the problem of maintaining consistency among different copies of the same data. In practice, a data may have different views which present it in some special ways. When the data is changed, all its views need to be notified and changed. The Observer pattern provides a solution to this problem by loosely connecting the data and its views through an attaching and detaching mechanism. Fig. 11 shows the class diagram of the Observer pattern. The CCS specification of the behavior of the Observer pattern is shown in Section 2.

In normal Observer pattern, there is no security concern. However, there are many situations where both the subject and the observers need to authenticate each other. The subject can be viewed only by authorized observers, whereas an observer may only intend to view trusted subject. For example, the sales information of a company can be only viewed by the sales manager who has the most up-to-date information. A stock market watcher may only observe the market information from trusted source (subject). In addition, the communication between the subject and the observers should be secure. The sensitive data should not be intercepted by malicious attackers. In these situations, we need to add security solutions to the Observer pattern. In particular, the Secure Pipe and Authentication Enforcer patterns can be applied to solve these problems. Figs. 12 and 13 depict a composition of the Observer, Secure Pipe, and Authentication Enforcer patterns. The Authentication Enforcer class is used to authenticate the subject and observers. The Secure Pipe class is used to provide secure communication between the subject and observers.

3.4. Analysis of security pattern compositions

In the previous section, we introduced a composition of two security patterns with the Observer pattern. In this section, we will analyze this composition. We are interested in knowing whether there are any problems or errors in the composition shown in Figs. 12 and 13.

Based on our analysis approach described in Section 2, we have formally specified the behavior of the Secure Pipe, Authentication Enforcer, and Observer patterns in the previous sections. Therefore, the behavioral composition of the three patterns can be specified based on Fig. 13 as follows:

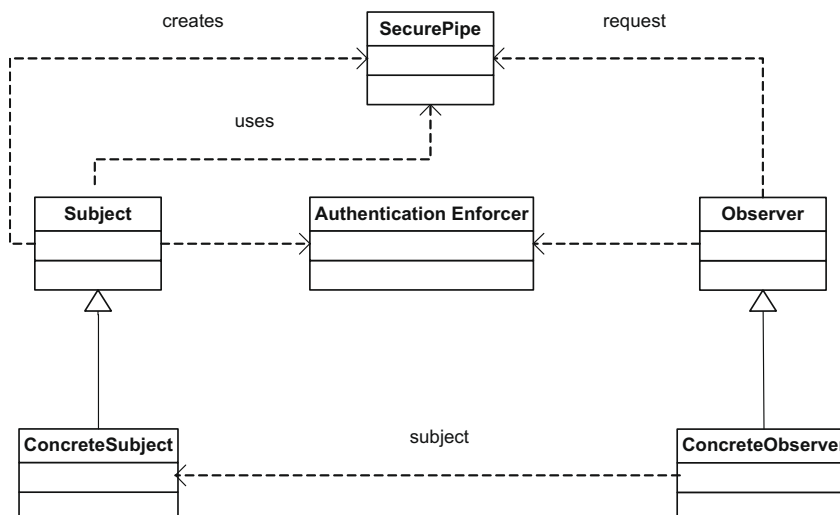


Fig. 12. Secure Observer class diagram.

```

Set Internals = {splogin,spcreate,sprequest,spforward,
spconfirm,splogout,spdestroy,aeidentity,aepassword,
aeauthresult,osetstate,ouupdate}
*integration of the Observer, Securepipe and Authen-
tication patterns*
proc INT = (SUB|SPPIPE|AEMAIN|OBS)\Internals
    
```

where the SUB, SPPIPE, AEMAIN, and OBS stand for subject process, securepipe process, authentication enforcer process, and observer process, respectively. The definition of these processes can be found in file int.ccs in the [Appendix](#).

In our analysis, we are interested in whether the communications between the Subject and the Observers are really secure in this composition. In particular, we are interested in the following properties:

1. The authentication between the Subject and Observer always happens after a secure pipe has been established between them.
2. Any message from Subject to Observer is always sent after they authenticate each other. Any message from Observer to Subject is always sent after they authenticate each other.

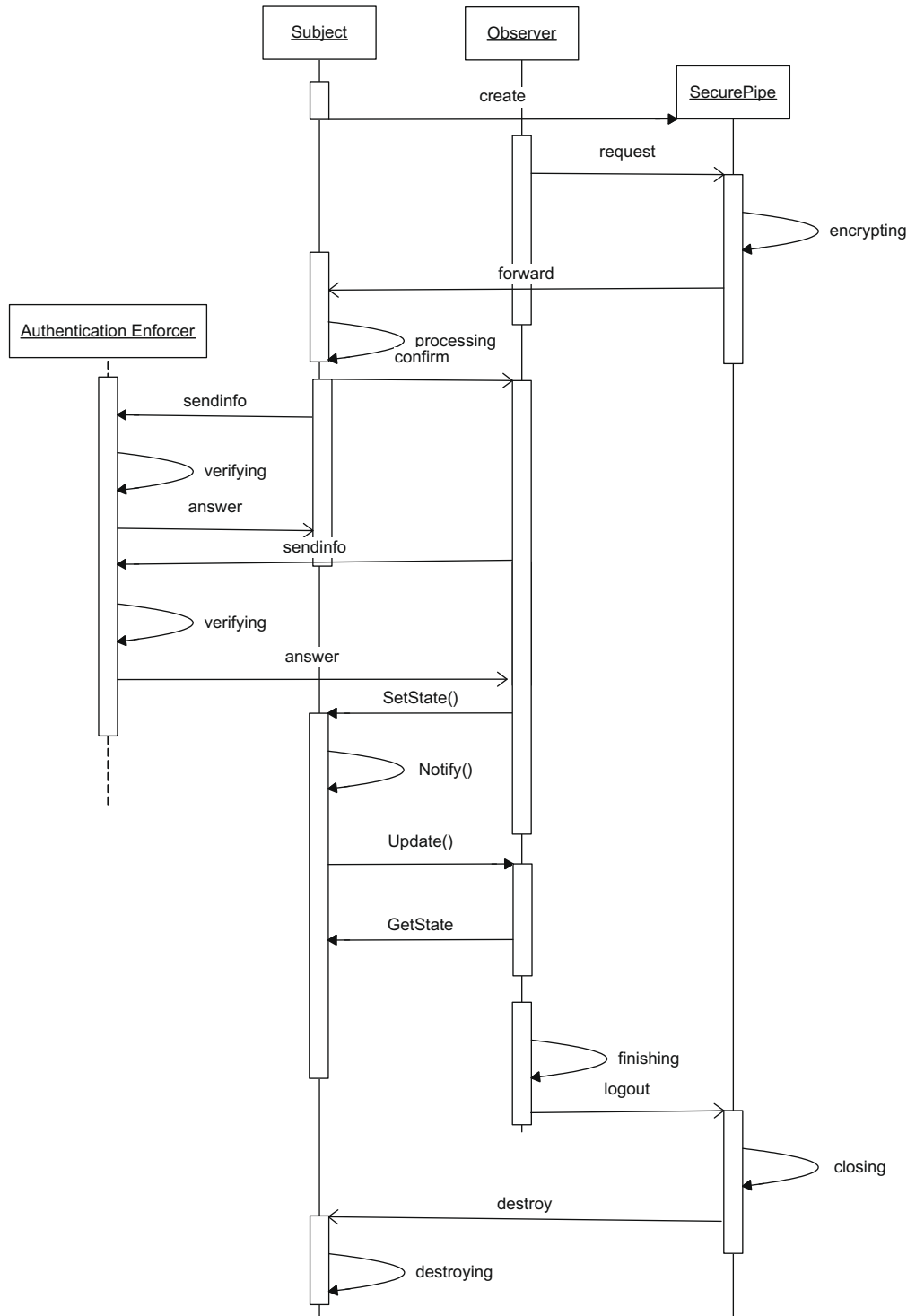


Fig. 13. Secure Observer sequence diagram.

