Model-Driven Application and Validation of Security Patterns

YUKI SHIROMA, HIRONORI WASHIZAKI AND YOSHIAKI FUKAZAWA, Waseda University

ATSUTO KUBO Aoyama Media Laboratory

EDUARDO B FERNANDEZ Florida Atlantic University

NOBUKAZU YOSHIOKA National Institute of Informatics

The spread of software services through the Internet has increased the importance of software security. Security patterns is a tool that helps developers, architects and security specialists utilize security experts' knowledge and standardize how they respond to security threats. Security patterns contain recurring solutions about security problems. However, there is a possibility that developers may apply security patterns in inappropriate ways due to their lack of knowledge about dependencies among patterns. We propose an automated technique of applying security patterns in model-driven software development by defining model transformation rules that take into consideration pattern dependencies. Our technique prevents inappropriate applications such as applying security patterns to the wrong model elements or in the wrong order. We believe that our technique can help developers to apply security patterns to their own models automatically in appropriate ways.

General Terms: Security Pattern

Additional Key Words and Phrases: Model Driven Development, Unified Model Language, ATLAS Transformation Language

ACM Reference Format:

Pineo, D. and Ware, C. 2010. Neural Modeling of Flow Rendering Effectiveness. ACM Trans. Appl. Percept. 2, 3, Article 1 (May 2010), 10 pages.

DOI=10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

This work is supported in part by the Mizuho Foundation for the Promotion of Sciences and the JGC-S SCHOLARSHIP FOUNDATION.

Author's address: Y. Shiroma, H. Washizaki, Y. Fukazawa, 3-4-1 Okubo, Shinjuku-ku, Tokyo Japan; email: shiroma1986@moegi.waseda.jp washizaki@waseda.jp Fukazawa@waseda.jp; A. Kubo, Room 716. Win Aoyama 2-2-15, Minami-Aoyama Minato-ku, Tokyo, Japan; email: kubo@nii.ac.jp Author's address: E. B. Fernandez, 777 Glades Road SE, Room 300, Boca Raton, FL 33431; email: ed@cse.fau.edu; N. Yoshioka, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan; email: nobukazu@nii.ac.jp

1. INTRODUCTION

The spread of software services through the Internet has highlighted the growing importance of software security. It is imperative to improve security because security is at risk in every phase of development but how to do so is not straightforward. Moreover, even when security has been achieved, it is necessary to consider the trade-off with other quality characteristics, and a lot of development experience is required to make appropriate judgments. Security patterns have been proposed to assist developers in handling security concerns. By using security patterns, software developers can utilize security specialists' knowledge.

Security patterns have mutual dependencies, and developers should decide the sequences of the pattern application by considering such dependencies. If a dependency is not considered, there's a chance that the affected security patterns will be incorrectly applied and the security of the entire system would suffer. The "See Also" and "Related Pattern" sections in [2] describe information of dependencies but it is not enough for users to know how to apply security patterns in the correct sequence and place. However, as far as we know, no application support technique that considers dependencies between patterns has been proposed.

To remedy this situation, we propose a security pattern application technique that considers dependencies between patterns. Patterns are applied by making model transformations. Our approach leaves a mark whenever a security pattern is applied in the model, and subsequent security patterns are applied at the previously left marks. This enables consecutive applications of security patterns. Also "weaving" in area of Aspect Oriented Programming can considers dependencies security patterns but it is not as widely used as model transformation. In addition, we think that our approach is adaptable to other patterns that have dependencies among them.

2. CORTICAL PROCESSING OF CONTOURS

This section describes the ideas behind model-driven development and security patterns. In addition, it discusses the concept of dependencies between security patterns.

2.1 Model Driven Development(MDD)

Model driven development is a methodology that builds software around a model[7]. The developers translate an abstract model into a more concrete model. The developer can obtain the source code semi-automatically by making repeated transformations to arrive at a more concrete model. Moreover, there are various model transformations, e.g., model merging and model marking.

2.2 Security Pattern

A pattern is a proven solution to a problem, described in context. Security patterns are proposed in [2,13,15] and are described in terms of a structure, context, problem, solution, and consequences [2]. The solution is based on proven experience. So it is something that has been tried, tested and refined over a number of solutions at a number of different organizations. The advantage of security patterns is that pattern users can utilize the knowledge of security specialists. Security patterns provide guidelines for improving confidentiality, integrity, and availability in software development.

Security patterns have dependencies between each other and when developers apply security patterns consecutively, they should be applied consistently throughout the development process. Thus, it is very important to consider the dependencies in the entire development process. Figure 1 shows an example of dependencies between security patterns. For instance, “Authenticator <- Authorization” means that the Authenticator pattern should be applied before the Authorization pattern. The sequences of possible applications are as follows.

- Authenticator, Authorization, Reference Monitor
- Authenticator, RBAC, Reference Monitor

Notice that four problems can occur.

