Pattern-Based Transformation Rules for Developing Interaction Models of Access Control Systems

Dae-Kyoo Kim and Lunjin Lu

Department of Computer Science and Engineering
Oakland University, Rochester, MI 48309, USA
{kim2,l2lu}@oakland.edu

Abstract. This paper presents a set of transformation rules for transforming a non-secure interaction model to a secure interaction model using an access control pattern. The transformation rules resolve conflicts, uncertainties and type mismatches that may arise during pattern application. We demonstrate a case study using the Mandatory Access Control pattern and a defense messaging system in the military domain, and discuss about an analysis of the resulting model for pattern conformance.

1 Introduction

Access control provides integrity, confidentiality and availability of shared resources in a system. The development of an access control system involves high complexity due to the cross-cutting nature of access control. The complexity can be effectively managed by systematic use of access control models (e.g., see [2,4,10]) which describe a mechanism for governing access requests to shared resources at a high level of abstraction. We view an access control model as a design pattern that provides a generic solution for access control problems. This view promotes the reusability of an access control model and helps in detecting errors at earlier stages.

In this paper, we present a set of transformation rules for transforming a non-secure interaction model to a secure interaction model using an access control pattern. The transformation rules are used to resolve uncertainties, conflicts and type mismatches that may arise during pattern application. In our work, we describe interaction models in the Unified Modeling Language (UML) [7], a de facto standard language for modeling software systems, and access control patterns in the Role-Based Metamodeling Language (RBML) [5], a sub-language of the UML for precisely specifying design patterns. Use of precise specifications of access control patterns enables systematic reuse of access control patterns.

A major contribution of this paper is the transformation rules that 1) resolve uncertainties in determining the location to add pattern behavior in the model, 2) handle conflicts associated with operator fragments (e.g., *alt, break, opt*) in UML 2.0, and 3) address the problem of type mismatches where the type of a model element is different from that of its corresponding pattern element.

H. Mei (Ed.): ICSR 2008, LNCS 5030, pp. 306–317, 2008.

We demonstrate how the presented transformation rules can be used for transforming a model of a defense messaging system using the Mandatory Access Control (MAC) pattern [7]. The transformed model is analyzed for conformance to the applied pattern and the transformation rules. The remainder of the paper is organized as follows. Section 2 relates our work to other work. Section 3 describes an RBML specification of the behavior of the MAC pattern. Section 4 presents transformation rules. Section 5 demonstrates a case study using the technique, and Section 6 concludes the paper.

2 Related Work

Model transformation has gained great attention in aspect-oriented modeling [1,8,9] where cross-cutting concerns are modeled as design aspects separately from functional aspects. Clarke and Walker [1] proposed composition patterns to decompose and compose cross-cutting aspects based on subject-oriented techniques. The composition patterns are described in UML templates and composed with a functional model through parameter binding. Their work suffers from duplication problem [8] caused by one-to-one binding when a pattern is instantiated multiple times. The concept of roles in our work overcomes this limitation.

Reddy *et al.* [9] proposed a tag-based approach for composing sequence diagrams. Similar to Clarke and Walker's work, they use a variation of UML templates to design a cross-cutting behavior as a design aspect. The sequence diagram being composed can have two types of tags (simpleAspect and compositeAspect) that specify insertion points of the aspect in the model. A composite aspect includes position fragments (e.g., begin, end) which constrain the location of the fragment interactions that are added to the sequence diagram. Aspect parameters are bound to corresponding elements of the sequence diagram based on an implicit binding semantics. Their work has a similar limitation to Clark and Walker's work due to use of templates. The position fragments influenced the position directives in our work for designating insertion points.

Klein and Plouzeau [8] proposed a three-step approach for composing sequence diagrams. In the first step, the sequence diagram to be composed is decomposed into basic sequence diagrams which contain only sending and receiving messages and high-level sequence diagrams which contain fragment operators. In the second step, interface sequence diagrams that capture the common behaviors of the basic sequence diagrams are designed. In the third step, the designer determines if the pattern sequence diagram can be simply added into high-level sequence diagrams, or should be composed with the basic sequence diagrams. They accurately point out the problem of duplicate behaviors with templates when multiple instantiations are made. To address this issue, they use the interface sequence diagrams from the second step to exclude duplicate behaviors. However, use of interface sequence diagrams introduces places for potential errors and makes the composition process complicated. The concept of interface sequence diagrams is similar to the interaction patterns in our work in that they capture a common behavior.