3. After authentication, any message sent from Subject to Observer should always be secure.

We used the CWB-NC model checker to check Property 1, which is implemented as 'safe_auth_guarantee' (see int.gctl in the Appendix) and found that the communication is actually not secure, because the following model checking result shows that property safe_auth_guarantee is not satisfied.

```
cwb-nc> chk -L gctl INT safe_auth_guarantee
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 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 5 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 4 times.
Stored 0 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.016,0.000,0.000,0.016)
```

After studying the composition of this application, we found the source of the problem. When the Observer tries to authenticate itself to the Subject, it connects to the Authentication Enforcer and provides its identity information to the Authentication Enforcer. However, the problem is that the communications between the Subject/Observer and Authentication Enforcer are not secure. They are not protected by the Secure Pipe. To secure this communication, thus, new Secure Pipes are needed in-between the Subject/Observer and the Authentication Enforcer. The improved integration is shown in Fig. 14, where two new Secure Pipes, Secure Pipe A and Secure Pipe B, are added. Fig. 15 shows how the observer verifies its identity by the authentication enforcer through a secured channel. The observer sends its identity information to the

securepipe using the "request" method. The securepipe encrypts this information and forwards it to the authentication enforcer, which will decrypt and verify it. The interactions between authentication enforcer and subject are the same. The behavioral composition of these five pattern instances is specified in CCS as follows:

```
*improvement of INT*
set Internals= {splogin,spcreate,sprequest,spforward,spconfirm,splogout,spdestroy, aeidentity, aepassword, aeauthresult, osetstate, oupdate}
*integration of Observer, Securepipe and Authentication pattern*
proc INT1= (SUB|SPPIPE|AUTH|OBS|SPPIPEA|SPPIPEB)\Internals
```

where the SUB, SPPPIPE, AUTH, OBS, SPPPIPEA, SPPPIPEB refer to process Subject, Secure Pipe, Authentication Enforcer, Observer, Secure Pipe A, Secure Pipe B, respectively. The details of specification INT1 can be found in file int1.ccs in the Appendix. The following results show that Property 1 is satisfied in the revised composition, by checking two processes SPPPIPEA and SPPPIPEB (see Appendix int1.ccs) against safe_spa_guarantee and safe_spb_guarantee (see Appendix int1.gctl), respectively.

```
cwb-nc> chk -L gctl SPPPIPEA safe_spa_guarantee
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 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 5 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 13 times.
Stored 8 dependencies.
TRUE, the agent satisfies the formula. Execution time (user,system,gc,real):(0.016,0.000,0.000,0.016)
```

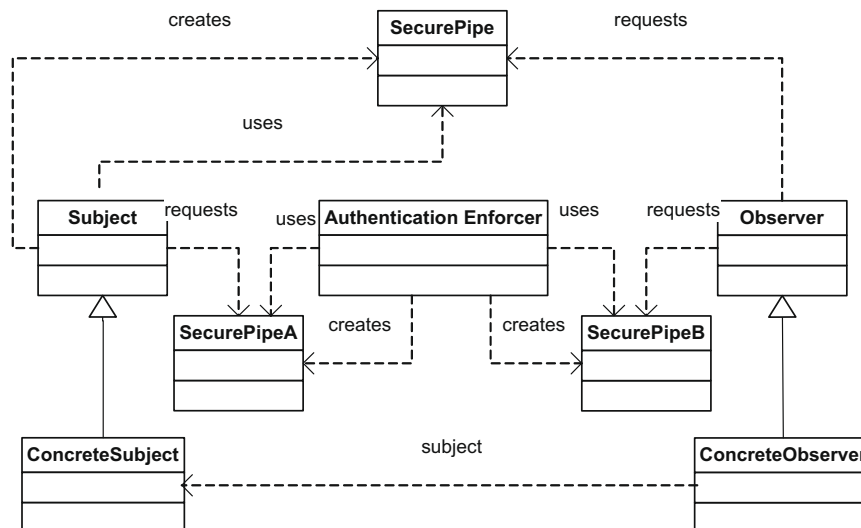


Fig. 14. Secure Observer class diagram (revised version one).

```

cwb-nc> chk -L gctl SPIPEB safe_spb_guarantee
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 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 5 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 13 times.
Stored 8 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.016,0.000,0.000,
0.016)
    
```

```

cwb-nc> chk -L gctl INT1 safe_send_setstate
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 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 5 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 4 times.
Stored 0 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.422,0.000,0.000,
0.422)
    
```

We will then check integration version one against other properties. The following result shows that Property 2, which is implemented as `safe_send_setstate` (see [Appendix int1.gctl](#)), is satisfied by integration version one.

We further studied the secure pipes in the revised design in [Fig. 14](#) (class diagram) and [Fig. 15](#) (sequence diagram) by using the CWB-NC model checker to check it against Property 3 (see [Appendix int1.gctl](#)).

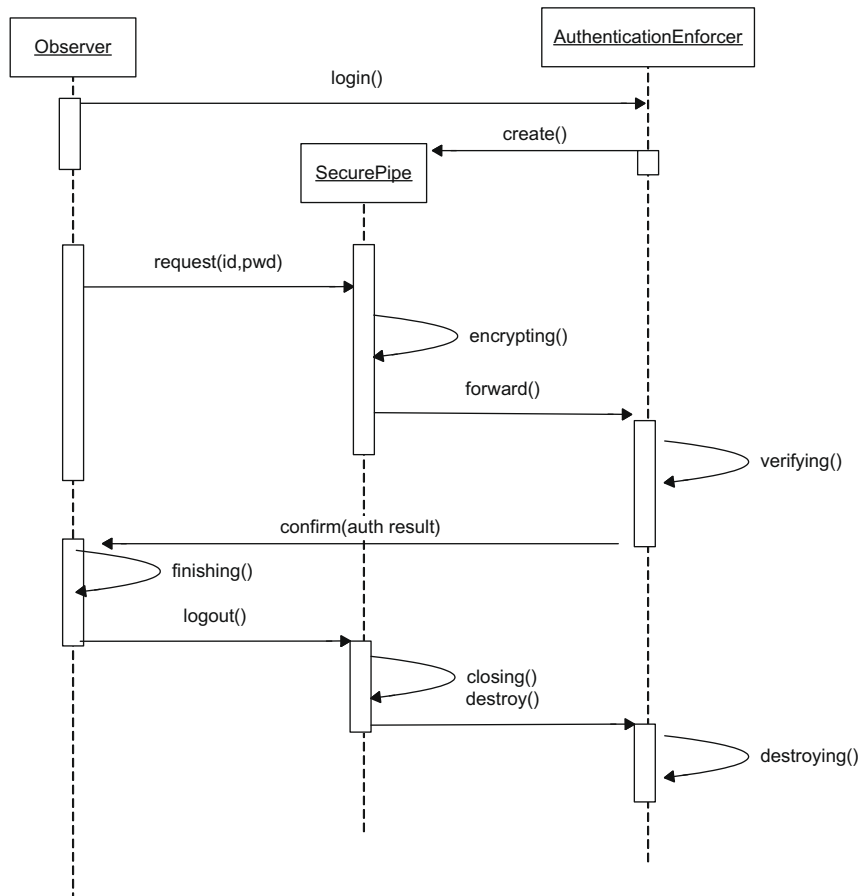


Fig. 15. Part of Secure Observer sequence diagram (revised version one).

```

cwb-nc> chk -L gctl INT1 secure_send_setstate
Generating ABTA from GCTL* formula...done
Initial ABTA has 11 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 9 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 5 times.
Stored 0 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.047,0.000,0.000,
0.047)
    
```