P1. The security pattern may be incorrectly applied if the dependencies among patterns are not considered.

P2. The security pattern may be applied to the wrong part of the model.

P3. Time and labor may be prohibitively large.

P4. Later applications may cause the applied security patterns to lose their properties (i.e. One of properties of the Authorization pattern is that there is no relation between the Subject role and the Protection Object role).

The known supports for developers include only classification [3] and unit security pattern application support [5], and these have trouble dealing with the above problems.

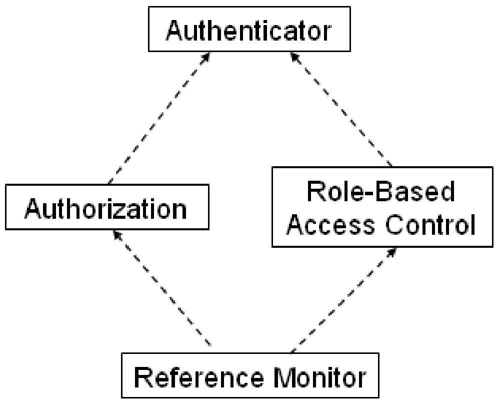


Fig. 1. Example of dependencies among security patterns

3. APPROACH

This section describes the method of making transformation rules, the use of two our system, and the solution to the above problems. There are four solutions to the above problems.

S1. Automation of model transformation

S2. Making a security pattern transformation rule library

S3. Leaving marks about the result of a pattern's applications in the class, and referring to them to apply subsequent security patterns.

S4. Automatically detecting the applied security pattern in the developing UML model and generating validation code from existing transformation rules of security patterns.

S1 is the solution of P2 and P3. Automation of most of an application by model transformation mitigates the possibility of its application to an incorrect class. S2 is the solution of P3. S2 mitigates P3 because software developers can utilize the transformation rules at any time after security specialists describe them. S3 is the solution of P1 and P2. Our technique leaves a UML stereotype describing the applied pattern name and role name as a mark in the class corresponding to role of the applied pattern. Thus, when A depends on B, if any marks of B don't exist, our technique prevents B from being applied to the developing model. S3 thus mitigates P1. Moreover, we prevent security pattern from being applied to the incorrect class, because we leave a stereotype homologize, the role of the security pattern, the class in the developing model. So S3 mitigates P2.

3.1 Method of Describing Transformation Rules

A security specialist describes transformation rules derivable from a security pattern by using ATL (ATLAS Transformation Language) [8]. The transformation rules each consist of a pre-condition, an argument, and an operation.

A pre-condition is an assumption of the security pattern application. The definition of a pre-condition varies according to the presence of an existing pattern upon which the to-be-applied security pattern depends. If there is no pattern on which the applied security patterns depend, security specialists define which part corresponds to the role of the security pattern in the model. If such a pattern exists, security specialists place a mark in the model that indicates that the existing pattern is a pre-condition for applying the new pattern. Our technique supports consecutive applications of security patterns that are dependent on other patterns by defining the output of the previous model transformation as a pre-condition in the subsequent model transformation. Security specialists should determine the dependencies among patterns by referring to the "Related patterns" section in the security pattern catalog. Moreover, there is a possibility that the dependency could be ascertained by looking at the problem and the context of the applied pattern, and also by looking at the context, solution, and consequence sections of all the other patterns.

For instance, it is described that "The authenticated user, represented by processes running on its behalf, is then allowed to access resources according to their rights" (p. 323) in the context of the Authenticator pattern in [1]. On the other hand, it is described that "Any environment in which we have resources whose access needs to be controlled." (p. 245) in the context of the Authorization pattern. It can be judged that a relation exists between these two patterns because their context sections resemble each other with regard to controlling access to the protected property, even though this relationship is not explicitly written in either pattern. Moreover, the description "What the attested user accesses the protection property by the authority is permitted" shows that we should authorize after authenticating. We can see that the Authenticator pattern should be applied before the Authorization pattern.

The argument is a parameter that the software developer should input (i.e., the name of the class that corresponds to the role of the applied security pattern).

The operation maps a set of classes and relations among classes in the model when the pattern is applied. Security specialists should look for common roles between the applied pattern and to-be-applied pattern in their structural description.

For instance, the Subject role of the Authenticator pattern is described in [1] as, "A Subject, typically a user, and requests access to system resources" (p. 324). Moreover, the Subject role of the Authorization pattern is described in [1] as, "The Subject class describes an active entity that attempts to access a resource (Protection Object) in some way" (p. 246). The Subject roles of the two security patterns are identical because the descriptions of the Subject role are very similar. In a word, when there is the dependency between the two patterns, a common role is the basis for deriving the transformation rule of each pattern. Therefore, security specialists should describe a transformation rule whereby the Subject role of the Authenticator pattern corresponds to the Subject role of the Authorization pattern. A security pattern is applied by transforming the model by using the above-mentioned transformation rule.