3 Specifying Mandatory Access Control

Mandatory Access Control (MAC) is an access control model that governs access based on security levels [10]. We presented MAC as an access control pattern [7], and henceforth refer to MAC as the MAC pattern. The MAC pattern consists of the following concepts: *User*, *Subject*, *Object*, *Operation*, *Security Level*, *Category* and *Reference Monitor* [7]. *User* represents a user or a group of users who interact with the system. A user is assigned a hierarchical security level (e.g., SECRET, CONFIDENTIAL) and a non-hierarchical category (e.g., U.S., Allies). A user may have multiple login IDs which can be active simultaneously. A user may also create or delete one or more subjects. *Subject* represents a computer process which acts on behalf of the user to request an operation on the target object. *Object* represents any information resource in the system that can be accessed by user. Like a subject, an object is assigned a hierarchical security level and a non-hierarchical category. *Operation* is an action invoked by a subject to perform a task on the target object. *Security Level* represents a hierarchical classification assigned to users (subjects) and objects. *Category* represents any value from a non-hierarchical set. *Reference Monitor* checks the accessibility of the user by enforcing the following constraints. Given that *L(s)* is the security level of a subject *s* and *L(o)* is the security level of an object *o*:

- *Simple Security property*: A subject *s* can read an object *o* only if $L(s) \geq L(o)$.
- *Restricted *-property*: A subject *s* can write an object *o* only if $L(s) \leq L(o)$.

We use the Role-Based Metamodeling Language (RBML) [5] to specify the MAC pattern. RBML is a UML-based pattern specification language developed in our previous work [5] to precisely specify design patterns. The RBML defines a pattern in terms of roles which are played by UML model elements. Every role has a *base metaclass* in the UML metamodel and *metamodel-level constraints* which specialize the base metaclass to restrict the type of the model elements that can play the role. Every role has a realization multiplicity to constrain the number of elements playing the role. If the realization multiplicity is not specified, the default multiplicity *1..** is used requiring that there be at least one element playing the role.

Interaction Pattern Specifications (IPSs) are a type of RBML specifications capturing the interaction behavior of a pattern in a sequence diagram view. An IPS consists of lifeline roles and message roles whose base is the *Lifeline* metaclass and the *Message* metaclass in the UML metamodel. In the UML metamodel view, an IPS defines a specialization of the UML metamodel which characterizes a family of sequence diagrams. A member in the family is said to conform to the IPS [5]. A conforming sequence diagram must have lifelines that can play the lifeline roles in the IPS. A lifeline conforms to a lifeline role if the lifeline has messages whose sequence of incoming and outgoing messages is the same as that of the incoming and outgoing message roles on the lifeline role.

Figure 1 shows an IPS for the MAC pattern where roles are denoted by the symbol "|", and their base metaclass is shown implicitly by the graphical