The above results showed Property 3 is not satisfied. After reviewing the system design, we found that there is a problem with the secure pipe between the Subject and Observer. The Subject and Observer assume wrong roles in the Secure Pipe pattern. In Fig. 14, the Subject and Observer play the roles of Application and Client, respectively. However, the Subject is supposed to send notification to the Observer. Thus, the Subject should be the Client, whereas the Observer should play the role of Application in the Secure Pipe pattern. The Subject and Observer should switch their roles. A new revised design, shown in Fig. 16, corrects this problem.

3.5. Composition of web service security patterns

In the previous sections, we introduced a case study on the composition of the Secure Pipe and Authentication Enforcer patterns. In this section, we present another case study on the composition of web service security patterns.

Authentication is one of the most fundamental problems in web service security. The Authentication Enforcer pattern presents a general solution to this problem. In web service security, there can be different methods to authenticate clients, such as the direct and brokered authentication [42]. In the direct authentication, the client directly proves its identity to the server. In the brokered authentication, both client and server trust a third party that handles the authentication. There are several algorithms and protocols for brokered authentication, such as X.509 PKI, Kerberos, and Security Token [41]. Each algorithm has its own pros and cons and is suitable for different applications. The client may prove its identity by using any of these algorithms and protocols. In some applications, such as e-business, it is very helpful and even essential to let the client be able to choose among these protocols so that both convenience and security can be achieved.

Fig. 17 illustrates the structure of the Kerberos pattern. There are three participants, the Key Distributing Center (KDC), the Client, and the Service. The KDC is responsible for storing and distributing certificates. The Client and Service are two interacting participants in an application.

Fig. 18 illustrates the behavior of the Kerberos pattern, which implements the Kerberos key change protocol. In Kerberos, the client requests a session key in the form of ticket from the key distribution center (KDC), and passes it to the service for authentication. Then the system starts. The Client sends out ticketReq to KDC that generates a ticket for the Client. When the Client gets the ticket, it then sends to the Service by sendReq to ask for service. After validating the Client's ticket, the Service will reply and begin its service.

The processes in the Kerberos pattern can be specified as follows:

```

proc Kerberos_Client = ' ticketReq+servTicket+' send-
Req+sendResp
proc Kerberos_KDC = ticketReq.genTicket.' servTicket
proc Kerberos_Service = sendReq.validateTicket.
'sendResp
    
```

Fig. 19 shows the structure of the X509 pattern. There are two parts in the system, Client and Service. The Service generates certificate from the credentials provided by the Client and responds to the Client directly. The Service may use a broker. In this case, it is not necessary to involve a third party.

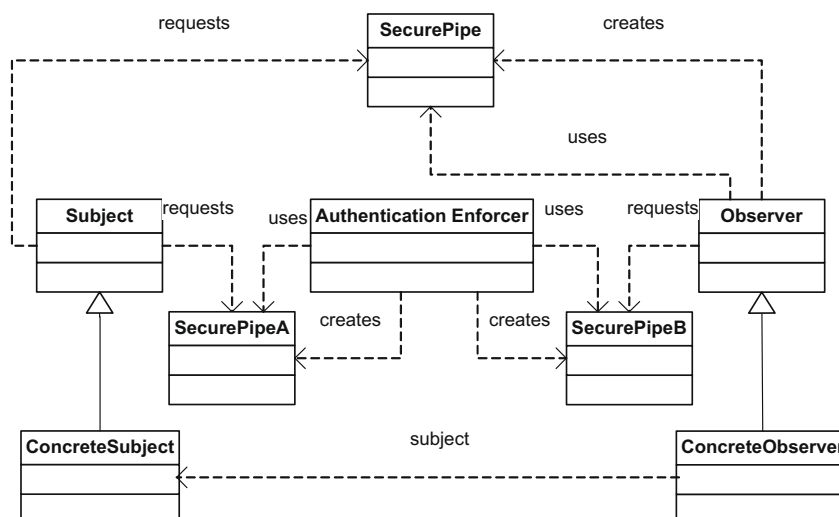


Fig. 16. Secure Observer (revised version two).

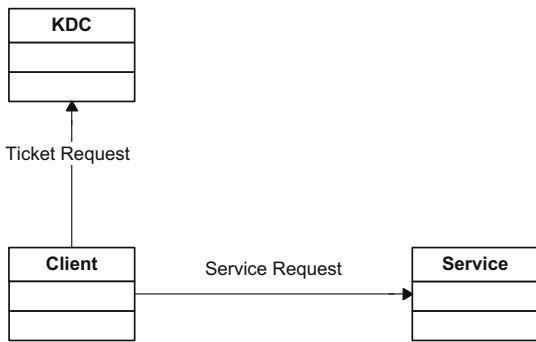


Fig. 17. Kerberos pattern class diagrams.

Fig. 20 shows the behavior of the X509 pattern, which implements the X509 key exchange protocol. The client communicates directly to the service which retrieves the client's certificate from its database or a certificate provider and then validates the certificate. In fact, the Kerberos and X509 are two protocols with the same intent, authenticating the client to the service. The process in the X509 pattern can be specified in CCS as follows:

```

proc X509_Client=sign.'request.response.Kerberos_
Strategy
proc X509_Service=request.retrieveCert.validate-
Cert.verifySignature.'response
    
```

When the system starts, the client sends a request containing its credentials to the service, which retrieves a certificate according

to those credentials and validates the certificate. The service will then send a response to the client.

In order to allow the user to flexibly change different key exchange protocols, the Strategy pattern [19] can be applied. Fig. 21 displays the class diagram of the Strategy pattern.

The composition of the Kerberos, X509 and Strategy patterns is shown in Fig. 22, where either Kerberos or X509 protocol can be used. The application of the Strategy pattern allows the user to dynamically change the protocol being used for authentication. Therefore, the composition of these three patterns provides multiple options for brokered authentication.

Consider the processes of Kerberos and X509 defined previously. The CCS expression of the integration is as follows (note that we name some process differently from their original names in the X509 or Kerberos patterns):

```

proc KDC=ticketReq.genTicket.'servTicket.KDC
proc KService=sendReq.validateTicket.'sendResp.
KService
proc X_Strategy=use_ X509.sign.'request.response.
Kerberos_Strategy
proc XService=request.retrieveCert.validateCert.
verifySignature.'response.XService
proc testMauth= (Strategy|Kerberos_Strategy|X_
Strategy)\{use_kerberos, use_X509}
set L={use_kerberos,use_X509,request,response,tick-
etReq,servTicket, sendReq,sendResp}
proc Multi_Authentication=
(KService|KDC|Kerberos_Strategy|Strategy|X_
Strategy|XService)\L
    
```

where L is the set of internal messages in the integration. L actually includes all messages used in this integration.