3.2 Example Description of Transformation Rules

What follows is an example of describing transformation rules when applying the Authorization pattern.

The pre-condition is that the class with the stereotype 'Authenticator.Subject' exists. This stereotype indicates where the Authenticator pattern, on which the Authorization pattern depends, is applied to the model.

The argument is a class name that corresponds to the Protection Object role of the Authorization pattern.

The operation has five steps.

- Add the stereotype ‘Authorization.ProtectionObject’ to the Class that corresponds to the Protection Object role that the developer inputs as an argument.

- Add the class that corresponds to the Right role.
- Add the relation between the class that corresponds to the Subject role and the class that corresponds to the Right role.
- Add the relation between the class that corresponds to the Protection Object role and the class that corresponds to the Right role.
- Remove the relation between the class that corresponds to the Protection Object role and the class that corresponds to the Subject role.

Security specialists describe the transformation rules by using ATL. Figure 2 shows part of a transformation rule of the Authorization pattern described in ATL. The isProtOb function of the first line in Figure 2 judges whether the character string of the class name that corresponds to the Protection Object role that the software developer inputs corresponds to the class name in the model. The hasStereotype function of the fourth line judges whether the class in the model has the stereotype 'AuthenticatorSubject'.

Marking by stereotype was chosen as the form of the model transformation in order to show where a security pattern is applied. If there is common role between two patterns, the proposed system decides the application place with stereotype in developing model.

The reason for choosing ATL as the model transformation language is that it is easy for software developers to understand and it can easily be extended because it is based on Queries/Views/Transformations (QVT), which is a standard model transformation.

```
helper context UML!Class def:isProtOb() : Boolean =
if self.name = thisModule.ProtObName
    then true else false endif;
Helper context UML!Class def:hasStereotype(stereotype : String) : Boolean = self.stereotype ->
collect(s|s.name)->includes(stereotype)…
rule AuthorizationProtectionObjectClass {
    from s : UML!Class (s.isProtectionObject())…
        stereotype <- stereotypePO),
        stereotypePO : UML!Stereotype (
        name <- 'Authorization.ProtectionObject',   …
rule SubjectClass {
    from s : UML!Class (s.hasStereotype('Authenticator.Subject')) …
```

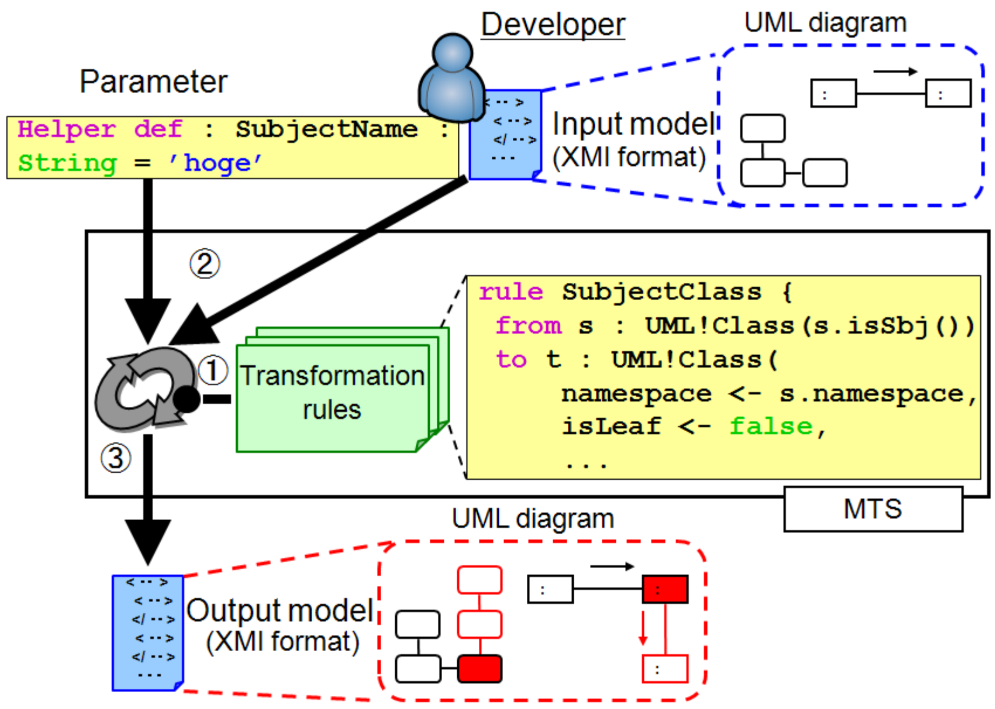
Fig. 2. Part of the transformation rule of the Authorization pattern

3.3 Application Procedure

The Model Transformation System (MTS) transforms a UML model in XMI format inputted by software developers, and the security pattern is applied by making a model transformation. Figure 3 illustrates the proposed system. The security pattern is applied as follows.