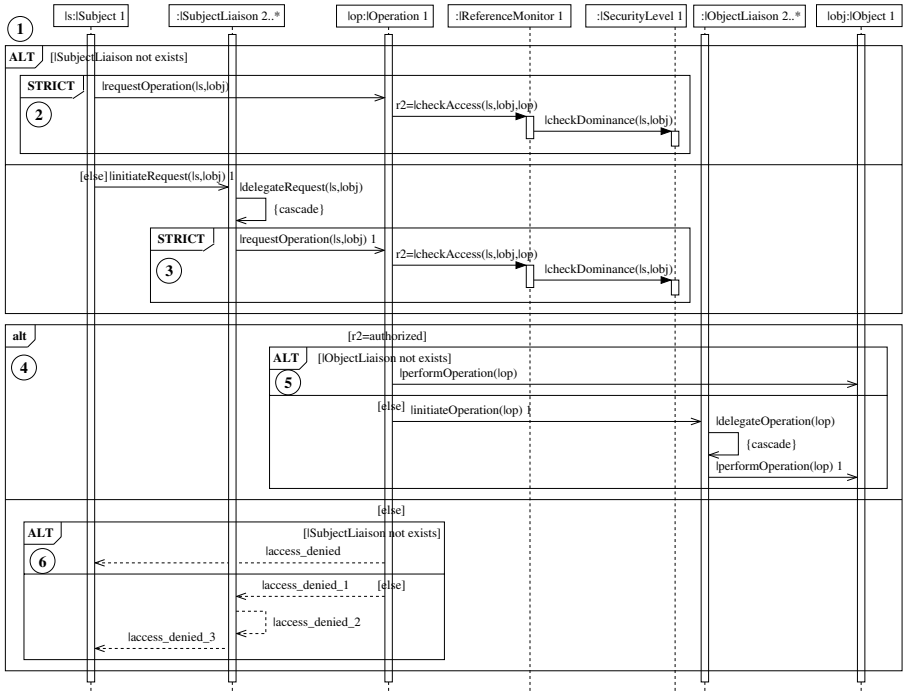

Fig. 1. MAC IPS

notation. The roles that have a realization multiplicity other than the default multiplicity *1..** are explicitly specified. The IPS shows use of two metamodel operators, **ALT** and **STRICT** which are defined in the RBML to constrain the structure of a conforming sequence diagram. The **ALT** operator is used to define alternative scenarios with a guard condition for conforming sequence diagrams. Only one scenario that satisfies the guard condition can appear in a conforming model. The **STRICT** operator preserves the message sequence in the fragment which should not to be disturbed by any other messages in a conforming model. This operator preserves a critical sequence of pattern behaviors. Note that these operators are different from the UML model operators (e.g., *alt*, *strict*) which design the behavior of objects at runtime. The RBML metamodel operators are distinguished in capital letters from UML model operators. An IPS may also have UML model operators as shown in Figure 1 (e.g., *alt*). Use of the *alt* operator requires a conforming model to have a corresponding operator that exhibits lifelines and messages that play the lifeline roles and message roles in the *alt* operator in the MAC IPS.

Given the notational background, the MAC IPS describes the following. A subject requests an operation with parameters including itself and the target object. There are two ways of sending the request as specified in the **ALT** fragment ①. One way is to send the request directly to the operation lifeline, and the other way is to send the request through subject liaisons which are intermediate

lifelines delegating the request to the operation object. Only one scenario may appear in a conforming model. It should be noted that the |*SubjectLiaison* lifeline role and its associated message roles (|*initiateRequest()*, |*delegateRequest()*, |*requestOperation()*) are required only in the second scenario of the fragment ①. The message roles have the following dependencies:

- An |*initiateRequest()* operation requires a |*requestOperation()* operation.
- A |*delegateOperation()* operation requires a |*requestOperation()* operation.

The *cascade* constraint on the |*delegateRequest()* role specifies that an *initialRequest()* message is delegated through intermediate lifelines playing the |*:SubjectLiaison* lifeline role until a *requestOperation* message is invoked on the *op:Operation* lifeline. A lifeline role that has a *cascade* constraint must have a realization multiplicity with a lower bound greater than or equal to 2 so as to obtain a meaningful delegation as shown in the |*:SubjectLiaison* role. The request is checked for accessibility by the *checkDominance()* operation which enforces the *simple security property* and *restricted-* property*, as described in the **STRICT** fragments ② and ③. These fragments mandate the interaction sequences not to be interfered by any other interactions in a conforming model. The *alt* fragment ④ describes the authorized case and denied case. If the request is authorized, it can be sent directly to the target object or through object liaisons (intermediate lifelines delegating the requests) as specified in the **ALT** fragment ⑤. If the access is denied, the request is sent back to the subject, which is described in the **ALT** fragment ⑥. A conforming sequence diagram must have an *alt* fragment corresponding to the *alt* fragment ④ with the same relative sequence of interactions as specified in the fragment ④.