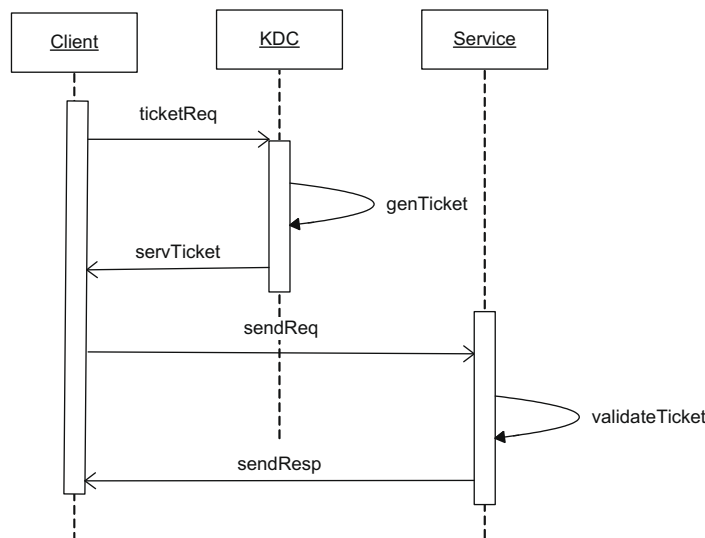


Fig. 18. Kerberos pattern sequence diagrams.

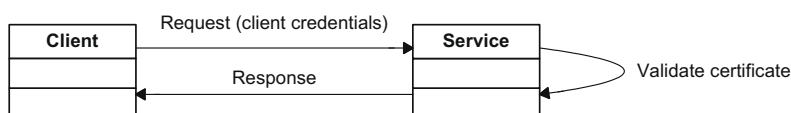


Fig. 19. X509 pattern class diagram.

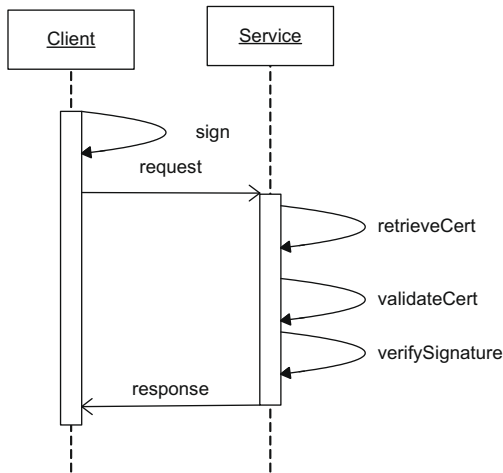


Fig. 20. X509 pattern sequence diagram.

There are several properties, such as safety and liveness, which can be checked to ensure the correctness of the composition. The following results show that our system specification satisfies a secure liveness property, that is, the authentication will eventually happen, either authenticated by X509 or by Kerberos. The specification is in “testwss.ccs” and property is in “testwss.gctl” in the Appendix. The following result shows that the composition satisfies the secure liveness property.

```
cwb-nc> chk -L gctl Multi_Authentication auth_happen
Generating ABTA from GCTL* formula...done
Initial ABTA has 8 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 5 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 35 times.
Stored 0 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.016,0.000,0.000,
0.016)
```

Besides liveness, other properties can be checked. In the following, we show the result of checking the atomicity property in the authentication process. A group of actions, such as retrieving certificate and validating certificate, need to be performed when authenticating. There should not be any other action that intercepts this group of actions. Thus, retrieving certificate and validating certificate have to be performed together without any interruption. This atomicity property can be specified in GCTL as follows:

$$prop \ x_integrity = AG(\{retrieveCert\} -> X\{validateCert\})$$

This GCTL expression represents that in all paths of the software system, there globally holds a fact that whenever action retrieveCert is performed, validateCert must be performed in the next step. In other words, action retrieveCert and action validateCert cannot be interrupted by any other actions. Similarly, the validateCert and verifySignature actions should also be performed together without any interruption by other actions. This property ensures the successful authentication of the XService process. The detailed specification of this property is presented in “testwss.gctl” in the Appendix. We run the model checker to test if this property is satisfied by this multi-authentication composition.

```
cwb-nc> chk -L gctl Multi_Authentication x_integrity
Generating ABTA from GCTL* formula...done
Initial ABTA has 9 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 6 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 17 times.
Stored 0 dependencies.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.016,0.000,0.000,
0.016)
```

The above result shows that this property is not satisfied by our composition, which indicates that the X509 authentication process may be interrupted by some other actions. Thus, the violation of this property could cause serious security problems such that some malicious actions may be inserted between the

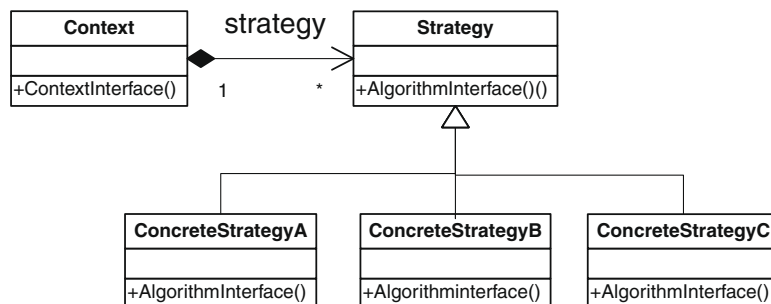


Fig. 21. Strategy pattern.

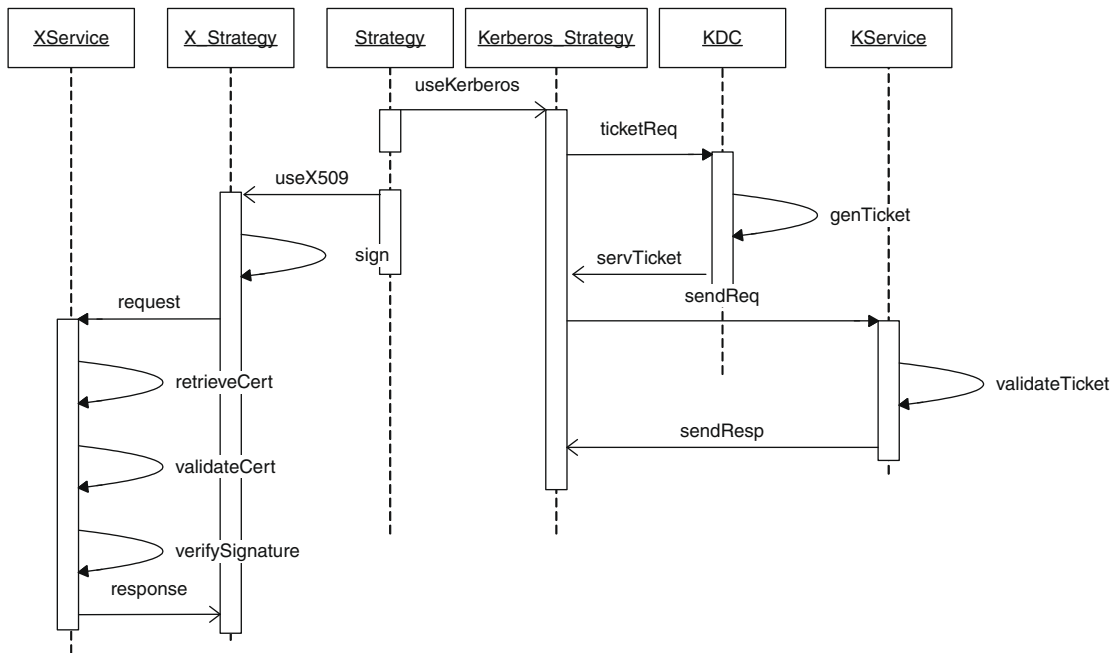


Fig. 22. Composition of the brokered authentication patterns.