1. The developer selects the security pattern.
2. The developer inputs the model and the parameters to the proposed system.
3. The system transforms the model and outputs the model with the applied security pattern.

The system deals with two models as input and output: Class diagram and Communication diagram. These are described in XMI format. The transformation rules, once described, can be reused. Consecutive application of security patterns considering the dependencies among patterns becomes possible by using the obtained output model as the input model for the subsequent transformation. By using marks, it can be automatically judged whether the class and the pattern role are the same.



```
Helper def : SubjectName :
String = 'hoge'
```

```
rule SubjectClass {
 from s : UML!Class(s.isSbj())
 to t : UML!Class(
     namespace <- s.namespace,
     isLeaf <- false,
     ...
```

Fig. 3. Model Transformation System (MTS)

3.4 Validation

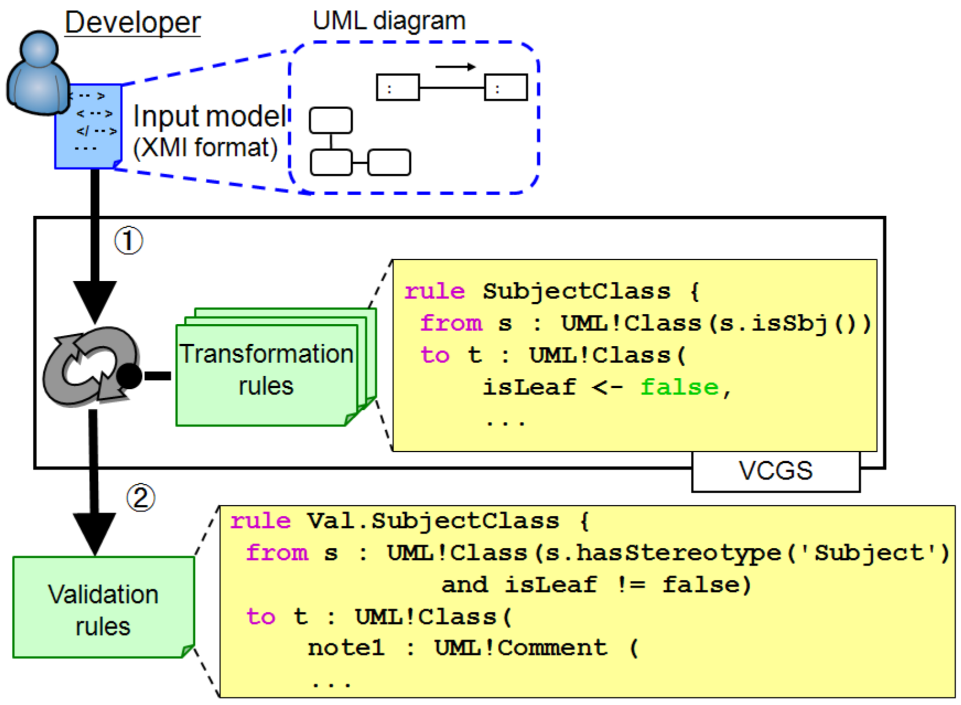
The Validation Code Generator System (VCGS) ensures that the developing model has the properties of the applied security pattern. VCGS shows validation results in a new UML note in the developing model. Figure 4 illustrates VCGS. The validation code is generated as follows.

1. The developer inputs the developing UML model in XMI format into VCGS.

2. VCGS detects security patterns that have been applied to the developing UML model from stereotypes in each class of the model.

3. VCGS generates validation code in ATL format from the transformation rules of the detected security patterns.

VCGS generates a validation code about all properties except “name” because “name” is not a property that security patterns should maintain but an identifier that is subject to change by the developer. Figure 5 shows the validation code generated from a transformation rule.



```
rule SubjectClass {
 from s : UML!Class(s.isSbj())
 to t : UML!Class(
     isLeaf <- false,
     ...
```

```
rule Val.SubjectClass {
 from s : UML!Class(s.hasStereotype('Subject')
             and isLeaf != false)
 to t : UML!Class(
     note1 : UML!Comment (
     ...
```

Fig. 4. Validation Code Generator System (VCGS)

```
rule ValidateAuthorizationProtectionObjectClass01 {
    from s : UML!Class (s.hasStereotype(Authorization.ProtectionObject) == false)
    to t :  UML!Class (
        comment1 : UML!Note (
        body <- “No Class corresponding to protection    object role of Authorization pattern.”)
}…
rule ValidateAuthorizationSubjectClass01 {
    from s : UML!Class (s.hasStereotype('Authenticator.Subject' == true and…
```

Fig. 5. Part of the transformation rule of the Authorization pattern

Figure 6 shows the entire validation procedure.

1. The developer inputs the validation code and developing models into MTS.

2. MTS does the model transformation.

If MTS does the model transformation after the validation code has been inputted, it outputs the model with an additional UML note describing the validation result.