4 Transformation Rules

Pattern-based model transformation is a process of transforming a model using a design pattern to improve a certain quality of the system. During transformation, conflicts, uncertainties or type mismatches may occur. To handle these issues, we define the following rules:

Unmapped Message Roles (UMR). Given a mapping, the location of the mapped message roles in the model is automatically determined to be where the mapped messages are present. However, the location for unmapped roles may not be determined. For example, in Figure 2(a) while the location of the |*m1'()* role is determined at the mapped message *m1()* in the transformed model, the location for the unmapped role |*m2'()* is not determined. According to the IPS, an instance (*m2'()*) of the |*m2'()* must be placed after the *m1()* message playing the *m1'()* role. However, the sequence diagram has another message *m2()* after *m1()* which should be taken into account in determining the location of *m2'()*. In consideration of *m2()*, *m2'()* can be placed either before *m2()* or after *m2()*, which is nondeterministic.

To resolve such an uncertainty, we define two position directives, [(*message*|*message role*) **before** (*message*|*message role*)] and [(*message*|*message role*) **after**

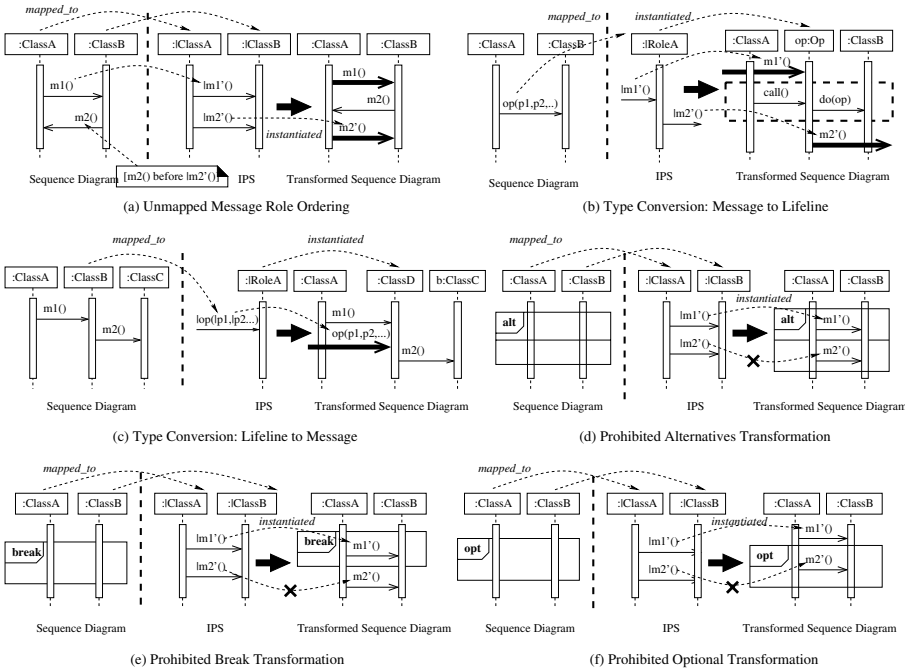

Fig. 2. Transformation Rules

(*message*|*message role*)] to designate a particular location for a message or a role instance. For example, [*a()* **before** |*b()*] stipulates that the message *a()* be placed immediately before an instance of the |*b()* role. Using position directives, the problem in Figure 2(a) can be resolved by specifying [*m2()* **before** |*m2'()*] which places the *m2()* message right before an instance of the |m2'() role (*m2'()*). Note that position constraints should not violate the pattern behavior. For example, having [|*m2'()* **before** *m1()*] would violate the pattern behavior due to the reverse sequence of the messages playing the |*m1'()* and |*m2'()* roles.

Type Mismatches (TM). Given a mapping, an element may be mapped to a role whose base is different from the type of the element. In such a case, a type conversion is required. A concrete example is found in the Visitor design pattern [3] where cross-cutting operations over an object structure are captured as visitor classes. In general, these operations are designed as operations, and use of the Visitor pattern requires transforming them to classes. To handle a message-to-lifeline type mismatch, we define the following rule:

Mapping a message op() *to a lifeline role* |:RoleA *creates a new lifeline* op:Op *(an instance) of the* |:RoleA *role and adds a* call() *message from the source lifeline of* op() *to the new lifeline* op:Op *and a* do() *message from the new lifeline* op:Op *to the target lifeline of* op().

Figure 2(b) illustrates the rule. In the figure, the *op()* message whose type is the *Message* metaclass is mapped to the *:|RoleA* role whose type is the *Lifeline* metaclass. This requires to transform the *op()* message to a lifeline (*op:Op*) that can play the *:|RoleA* role.