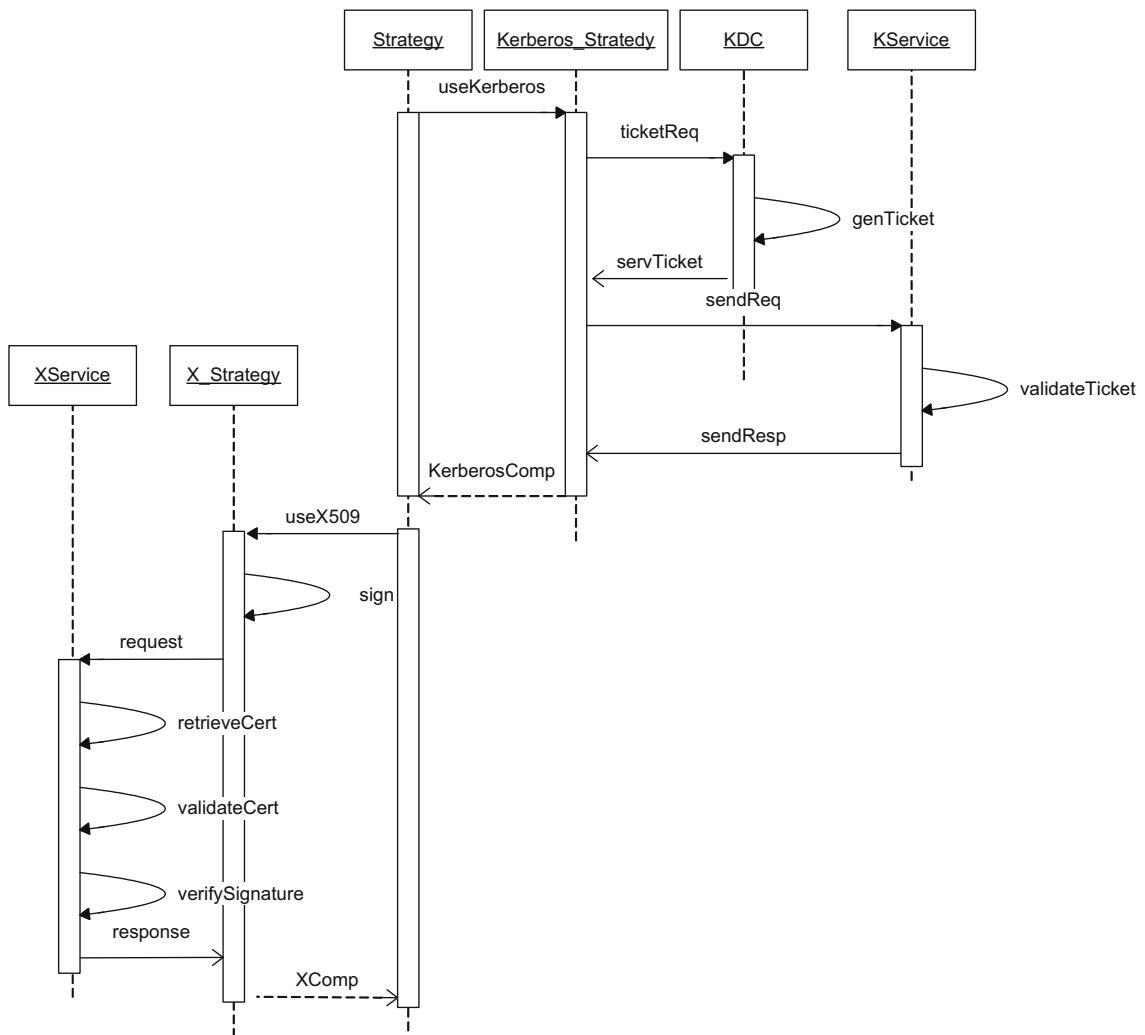


Fig. 23. Composition of the brokered authentication patterns (revised).

retrieveCert and validateCert actions by attackers. After studying the sequence diagram of the system design, we discover that this is a very subtle error because the cause of the error appears in different part of the sequence diagram from the effect of the error. As described previously, the effect of the error is that the retrieveCert and validateCert actions may be interrupted by an attacker. The cause of the error is actually that the messages (useKerberos, useX509) of Strategy were designed to be asynchronous messages, rather than synchronous messages, in the sequence diagram as shown in Fig. 22. Therefore, the Strategy can call the Kerberos authentication by sending out useKerberos while process XService is being used for X509 authentication. This flaw makes process XService functioning incorrectly by breaking the atomicity of the retrieveCert and validateCert actions. In other words, the X509 authentication interferes with the Kerberos authentication in this design. To correct the error and improve this design, we change these messages to be synchronous messages. This assures that whenever one authentication process is started, another authentication cannot be applied unless current authentication process is completed. The revised system design is shown in Fig. 23.

The CCS representation of the revised system design is shown in “testwss.ccs” as Multi_Authentication1 in the Appendix. The model checking result shows that `x_integrity` is satisfied by Multi_Authentication1 in the revised design, which is shown as follows:

```

cwb-n> chk -L gctl Multi_Authentication1 x_integrity
Generating ABTA from GCTL* formula...done
Initial ABTA has 9 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 6 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 25 times.
Stored 16 dependencies.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.016,0.000,0.000,
0.016)

```

In this case study, our experience shows that model checking can help analyzing security pattern compositions and finding security problems that are otherwise difficult for human inspection. Even for relatively simple sequence diagrams, it is hard for human to discover some subtle security errors by visual inspection, especially when the cause and effect of an error are apart from each other in a sequence diagram. In the previous case when checking the atomicity property of authentication, we could observe that the retrieveCert and validateCert actions are on the same time line of Xservice in the sequence diagram such that they are likely to be executed in sequence without any interruption. However, the model checking result shows that this visual observation is wrong. Therefore model checking helps us to discover this hidden design problem that is hard to find for manual inspection.

4. Related work

Existing work on security patterns is typically on the introduction of new security patterns or the applications of security patterns in different systems. To the best of our knowledge, there is no work on the analysis of security pattern compositions.

Rosado et al. provided a template to describe security patterns and a framework to compare them in [31]. Several security patterns are presented in their template. They also measure the security degree of the patterns and indicate the level that each pattern fulfills the properties and attributes common to all security patterns. No discussion of the security pattern compositions is presented.

To develop secure and efficient distributed systems, Yoshioka et al. [39] propose a method that uses patterns with security taken into account. They model the performance data associated with each pattern and provide the selection procedures for the detailed specifications. Their approach includes a pattern library that consists of several security patterns for inter-company coordination systems. Their patterns are applied in the environmentally conscious product design support system. However, their work does not formally analyze the composition of the security patterns.