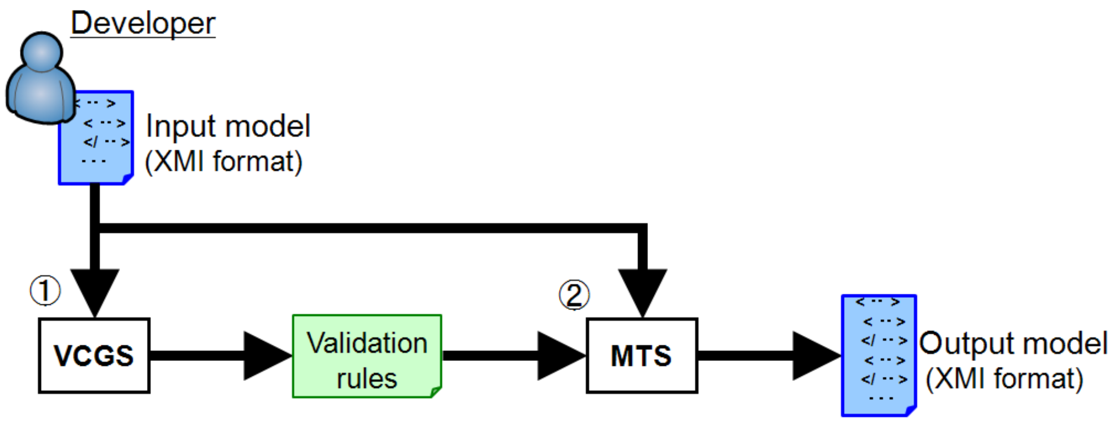


Fig. 6. Image of the Entire validation procedure

4. DISTRIBUTION OF SECURITY PATTERNS

Our technique can deal with security patterns that have a structural description. It can deal with 27 security patterns described in [1]. 19 security patterns in [1] cannot be dealt with because their structures are not described.

5. EXAMPLE APPLICATION

We shall consider a Patient Information Management System (PIMS) in a hospital as an example application. Figure 7 shows the use-case diagram of the PIMS. The PIMS has two security requirements. The PIMS faces two problems in regard to meeting the security requirements.

1. Only hospital employees can access the PIMS. Confidentiality is thus maintained.

2. The user of the PIMS can only do the use case with the allocated authority. Confidentiality is thus maintained.

The first problem is that there is no way to judge if the user is an employee or not. A third party could thus pose as an employee in order to steal patient information and sell it (misuse case 1). The second problem is that everyone related to the hospital has read and write access to the patient's information. Even if the first problem is solved, the second problem remains. A potential problem is that someone could illegally rewrite a patient’s examination results (misuse case 2). Figure 9 shows the class diagram of the PIMS, and Figure 8 shows part of the XMI of the class diagram. Two security patterns are applied as a solution to the above-mentioned threats. First, the Authenticator pattern is applied. Then, the Authorization pattern is applied.

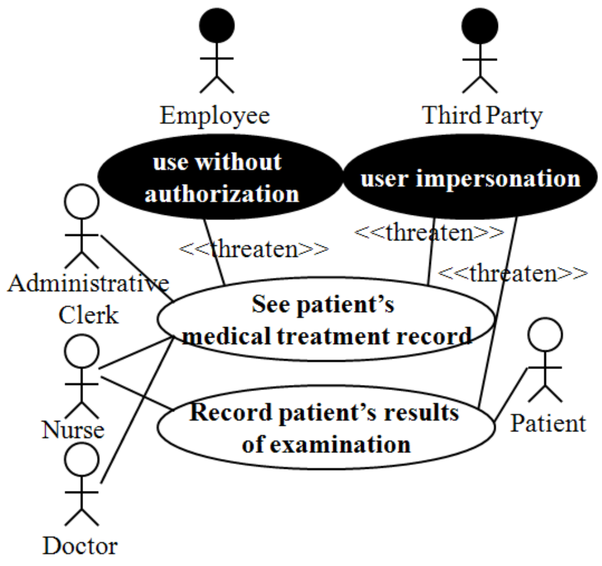


Fig. 7. Use-case diagram of the PIMS.

```
<UML:Class xmi.id = 'a2' name ='Employee' …
```

Fig. 8. Part of XMI in the class diagram

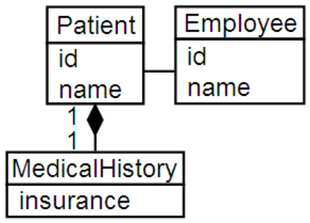


Fig. 9. Class diagram of the Patient Information Management System (PIMS)

5.1 Application of the Authenticator Pattern

After selecting the ATL file in which the Authenticator pattern is described, the developer inputs the model and the parameters to the system. If the developer inputs "sbjName ='Employee'", the system decides that the Employee class corresponds to the Subject role of the Authenticator pattern and applies the Authenticator pattern. A stereotype is added to indicate that the Employee class corresponds to the Subject role. Figure 10 shows the class diagram of the Authenticator pattern, and Figure 11 shows the class diagram after the Authenticator pattern has been applied. Figure 12 shows the XMI for the class diagram after the Authenticator pattern has been applied.