The TM rule reestablishes the interaction between the new lifeline *op:Op* and the source lifeline (*:ClassA*) and the target lifeline (*:ClassB*) of the *op()* message in the original sequence diagram using two auxiliary messages *call()* and *do()*. The *call()* message captures the operation call invoked by the source lifeline *:ClassA* and is added between the source lifeline *:ClassA* and the new lifeline *op:Op*. The *do()* message captures execution of the call on the target lifeline *:ClassB* and is added between the new lifeline *op:Op* and the target lifeline *:ClassB*. The *do()* message takes the new lifeline *:op:Op* as a parameter to execute it on the target lifeline *:ClassB*. There are two issues involved in a message-to-lifeline transformation. One is determining locations for the *call()* and the *do()* messages in the transformed model, which requires considering the message sequence in the sequence diagram and the sequence of the unmapped message roles in the IPS. To address this issue, we use the **before** and **after** directives presented in the UMR rule as follows:

op() $\mapsto$ *:|RoleA* ([*call()* **after** *|m1'()*],[*do()* **before** *|m2'()*])

This constraint specifies that the *call()* operation must be placed immediately after an instance of the *m1'()* role, and the *do()* operation immediately before an instance of the *m2'()* role. If there are multiple instances of the *|m1'()* role, the **after** directive places the *call()* message after the last instance. Similarly, if there are multiple instances of *|m2'()*, the **before** directive places the *do()* message before the first instance. The *call()* and the *do()* messages may be specified to play the *|m1'()* and the *|m2'()* roles, respectively, as follows:

op() $\mapsto$ *:|RoleA* ([*call()* $\mapsto$ *|m1'()*],[*do()* $\mapsto$ *|m2'()*])

In this case, the *|m1'()* and the *|m2'()* roles are not instantiated. If the message roles mapped to the *call()* and the *do()* messages involve parameter roles, they must be instantiated in the signature of the *call()* and the *do()* messages. The other issue to address is handling the parameters of the message being transformed. In Figure 2(b), the *op* message has two parameters (*p1, p2*). Since the message is transformed to a lifeline, the parameters of the message should be handled in some way. We assume that the parameters are transformed to attributes of the corresponding class of the new lifeline in the class diagram derived from the transformed sequence diagram (not shown in Figure 2(b)). This makes the parameters no longer expressive in the sequence diagram.

The opposite conversion from a lifeline to an operation is also possible. For example, a creator lifeline in a sequence diagram may be mapped to a creator message in a pattern (e.g., the Abstract Factory pattern [3]). To handle such a conversion, we define the following rule:

Mapping a lifeline :ClassA *to a message role* |op() *creates a new message* op() *(an instance) of the* |op() *role and a new lifeline* :ClassB *(an instance) of*

the target lifeline role of the |op() *role, and redirects the incoming and outgoing messages of the* :ClassA *lifeline to the new lifeline* :ClassB.

Figure 2(c) illustrates the rule. In the figure, the *:ClassB* lifeline is mapped to the |*op()* message role. The transformation rule creates instances (*op()*, *:ClassD*) of the |*op()* role and its target lifeline role *:|RoleA*, and redirects the messages *m1()* and *m2()* of the *:ClassB* lifeline to the new lifeline *:ClassD*. However, the location of the *op()* message in the transformed sequence diagram is not determined yet. There are three places where the *op()* message can be placed: before *m1()*, after *m2*, in between *m1()* and *m2()*. To designate a location, the following constraint is defined, placing the *op()* message in the third option:

:ClassB $\mapsto$ |*op()* ([|*op()* **after** *m1()*],[|*op()* **before** *m2()*])

In a lifeline-to-message transformation, we assume that the properties of the lifeline become properties of the target lifeline of the transformed message.