Kienzle et al. [24] introduce 29 security patterns and classify them into either structural or procedural patterns. Structural patterns can be implemented in an application whereas procedural patterns aim at improving the development process of security-critical software systems. Brown et al. [4] propose authenticator, a security pattern, to describe a general mechanism for providing identification and authentication to a server from a client. Fernandez et al. [18] present the security patterns that describe mechanisms for designing secure Voice over IP (VoIP) systems. Their security patterns are specific for the VoIP application domain. While these works introduce a number of security patterns, the composition analysis of security patterns is not in their scopes.

Several formal approaches on the specification and verification of design patterns have been presented in [2,27,1,6,9,32,15,36,10,34,11,12]. These works are for general design patterns. They do not concentrate on the patterns with security concerns and properties.

The structural and behavioral aspects of design patterns in terms of responsibilities and rewards are formally specified in [34]. 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 [27]. 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 (LO-TOS) is proposed in [32]. In particular, the behavioral aspect of the Command and Composite patterns and their combination is specified.

Property patterns [14] have been proposed to provide taxonomy of properties written in LTL, CTL and QRE. Each property pattern is defined in terms of its scope, which is the extent of the program execution over which the pattern should hold. Property patterns define general properties whereas we focus on security-related properties.

A UML profile (UMLSec) on security has been introduced in [23], which can help to model security-related concerns in UML

diagrams. Formal semantics have also been provided for a subset of each kind of UML diagrams. In particular, the behavioral semantics of a simplified fragment of UML sequence diagram was presented. However, it cannot specify alternative flows, which is common in sequence diagram. It can only specify sequential behaviors. Model checking has been applied to check the security properties, such as secrecy, integrity, authenticity, and refinement, against the UML-Sec specifications. The application of a security pattern was also discussed. In contrast, our work focuses on security pattern compositions rather than just the application of a single security pattern. In addition, we analyze the compositions of security patterns with model checking techniques, instead of general security properties checking in UMLSec specifications. Furthermore, our behavioral semantics of sequence diagram can specify alternative flows.

A model checking framework has been presented for specifying and verifying Web service compositions in BPEL4WS. The Web service specification is annotated with timing constraints, including time points, duration, and interval constraints. It is then transformed into an extended TPPN (Timed Predicate Petri-Net) for model checking. While the paper concentrates on the specification of the Web service model, it does not provide any concrete verification using any model checker.

The interactions of composite web services have been modeled as conversations that specify the global sequence of messages exchanged by the web services in [21]. More specifically, the BPEL specifications of web services are translated to an intermediate representation that is later translated to the target verification language, Promela, the input language of the model checker SPIN. The concept of synchronizability is proposed to allow the verification of large class of composite web services with unbounded input queues using a finite state model checker. In contrast, our analysis framework concentrates on model checking security properties in web services.

A model-based approach to verifying Web service compositions has been proposed in [20]. The design can be specified in the message sequence charts (MSC), and mechanically compiled into the finite state process notation (FSP). The implementations are also translated for verifying trace equivalence. The behavior of implementation model can be verified against the specification models and assigns semantics so as to confirm expected results for both the designer and implementer. In particular, the semantic representation is added to the BPEL4WS notation. A tool has been identified for verification of implementations against abstract functional specifications. While their approach focuses on providing a general solution for verifying web service workflow specifications, the security properties are not concerned.

5. Conclusions

Security patterns have been adopted in the software industry to reuse expert design experience on solving security problems. While each security pattern presents a good solution to a security problem, there lacks work on analyzing their compositions that may have inconsistencies and interactions. Failure of detecting such errors and problems may result in security holes that suffer malicious attacks. Analysis at a high level (architecture and design) may greatly save time and effort than that at the lower implementation level.

In this paper, we provide an approach to analyzing the compositions of security patterns using model checking techniques based on our initial work [13]. In our approach, we presented the guideline to specify the behavior of security patterns in the model spec-

ification language. We define the synchronous message, asynchronous message, and alternative flows of a UML sequence diagram and transform them into CCS specifications. We provide a proof on the faithfulness of our specification with respect to the behavioral model of the security pattern in a sequence diagram. We also conducted two case studies to illustrate our approach by model checking the properties of security pattern compositions and show the detection of several security problems. Our results showed that our approach can find several subtle errors in the compositions.

Our approach consists of automatic process (model checking process) and formalism (CCS specification) whose preciseness is based on mechanism and mathematical derivation. Hence the verification result is rigorous and trustworthy. This makes our approach suitable for checking security problems in software design and provides security assurance. Moreover, since model checking process is largely automated, it reduces the potential errors causing by manual verification. We plan to develop an integrated tool to further automate the transformation of system behavioral specifications from a UML sequence diagram to CCS. This tool may take a sequence diagram as an input and perform the transformation of the synchronous message, asynchronous message, and alternative flows of a UML sequence diagram into CCS specifications following our method described in Rule 1. The output of this tool, CCS specifications of system behavior, can be passed into the model checker for behavior analysis.

Meanwhile, a common limitation of model checking technique is scalability due to the exponential complexity. Based on model checking techniques, our approach may be less effective if the software design is too complex, that is, the software system contains more states than a model checker can exhaust (usually called state explosion). In this case, we can reduce the complexity of the system by splitting it into several sub-systems and apply our approach on these sub-systems. However, the interaction of these sub-systems needs to be checked truthfully as a tradeoff.

Our approach is not tied to a particular model checker. Instead, it is also applicable to other CCS-based model checker, such as XMC as shown in [13] where we show the first case study on the analysis of secure observer using XMC model checker. We plan to apply other model checkers, e.g., SPIN or Brutus [7], to verify system security. Different model checkers present different levels of system abstractions. For instance, SPIN needs implementation level system specifications, while CWB-NC needs design level system specifications. In addition, we plan to investigate any case study that includes the security patterns and properties with loop so that we can extend our approach to consider the transformation of loop in sequence diagrams into CCS.