The system judges that the Employee class corresponds to the Subject role of the Authenticator pattern and applies the Authenticator pattern to the model because developers inputted the parameters of the Authenticator transformation rule "sbjName ='Employee'". At this time, a mark is applied to indicate that Employee class corresponds to the Subject role as a result of adding the stereotype to it.

The authentication structure added to the model as a result of applying the Authenticator pattern is a countermeasure against a malicious third party disguised as a user. However, the problem that a malicious employee can access patient information remains because every employee is granted access to the information. To combat this problem, an Authorization pattern is required.

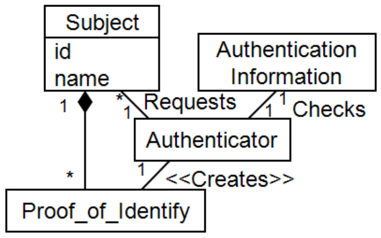


Fig. 10. Class diagram of the Authenticator pattern

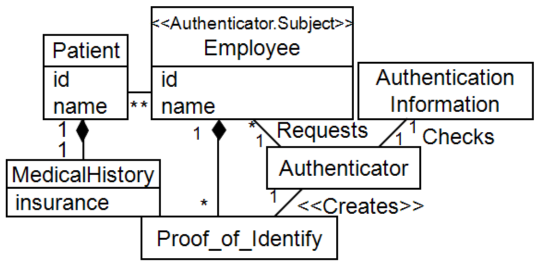


Fig. 11. Class diagram after the Authenticator pattern has been applied

```
<UML:Class xmi.id = 'a2' name ='Employee'   …
    <UML:Stereotype xmi.idref = 'a3'/>      …
        <UML:Stereotype xmi.id = 'a3' name = 'Authenticator.Subject'…
<UML:Class xmi.id = 'a10' name = 'Authenticator'…
```

Fig. 12. Part of XMI in the class diagram

5.2 Application of the Authorization Pattern

After selecting the ATL file in which the Authorization pattern is described, the developer inputs the model and the parameters to the system. If the developer inputs “protObName =‘Patient”, the system judges that the Patient class corresponds to the Protection Object role of the Authorization pattern and applies the Authorization pattern. Moreover, because the Employee class that corresponds to the Subject role applies the stereotype ‘Authenticator.Subject' when the Authenticator pattern was applied, the system judges that the Employee class corresponds to the Subject role of the Authenticator pattern and to the Subject role of the Authorization pattern.

A stereotype is added to indicate that the Patient class corresponds to the Protection Object role and to indicate that the Subject role corresponds to the Employee class of the Authorization pattern. Figure 13 shows the class diagram of the Authorization pattern, and Figure 14 shows the class diagram after the Authorization pattern has been applied.

The PIMS had two security problems in that there was no authentication and authorization structure. The countermeasure against user impersonation was taken by applying the Authenticator pattern, and the countermeasure against use without authorization was taken by applying the Authorization pattern.

Figure 15 shows the use case diagram after applying the security patterns. Figure 16 shows the process of two consecutive security pattern applications.