Operator Fragments (OF). The model being transformed may have fragments of the **alt**, **break** and **opt** operators [11] whose execution depends on a guard condition. If a pattern behavior is composed with a fragment of these operators, the pattern behavior cannot be guaranteed to be executed because of the conditional execution. To prevent this, we define the following rules:

- **alt** Rule. An **alt** fragment describes alternative scenarios determined by a guard condition. If the pattern behavior is split into the two choices of an **alt** fragment, the pattern behavior exhibited in the unselected choice at runtime will not be executed. To prevent this, the following rule is defined:

  A pattern cannot be split into the alternatives in an **alt** *fragment.*

  Figure 2(d) shows a prohibited transformation for an *alt* fragment. In the figure, the *m1'()* instance of the |*m1'()* role is composed with the first choice of an **alt** fragment, while the (*m2'()* instance of the |*m2'()* role is composed with the second choice. This is invalid because either of *m1'()* or *m2'()* in the *alt* fragment will not be executed, which violates the pattern.
- **break** Rule. A **break** fragment describes a terminating scenario for the sequence diagram. If a pattern behavior is composed with a **break** fragment, the pattern behavior will not be executed if the guard condition of the fragment is false. To prevent this, the following rule is defined:

  A pattern cannot be split into a **break** *fragment and the normal scenarios (the scenarios outside of the* **break** *fragment).*

  Figure 2(e) shows a prohibited transformation for a **break** fragment. The transformation shows that the *m1'()* instance of the |*m1'()* role is composed with the **break** fragment while the *m2'()* instance of the |*m2'()* role is composed with the normal scenarios. This should be prohibited because *m2'()* cannot be executed when the **break** fragment is enabled or vice versa.
- **opt** Rule. An **opt** fragment describes a choice of behavior depending on a guard condition. An **opt** fragment is similar to a *break* fragment in terms of

structure, but does not require to break out the normal scenario. As a matter of fact, an option is semantically equivalent to an alternative fragment where the first choice has non-empty content and the second choice is empty [11]. A similar rule to the **break** rule is defined for **opt** fragments as follows:

A pattern cannot be split into an **opt** *fragment and the regular scenarios (the scenarios outside of the* **opt** *fragment).*

Figure 2(f) shows a prohibited transformation for an **opt** fragment. In the figure, the (*m1'()*) instance of the |*m1'()* role is composed with the regular scenario, while the (*m2'()*) instance of the |*m2'()* role is composed with an *opt* fragment. This violates the pattern because the *m2'()* message will not be executed when the guard condition of the fragment is false.

5 A Case Study

We demonstrate the transformation rules using the MAC pattern applied to a defense messaging system (DMS) in the military domain. The DMS allows a user to create a new message, set up a sensitivity level for the message, and send and receive the message. Only an authorized and uniquely identified user can use the system. A sent message is sorted by the message sorter to identify the recipient. The recipient is checked for accessibility to the message based on MAC policies before receiving. If the sensitivity level of the recipient does not satisfy the sensitivity level set in the message, the recipient cannot receive the message, and the message is sent back to the sender. Every successful and erroneous transaction must be logged in persistence. In this case study, we assume that the security level of the message is same as the sender's. Figure 3 shows a sequence diagram describing sending a message without access control. We apply the MAC pattern to the sequence diagram based on the following mapping:

(*o:MsgSender* $\mapsto$ |*s:*|*Subject*), (*sendMsg_1()* $\mapsto$ |*initiateRequest()*), (*:MsgSorter* $\mapsto$ *:*|*SubjectLiaison*), (*:Delivery* $\mapsto$ *:*|*SubjectLiaison*), (*sendMsg_2()* $\mapsto$ |*delegateRequest()*), (*r:MsgRecipient* $\mapsto$ |*obj:*|*Object*), (*sendMsg_3()* $\mapsto$ |*op:*|*Operation* ([*call()* $\mapsto$ |*requestOperation()*],[*do()* $\mapsto$ |*performOperation()*])).

The mapping is given as input to the transformation algorithm [6] to evaluate conformance of the elements to the mapped roles by enforcing the metamodel-level constraints of the roles. If any nonconformance exists, the element is transformed to be conformant by the algorithm. The only metamodel-level constraint in the MAC IPS is the base metaclass which requires type matching. The only violating mapping of this constraint is (*sendMsg_3()* $\mapsto$ |*op:*|*Operation*) where the type of *sendMsg_3()* is the *Message* metaclass, while the type of the |*op:*|*Operation* role is the *Lifeline* metaclass. The transformation algorithm applies the TM rule to this pair to convert the type of *sendMsg_3()* to the *Lifeline* metaclass.