Appendix A

```
*int.ccs*

set
Internals={splogin,spcreate,sprequest,spforward,
spconfirm,splogout,spdestroy, aeidentity, aepassword,
aeauthresult,osetstate,ouupdate}
*integration of Observer,Securepipe and Authentica-
tion pattern*
proc INT=(SUB|SPPIPE|AEMAIN|OBS)\Internals
proc SPPIPE=spcreate.SPPIPEO=
```

```

srequest.spencrypt.'spforward.SPPIPE1+
splogout.spclosing.'spdestroy.SPPIPE1
proc SPPIPE1=SPPIPE0
proc AEMAIN=aesendinfo.verifying.'aeanswer.AEMAIN1
proc AEMAIN1=AEMAIN
proc OBS=OBSERVER+SPCLIENT+AECLIENT2
*spclient process in securepipe pattern*
proc SPCLIENT=
  spstart.'splogin.
SPCLIENT1+spnegotiate.'srequest.SPCLIENT1+
  spconfirm.spfinishing.'splogout.SPCLIENT1
proc SPCLIENT1=SPCLIENT
*client2 process in authentication pattern*
proc AECLIENT2 = astart.aestartup. 'aidentity.
'aepassword.AECLIENT21
+aeauthresult.'aeauthen_finish.AECLIENT21
proc AECLIENT21=AECLIENT2
*observer process in observer pattern*
proc OOBERVER= ochange.'osetstate.OOBERVER1+
ouupdate.ogetstate.OOBERVER1
proc OOBERVER1=OOBERVER
proc SUB=OSUBJECT+SPAPP+AECLIENT1
*subject process in observer pattern*
proc OSUBJECT=osetstate.ochange.'ouupdate.
OSUBJECT1
proc OSUBJECT1=OSUBJECT
*application process in secure pipe pattern*
proc SPAPP=
  splogin.'spcreat.SPAPP1+
  spforward.spprocessing.'spconfirm.SPAPP1+
  spdestroy.spdestroying.SPAPP1
proc SPAPP1=SPAPP
*client1 process in authentication pattern
proc AECLIENT1 =
  astart.'aesendinfo.AECLIENT11
  + aeanswer.'aidentity.AECLIENT11
  +'acommunicate.AECLIENT11
proc AECLIENT11=AECLIENT1

*int.gctl*

* there doesn't exist a path where no securepipe
established before authentication start *
prop safe_auth_guarantee=
not (E({'spfinishing'}U{aestart}))
prop safe_observer1=
E F{aeauthen_finish}-> (E F {'ochange'})
prop safe_observer2=
E F{aeauthen_finish}-> (E F {'ochange'})
prop safe_observer3=
E F{aeauthen_finish}-> (E F {'ogetstate'})
prop safe_observer =
safe_observer1/\safe_observer1 /\safe_observer1
prop message_encrypted1=
(E F{'spfinishing'}-> (E F {'change'}))/\safe_
observer1
prop message_encrypted2=
(E F{'spfinishing'}-> (E F {'change'}))/\safe_
observer2
prop message_encrypted3=
(E F{'spfinishing'}-> (E F {'getstate'}))/\safe_
observer3
prop message_encrypted=
message_encrypted1/\message_encrypted2/\message_
encrypted3

*intl.gctl*

* there doesn't exist a path where no securepipe
established before authentication start *
prop safe_spa_guarantee=
not E({'spa_confirm'}U{aestart})
prop safe_spb_guarantee=
not E({'spb_confirm'}U{aestart})
*any message from subject must be sent after it has
been authenticated*
prop safe_send_setstate=
not E({'aeanswer'}U{osetstate})
*after authentication, message sent from subject to
observer should always be secure*
prop secure_send_setstate=
safe_send_setstate/\(not E({'spencrypt'}->{osetstate}))

*intl.ccs*

*improvement of INT*
set
Internals={splogin,spcreate,srequest,spforward,
spconfirm,splogout, spdestroy, aidentity, aepas-
sword, aeauthresult,osetstate,ouupdate}
*integration of Observer,Securepipe and Authentica-
tion pattern*
proc INT1=(SUB)SPPIPE[AUTH][OBS][SPPIPEA|
SPPIPEB]\Internals
proc SPPIPE=spcreate.SPPIPE0
proc SPPIPE0=
  srequest.spencrypt.'spforward.SPPIPE1+
  splogout.spclosing.'spdestroy.SPPIPE1
proc SPPIPE1=SPPIPE0
proc SPPIPEA=spa_create.SO
proc SO=SSPPPEAO.AECLIENT1.SPPIPEA1
proc SPPIPEAO=
  spa_request.spa_encrypt.'spa_forward.nil+
  spa_logout.spa_closing.'spa_destroy.nil
proc AECLIENT1=
  astart.'aesendinfo.aeanswer.'aestart.nil
proc SPPIPEA1=SO
proc SPPIPEB=spb_create.TO
proc TO=SPPIPEBO.AECLIENT2.SPPIPEB1
proc SPPIPEBO=
  spb_request.spb_encrypt.'spb_forward.nil+
  spb_logout.spb_closing.'spb_destroy.nil
proc AECLIENT2 =
  astart.'aesendinfo.aeanswer.nil
proc SPPIPEB1=TO
proc AUTH=AEMAIN+SPA_APP+SPB_APP
proc AEMAIN=aesendinfo.verifying.'aeanswer.AEMAIN1
proc AEMAIN1=AEMAIN
*SPA_APP is the application process in secure pipe
pattern*
proc SPA_APP=
  spa_login.'spa_creat.SPA_APP1+
  spa_forward.spa_processing.'spa_confirm.SPA_APP1+
  spa_destroy.spa_destroying.SPA_APP1
proc SPA_APP1=SPA_APP
*SPB_APP is the application process in secure pipe
pattern*
proc SPB_APP=
  spb_login.'spb_creat.SPA_APP1+
  spb_forward.spb_processing.'spb_confirm.SPB_APP1+
  spb_destroy.spb_destroying.SPB_APP1

```

```

proc SPB_APP1=SPB_APP
proc OBS=OBSERVER+SPCLIENT+SPB_CLIENT
*spclient process in securepipe pattern*
proc SPCLIENT=
  spstart.' splogin.
SPCLIENT1+spnegotiate.' sprequest.SPCLIENT1+
  sponconfirm.spfinishing.' splogout.SPCLIENT1
proc SPCLIENT1=SPCLIENT
*observer process in observer pattern*
proc OOBSERVER= ochange.' osetstate.OOBSERVER1+
  oupdate.ogetstate.OOBSERVER1
proc OOBSERVER1=OOBSERVER
*spclient process in securepipe pattern B*
proc SPB_CLIENT=
  spb_start.' spb_login.SPB_CLIENT1+spb_negoti-
  ate.' spb_request. SPB_CLIENT1+
  spb_confirm.spb_finishing.' spb_logout.SPB_
  CLIENT1
proc SPB_CLIENT1=SPB_CLIENT
proc SUB=OSUBJECT+SPAPP+SPA_CLIENT
*subject process in observer pattern*
proc OSUBJECT=osetstate.ochange.' oupdate.
  OSUBJECT1
proc OSUBJECT1=OSUBJECT
*application process in secure pipe pattern*
proc SPAPP=
  splogin.' spcreat.SPAPP1+
  spforward.spprocessing.' spconfirm.SPAPP1+
  spdestroy.spdestroying.SPAPP1
proc SPAPP1=SPAPP
*spclient process in securepipe pattern A*
proc SPA_CLIENT=
  spa_start.' spa_login.SPA_CLIENT1+
  spa_negotiate.' spa_request. SPA_CLIENT1+
  spa_confirm.spa_finishing.' spa_logout.SPA_
  CLIENT1
proc SPA_CLIENT1=SPA_CLIENT

*testwss.ccs*

proc KDC=ticketReq.genTicket.' servTicket.KDC
proc KService=sendReq.validateTicket.' sendResp.
  KService
proc X_Strategy=use_X509.sign.' request.response.
  Kerberos_Strategy
proc XService=
  request.retrieveCert.validateCert.
  verifySignature.' response.XService
proc testMauth= (Strategy|Kerberos_Strategy|X_
  Strategy)\{use_kerberos, use_X509}
set L={use_kerberos,use_X509,request,response,tick-
  etReq,servTicket, sendReq,sendResp}
proc Multi_Authentication= (KService|KDC|Kerberos_
  Strategy|Strategy|X_Strategy|XService)\L
*improved multi-authentications
proc Strategy1='use_kerberos.kcomp.'use_X509.xcomp.
  nil
proc Kerberos_Strategy1=use_kerberos.' ticketReq.
  servTicket.' sendReq.sendResp.kcomp.Kerberos_
  Strategy1
proc X_Strategy1=use_X509.sign.' request.response.'
  xcomp.X_Strategy1
proc testMauth1= (Strategy1|Kerberos_Strategy1|X_
  Strategy1)\{use_kerbero s,use_X509,kcomp,xcomp}