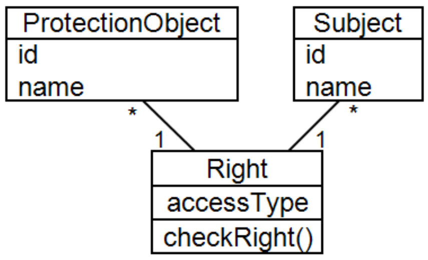


Fig. 13. Class diagram after the Authenticator pattern has been applied

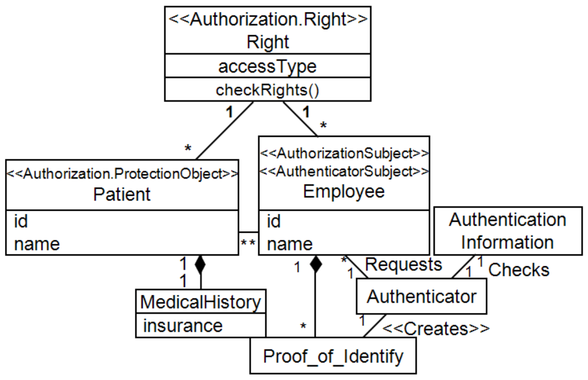


Fig. 14. Class diagram after applying the Authorization pattern

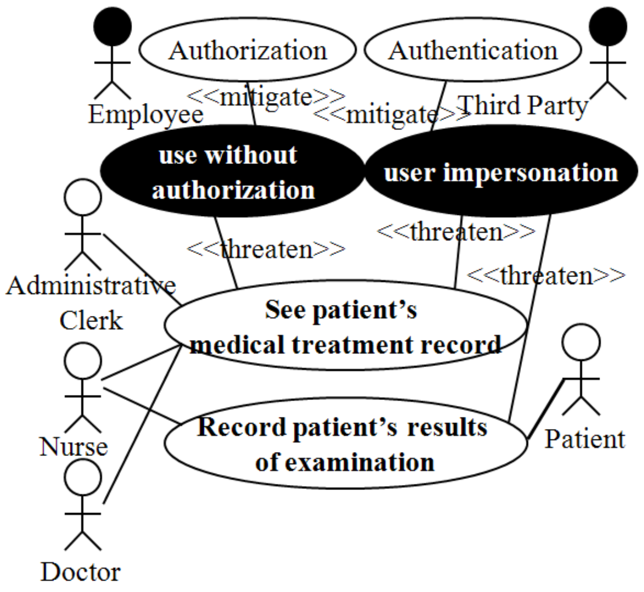


Fig. 15. Use-case diagram of PIMS after applying security patterns

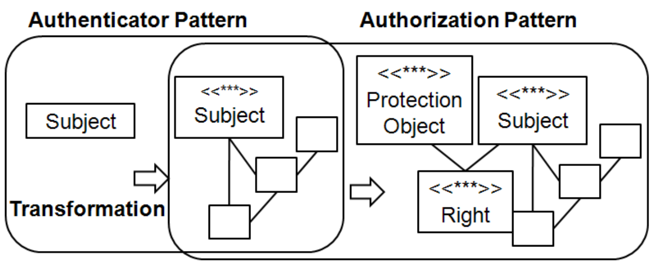


Fig. 16. Two consecutive security pattern applications

5.3 Validation of the Authenticator Pattern and Authorization Pattern

Next, we verified that the developing model keeps the properties of the Authenticator and Authorization patterns.

The developer inputs the class model in XMI format shown in Figure 8 into VCGS. VCGS identifies that the inputted model has the Authenticator and Authorization patterns derived from stereotypes described in each class. VCGS then generates validation code from the transformations of the two patterns. Figure 17 shows the resulting validation code.

```
rule ValidateAuthenticatorProtectionObjectClass01 {
 from s : UML!Class (s.hasStereotype(Authenticator.Subject) == false)
 to t :  UML!Class (
             comment1 : UML!Note (
                body <- “No Class corresponding Subject role of Authenticator pattern.”)
}

rule ValidateAuthenticatorProtectionObjectClass02 {
 from s : UML!Class (s.hasStereotype(Authenticator.Subject) == true and )
 to t :  UML!Class (
             comment1 : UML!Note (
                body <- “No Class corresponding Subject role of Authenticator pattern.”)
}…

rule ValidateAuthorizationSubjectClass01 {
 from s : UML!Class (s.hasStereotype(Authorization.ProtectionObject) == false)
 to t :  UML!Class (
             comment1 : UML!Note (
                body <- “No Class corresponding Subject role of Authenticator pattern.”
```

```
    ) …
    rule ValidateAuthenticatorSubjectClass01 {
     from s : UML!Class (s.hasStereotype('Authenticator.Subject' == true and
    hasRelationTo('Authenticator.Authenticator')
     to t : UML!Class (
             comment1 : UML!Note (
                body <- "No relation between the class corresponding to Subject role of
Authenticator pattern and Authenticator role of Authenticator pattern.")
    }

    rule ValidateAuthorizationProtectionObjectClass01 {
     from s : UML!Class (s.hasStereotype(Authorization.ProtectionObject) == false)
     to t : UML!Class (
             comment1 : UML!Note (
                body <- "No Class corresponding Protection object role of Authorization
pattern."
    ) …
}
```

Fig. 17. Validation code of the Authenticator and Authorization pattern.

MTS adds a UML note describing the validation results in the inputs the validation code and the model in the XMI format of Figure 8. In the example shown in Fig17, the UML note does not describe anything because each pattern is correctly applied to the developing model.

6. EVALUATION

Here, we weigh the merits of our security pattern application against those of manual security pattern application. The comparison will be in terms of the number of work steps and the time required for the security pattern application. There are five work steps: (1) addition and deletion of a class, (2) addition and deletion of a relation, (3) input the name of a class, (4) automatic model transformation, and (5) input an argument in the model transformation. The time required was taken to be the mean of the times required to apply security patterns measured in an experiment involving six senior year university students who had experience with a UML modeling tool.

Figure 18 shows the times required for the security pattern application, and Table I lists the number of work steps. The proposed technique saves 70-90% of the time spent manually, and it reduces the number of steps by more than 50%.