Applying the rule results in creation of a lifeline *op:sendMsg_3* and two messages *call()* and *do()*. The mappings of (*call()* $\mapsto$ |*requestOperation()*) and (*do()*

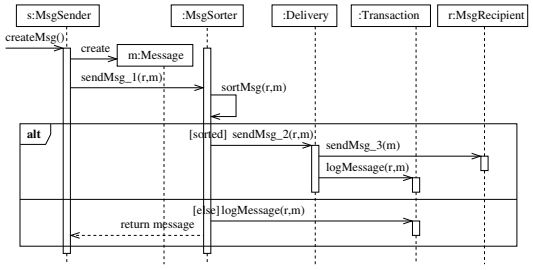

Fig. 3. Defense Messaging System

$\mapsto$ |*performOperation()*) require the parameter roles (|*s*, |*obj*, |*op*) of the *requestOpration()* and *performOperation()* roles to be instantiated into the signature of the *call()* and *do()* messages.

The mapping determines which scenario in the **ALT** fragments ①, ⑤ and ⑥ in Figure 1 the DMS model should conform to. The mappings (*:MsgSorter* $\mapsto$ *:|SubjectLiaison*) and (*sendMsg_2()* $\mapsto$ |*delegateOperation()*) determine the second scenario in the **ALT** fragment ① which describes delegation of the request through subject liaisons. This also determines the second scenario in the fragment ⑥. Similarly, the mapping (*do()* $\mapsto$ |*performOperation()*) determines the first scenario in the **ALT** fragment ⑤ which describes sending the request directly to the target object. These require the DMS model to have message sequences conforming to the second scenario of the **ALT** fragment ① and the first scenario of the **ALT** fragment ⑤.

The unmapped roles (*:|ReferenceMonitor*, *:|SecurityLevel*, |*requestOperation()*, |*checkAccess()*, |*checkDominance()*, |*access_denied_1*, |*access_denied_2*, |*access_denied_3*) in the MAC IPS are instantiated to be added into the model. The UMR rule is applied to determine the location of the unmapped message roles in the transformed model. In Figure 1, the |*requestOperation()*, |*checkAccess()* and |*checkDominance()* roles are specified in a **STRICT** fragment which requires their sequence to be preserved. Thus, instances of the roles are treated as one block. The location of the block can be determined relative to the locations of the messages mapped to the |*delegateRequest()* and |*performOperation()* roles which are immediately before and after the block. The above mapping shows that the two roles are mapped to the *sendMsg_2()* message and the *do()* message, respectively. This determines that the instance block must be placed between the *sendMsg_2()* and the *do()* messages. The return message roles in the second scenario of the fragment ⑥ are instantiated according to the mapping for the |*s:|Subject*, *:|SubjectLiaison* and |*op:|Operation* roles. Note that the unmapped roles in the unselected scenario in ①, ⑤ and ⑥ should not be instantiated unless they participate in the selected scenario.

The DMS model has one **alt** fragment for which the OF rule should be applied. The OF rule prohibits the MAC IPS from being split into the two different scenarios in the **alt** fragment. Given the mapping and enforcement of the *UMR* rule, the OF rule is observed.