```

```

set Ll={use_kerberos,use_X509,request,response,
  ticketReq,servTicket, sendReq,sendResp,kcomp,xcomp}
proc Multi_Authentication1= (KService|KDC|Kerber-
  os_Strategy1|Strategy1|X_Strategy1|XService)\Ll

*testwss.gctl*

*liveness property*
*authentication will eventually happen*
prop auth_happen= EF ({validateTicket})\/{validateCert})
*integrity property*
*no interruption between retrieveCert andValidate-
  Cert*
prop x_integrity=AG({retrieveCert}->X{validateCert})

```

References

- [1] Paulo Alencar, Donald Cowan, Jing Dong, Carlos Lucena, A pattern-based approach to structural design composition, in: Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference, Phoenix USA, October 1999, pp. 160–165.
- [2] Paulo Alencar, Donald Cowan, Carlos Lucena, A formal approach to architectural design patterns, in: Proceedings of the Third International Symposium of Formal Methods Europe (FME), 1996, pp. 576–594.
- [3] Geoff Barrett, Model checking in practice: the T9000 virtual channel processor, IEEE Transactions on Software Engineering 21 (2) (1998) 69–78.
- [4] F.L. Brown, E.B. Fernandez, The authenticator pattern, in: Proceedings of the Pattern Languages of Programs (PLoP'99), 1999.
- [5] M.C. Browne, Edmund M. Clarke, D.L. Dill, Automatic verification of sequential circuits using temporal logic, IEEE Transactions on Computer C-35 (12) (1986) 1035–1044.
- [6] S. Chinnasamy, R.R. Raje, Z. Liu, Specification of design patterns: an analysis, in: Proceedings of the Seventh International Conference on Advanced Computing and Communications, 1999, pp. 300–304.
- [7] E.M. Clarke, S. Jha, W. Marrero, Verifying security protocols with Brutus, ACM Transactions on Software Engineering and Methodology (TOSEM) 9 (4) (2000).
- [8] Guilan Dai, Xiaoying Bai, Chongchong Zhao, A framework for model checking web service compositions based on BPEL4WS, in: Proceedings of the IEEE International Conference on e-Business Engineering, 2007.
- [9] Jing Dong, Paulo Alencar, Donald Cowan, Ensuring structure and behavior correctness in design composition, in: Proceedings of the Seventh Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS), Edinburgh UK, 2000, pp. 279–287.
- [10] Jing Dong, Paulo Alencar, Donald Cowan, A behavioral analysis and verification approach to pattern-based design composition, Software and Systems Modeling, vol. 3, Springer Verlag, 2004, pp. 262–272.
- [11] Jing Dong, Paulo Alencar, Donald Cowan, Automating the analysis of design component contracts, Software – Practice and Experience (SPE), Vol. 36, Wiley, 2006, 27–71.
- [12] Jing Dong, Tu Peng, Zongyan Qiu, Commutability of design pattern instantiation and integration, in: Proceedings of the First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE), China, June 2007.
- [13] Jing Dong, Tu Peng, Yajing Zhao, Model checking security pattern compositions, in: Proceedings of the Seventh International Conference on Quality Software (QSIC), Portland, Oregon, USA, October 2007, pp. 80–89.
- [14] Matthew B. Dwyer, George S. Avrunin, James C. Corbett, Patterns in property specifications for finite-state verification, in: Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, May 1999.
- [15] A.H. Eden, Y. Hirshfeld, Principles in formal specification of object-oriented architectures, in: Proceedings of the 11th CASCON, Toronto, Canada, November 2001.
- [16] E. Emerson, Edmund M. Clarke, Using branching time temporal logic to synthesize synchronization skeletons, Science of Computer Programming 2 (1982) 241–266.
- [17] E.A. Emerson, J.Y. Halpern, 'Sometime' and 'not never' revisited: on branching versus linear time temporal logic, Journal of the Association for Computing Machinery 33 (1) (1986) 151–178.
- [18] E.B. Fernandez, J.C. Pelaez, M.M. Larrondo-Petrie, Security patterns for voice over IP networks, in: Proceedings of the International Multi-Conference on Computing in the Global Information Technology (ICCGI'07), 2007.
- [19] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [20] H. Foster, S. Uchitel, J. Magee, J. Kramer, Model-based verification of web service compositions, in: Proceedings of the 18th IEEE International Conference on Automated Software Engineering Conference (ASE), 2003.

- [21] Xiang Fu, Tevfik Bultan, Jianwen Su, Analysis of interacting BPEL web services, Proceedings of the 13th International Conference on World Wide Web, New York, USA, 2004, pp. 621–630.
- [22] Gerard J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
- [23] Jan Jürjens, Secure Systems Development with UML, Springer-Verlag, Heidelberg, 2004.
- [24] D.M. Kienzle, M.C. Elder, D. Tyree, J. Edwards-Hewitt, Security Patterns Repository, 2002.
- [25] M.L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, PhD Thesis, Carnegie Mellon University, CMU-CS-92-131, 1992.
- [26] Bertrand Meyer, Applying “design by contract”, IEEE Computer (1992) 40–51.
- [27] Tommi Mikkonen, Formalizing design pattern, in: Proceedings of the 20th International Conference on Software Engineering, 1998, pp. 115–124.
- [28] Robin Milner, Communication and Concurrency. International Series in Computer Science, Prentice Hall, 1989.
- [29] A. Pnueli, E. Harel, Applications of Temporal Logic to the Specification of Real-time Systems, Springer, 1988.
- [30] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, D.S. Warren, Efficient model checking using tabled resolution, in: Proceedings of the Ninth International Conference on Computer Aided Verification (CAV), Haifa Israel, LNCS1243, Springer Verlag, July 1997, pp. 143–154.
- [31] D.G. Rosado, C. Gutierrez, E. Fernandez-Medina, M. Piattini, A study of security architectural patterns, in: Proceedings of the First International Conference on Availability, Reliability and Security (ARES'06), 2006.
- [32] Motoshi Saeki, Behavioral specification of GoF design patterns with LOTOS, in: Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC), December 2000, pp. 408–415.
- [33] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad, Security Patterns: Integrating Security and Systems Engineering, Wiley, 2006.
- [34] Neelam Soundarajan, Jason O. Hallstrom, Responsibilities and rewards: specifying design patterns, in: Proceedings of the 26th International Conference on Software Engineering, May 2004, pp. 666–675.
- [35] Christopher Steel, Ramesh Nagappan, Ray Lai, Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management, Prentice Hall PTR, 2005.
- [36] Toufik Taibi, David C.L. Ngo, Formal specification of design pattern combination using BPSL, Information and Software Technology (IST), vol. 45, Elsevier-Science, 2003, pp. 157–170.
- [37] John Viega, Gary McGraw, Building Secure Software: How to Avoid Security Problems the Right Way, Addison-Wesley, 2001.
- [38] Jeannette M. Wing, Mandana Vaziri-Farahani, A case study in model checking software systems, Science of Computer Programming 28 (1996) 273–299.
- [39] N. Yoshioka, S. Honiden, A. Finkelstein, Security patterns: a method for constructing secure and efficiency inter-company coordination systems, in: Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference (EDOC'04), 2004.
- [40] <http://www.cs.sunysb.edu/~cwb/>.
- [41] <http://msdn.microsoft.com/en-us/library/aa480545.aspx>.
- [42] <http://msdn.microsoft.com/en-us/library/aa480560.aspx>.