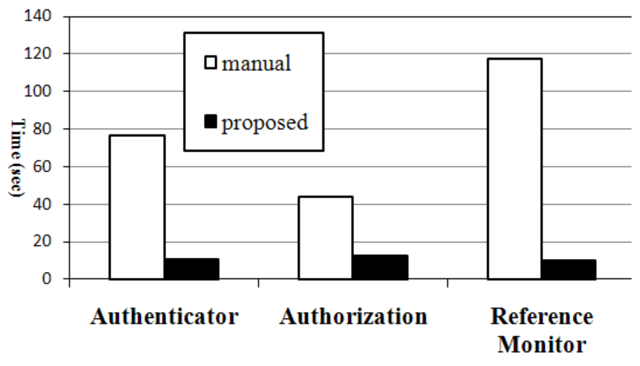


Fig. 18. Time required for the security pattern application

Table 1 Multivariate Changes in Image Quality Attributes, the Relationship of Psychometric and Objective Image Quality Estimations and the IBQ Approach

METHOD	Security Pattern		
	Authenticator	Authorization	Reference Monitor
MANUAL	7	4	5
PROPOSED	2	2	1

7. RELATED WORK

Yu et al. [3] proposed a security pattern technique that transforms the i* model using ATL. Moreover, Horvath proposed a technique for converting a model using the petri net. [5] Ours is different from those techniques because its model transformations use UML and its security patterns are written in ATL.

There is also a technique for ensuring that patterns are correctly applied[12] that works by validating a database application model from a stereotype. The validation done in [12] is similar to ours but has a different purpose.

8. CONCLUSION

Our technique enables automatic consecutive applications of security patterns that depend on each other by establishing a method of describing security pattern transformation rules and by marking the point at which the model was transformed. Although other patterns besides the ones discussed here can be applied, their dependencies may not be as obvious as those illustrated here. Our future work will include the following four tasks.

- Cover all 27 security patterns in [2] that can be treated by the proposed technique.
- Ensure the security of the entire system by using security patterns that strictly describe the security properties.

- Derive dependencies between patterns from the pattern documents by applying Kubo's technique [4].

- Quantitatively evaluate the accuracy of the security pattern applications.

At present our technique cannot be used for cloud computing and other area in which that security patterns are not used however, we believe that many security patterns will be proposed for such purposes in cloud and other field in the future.

REFERENCES

[1] Schumacher, M., Eduardo F. B., Duane, H., Frank, B. and Peter, S. 2006. Security Patterns, Wiley.

[2] Yoshioka, N., Hironori, W. and Katsuhisa, M. 2008. A Survey on Security Patterns. *Progress in Informatics*. no. 5, 3--47.

[3] Yu, Y. Haruhiko, K. Hironori, W. Yingfei, X. Zhenjiang, H. and Nobukazu, Y. 2008. Enforcing a Security Pattern in Stakeholder Goal Models. *ACM Proc. ACM workshop on Quality of protection (QoP'08)*

[4] Atsuto, K. Hironori, W. Atsuhiro, Takasu. and Yoshiaki, Fukazawa. 2005. Extracting Relations among Embedded Software Design Patterns. *Journal of Integrated Design and Process Science (SDPS)*. vol. 9, no. 3, 39--52.

[5] Horvath, V. and Dorges, T. 2008. From Security Patterns to Implementation Using Petri Nets. ACM.

[6] Schumacher, M. and Roeding, U. 2003. Security Engineering with Patterns. LNCS2754. 121--140.

[7] Frankel, D. 2003. Model Driven Architecture, Wiley.

[8] Eclipse.org. ATL Project. http://www.eclipse.org/m2m/atl/

[9] Jurjens, J. 2004. Secure Systems Development with UML. Springer.

[10] Torsten, L. David, B. and Jurgen, D. 2002. SecureUML: A UML-Based Modeling Language for Model-Driven Security. *In Proceedings of the 5th International Conference on the Unified Modeling Language*, 426--441.

[11] Guttorm, S. and Andreas, L. 2005. Elicting security requirements with misuse cases. Requir.Eng., vol. 10, no. 1, 34--44.

[12] Arnon, S. Jenny, A. and Perets, S. 2009. Validation and Implementing Security Patterns for Database Application. *3rd international Workshop on Software Patterns and Quality (SPAQu'09).*

[13] Christopher, S. Ramesh, N. and Ray, L. 2005. Core Security Patterns. Prentice Hall.

[14] Yuki, S. Atsuto, K. Eduardo F. B. Nobukazu, Y. Hironori, W. and Yoshiaki, F. 2010. Model-Driven Security Patterns Application Based on Dependencies among Patterns. *4th international Workshop on Secure systems methodologies using patterns (SPattern 2010)*. 555--559.

[15] MSDN. Web Security Service Patterns. http://msdn.microsoft.com/en-us/library/aa480545.aspx

Received August 2009; revised September 2010; accepted October 2010