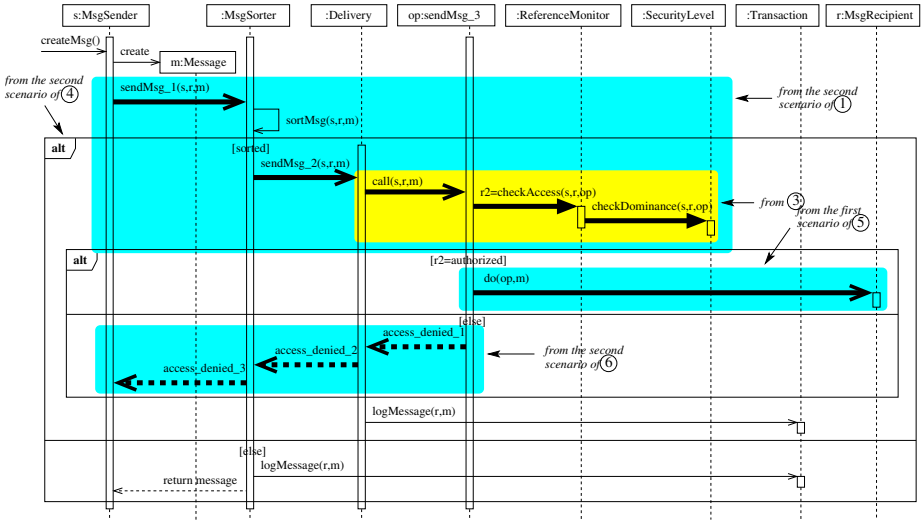

Fig. 4. Case 1: Defense Messaging System with MAC

Figure 4 shows the resulting sequence diagram where the pattern behaviors are highlighted. In the model, the MAC pattern intercepts the request from *Delivery* to the recipient to check dominance of the recipient's security level over the security level set on the message. If the recipient's security level is higher than the message's security level, the request is authorized, and the message is delivered to the recipient. Otherwise, the request is denied, and the message is sent back to the sender. The outer *alt* fragment is added by the **alt** fragment ④. The sequence of the *call()*, *checkAccess()* and *checkDomination()* messages in the outer *alt* fragment preserves the sequence in the *STRICT* fragment ③.

The transformed model should be checked for conformance to the pattern to ensure correct incorporation of the pattern behavior. We have conducted a conformance evaluation for the transformed model using logic programming. In the evaluation, we implemented the MAC pattern as a query and the transformed model as a logic program. The logic program is executed with the query to compute all feasible mappings by enforcing the conformance rules described in Section 4. The details of the approach are beyond the scope of this paper.

6 Conclusion

We have presented a set of transformation rules for developing interaction models using design patterns and demonstrated the application of the transformation rules via a defense messaging system and the MAC pattern. In addition to the case study presented in this paper, we have conducted two other case studies for a healthcare system and a database access system. The presented transformation rules are developed based on these case studies. We expect the

rules to be extended as more case studies are conducted. A possible extension would be converting an operation parameter to a lifeline or vice versa. Rigorous transformation rules presented in this paper together with a precise RBML specification of an access control pattern provides a solid foundation for mechanical pattern application. Also, the metamodeling design of a pattern facilitates the development of a prototype. We are currently developing a prototype tool for the proposed technique. Such a tool would enable not only automatic pattern application, but also automatic rollback of an applied pattern when necessary.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0523101.

References

1. Clarke, S., Walker, R.: Composition Patterns: An Approach to Designing Reusable Aspects. In: Proceedings of International Conference on Software Engineering, Toronto, Canada, pp. 5–14 (2001)
2. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST Standard for Role-Based Access Control. ACM Transactions on Information and Systems Security 4(3) (2001)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
4. Harrison, M.H., Ruzzo, W.L., Ullman, J.D.: Protection in Operating Systems. Communications of the ACM 19(8), 461–471 (1976)
5. Kim, D.: The Role-Based Metamodeling Language for Specifying Design Patterns. In: Taibi, T. (ed.) Design Pattern Formalization Techniques, pp. 183–205. Idea Group Inc. (2007)
6. Kim, D., Gokhale, P.: A Pattern-Based Technique for Developing UML Models of Access Control Systems. In: Proceedings of the 30th Annual International Computer Software and Applications Conference, Chigaco, IL, pp. 317–324. IEEE Computer Society Press, Los Alamitos (2006)
7. Kim, D., Mehta, P., Gokhal, P.: Describing Access Control Patterns Using Roles. In: Proceedings of Pattern Languages of Programming Conference (PLoP), Portland, OR (2006)
8. Klein, J., Plouzeau, N.: Transformation of Behavioral Models Based on Compositions of Sequence Diagrams. In: Proceedings of Model-Driven Architecture: Foundations and Applications 2004 (MDAFA), Linkoping, Sweden, p. 255 (2004)
9. Reddy, R., Solberg, A., France, R., Ghosh, S.: Composing Sequence Models using Tags. In: Proceedings of MoDELS workshop on Aspect Oriented Modeling, Genova, Italy (2006)
10. Sandhu, R., Samarati, P.: Access Control: Principles and Practice. IEEE Communications 32(9), 40–48 (1994)
11. The Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0 Formal/05-07-04, OMG (August 2005), `http://www.omg.org`