Using Security Patterns to Model and Analyze Security Requirements[*]

Sascha Konrad, Betty H.C. Cheng[†] Laura A. Campbell, and Ronald Wassermann
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824, USA
Email: {chengb,konradsa,campb222,wasser17}@cse.msu.edu

Abstract
Recently, there has been growing interest in identifying patterns for the domain of system security, termed *security patterns*. Currently, those patterns lack comprehensive structure that conveys essential information inherent to security engineering. This paper describes research into investigating an appropriate template for security patterns that is tailored to meet the needs of secure systems development. In order to maximize comprehensibility, we make use of well-known notations such as the Unified Modeling Language (UML) to represent structural and behavioral information. Furthermore, we investigate how verification of requirements properties can be enabled by adding formal constraints to the patterns.

1 Introduction

In today's systems with various communication features, security considerations are of greater interest than ever before. Therefore, a fair amount of additional security expertise is needed to meet non-functional security requirements [10]. A common approach to overcoming knowledge gaps among developers is to use patterns (e.g., analysis patterns [5], design patterns [6], specification patterns [3], etc.). In the security domain, it is challenging to capture and convey information in order to facilitate security engineering, which can often be an abstract goal. In this paper, we describe a collection of security patterns using a template similar to that used to describe design patterns [6]. We modified the design pattern template to better suit the presentation of security-specific and requirements-oriented information in order to facilitate reuse of security knowledge.

Providing expertise that significantly improves system development with respect to security is an ambitious goal. In contrast to functional requirements that have a concrete solution, security can be subjective and highly dependent on the environment. Other pattern-based approaches like the well-known *design patterns* from Gamma *et al.* [6] are believed to greatly enhance productivity of the software development process. Unfortunately, the structure used for various pattern templates is not sufficient to portray some key security-relevant aspects. More specifically, several recent approaches [4, 11, 14] that have attempted to apply patterns to the field of security use the original or a slightly modified design pattern template. The enhanced *security pattern template* presented herein contains additional information, including behavior, constraints, and related security principles, that addresses difficulties inherent to the development of security-critical systems.

The security needs of a system depend highly on the environment in which the system is deployed. By introducing and connecting general security properties with a pattern's substance, the developer may gain security insight by reading and applying the pattern. Furthermore, behavioral information and security-related constraints are included in our security pattern template. The developer can use this information to check if a specific design and implementation of the pattern is consistent with the essential security properties.

Our augmented security pattern template enhances the communication of security-specific knowledge that is related to a concrete application. Furthermore, by using our previously developed UML formalization framework [9] that supports the generation of formally specified models (defined in terms of Promela, the language for the Spin model checker [7]), the security pattern template can facilitate the verification of security-relevant properties. (Hydra [9] generates formal specifications from the UML diagrams, and MINERVA provides support for graphical construction of UML diagrams and visualization of analysis results depicted in terms of the original UML diagrams [2].) Hence, overall security can be improved during the development process by applying this pattern-based approach.

The remainder of this paper is organized as follows. Section 2 defines security patterns and introduces our template. Using our previous work into formalizing UML and formally analyzing UML diagrams against specific properties, Sec-

[*]This work is supported in part by NSF grants EIA-0000433, CDA-9700732, CCR-9901017, and Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744.

[†]Please contact B. Cheng for all correspondences.

tion 3 overviews example analyses that we applied to the models from a case study. Finally, in Section 4, we summarize the work and discuss future investigations. The Appendix contains an example security pattern, the *Check Point* Security Pattern.

2 Security Patterns

In order to facilitate technology transfer, we used the design pattern template [6] as the basis for describing security patterns. Fields retained from the design pattern template are: *Pattern Name and Classification*, *Intent*, *Also Known As*, *Motivation*, *Applicability*, *Structure*, *Participants*, and *Collaborations*. We also added several new fields to capture information specific to security and emphasizing requirements-level information, such as *Behavior, Constraints,* and *Supported Security Principles*. The next section describes the modifications to the design pattern template in more detail.

Security Pattern Template

We altered several of the fields from the design pattern template in order to convey more security-relevant information than the original template. Below is a list of the fields that we modified or added to the design pattern template in order to describe security patterns.

- **Applicability** (Altered) : The *Applicability* field is important in determining if a pattern is applicable to a system; it describes circumstances and identifies assumptions under which the pattern should be applied or when an application is not beneficial. This field also indicates whether it addresses application-level, host-level, or network-level security.
- **Behavior** (Added) : To illustrate behavior and interaction more rigorously, UML state and sequence diagrams depict the behavioral aspects of a pattern.
- **Constraints** (Added) : The *Constraints* field contains global conditions that have to apply in order for the security pattern to achieve its intended goal. For example, the *Single Access Point* Pattern constraints describe that access to internal entities must be only possible through the single access points. If those constraints are violated, such as by a user using a dial-up connection from within the system to external entities, then the *Single Access Point* Pattern can no longer achieve its intended functionality, and therefore security. Additionally, it contains LTL constraints in the spirit of the *specification patterns* by Dwyer *et al.* [3] that should be satisfied after a pattern was applied.
- **Consequences** (Altered) : Altered to convey a set of possible consequences, defined in terms of security properties, including *accountability, confidentiality, integrity, availability, performance, cost, manageability,* and *usability*. (The identification of these properties were based on a set of properties identified by Kienzle [8]. For each security pattern, we describe how the pattern affects these properties.)
- **Related Security Patterns** (Added) : Describes a pattern's relationships to other security patterns.
- **Supported Principles** (Added) : The *Supported Principles* field refers to the ten principles identified by Viega and McGraw [12] as being fundamental to the development of secure systems.[1] Therefore, as a guide to the developer, we include a list of principles that are supported for each security pattern. Sample principles include, "secure the weakest link," "practice defense in depth," "fail securely," etc.

Current Repository of Security Patterns

Table 1 contains a brief description of the intent of security patterns that we have currently specified in terms of our security template. More details about the security patterns can be found in [13] and an example pattern, the *Check Point* Security Pattern, can be found in the Appendix. Note that several of these patterns have been discussed at varying levels of detail by others [4, 14], where the emphasis of the previous work was on architectural-level specifications or security frameworks. In contrast, our pattern template contains fields that emphasize requirements-level information, such as behavior and global constraints.

Using our template, we have added information and/or clarified several issues surrounding the use of these patterns, including the construction of UML diagrams (both static and behavioral diagrams), how the pattern relates to fundamental security principles, and how security properties can be formally checked. By having a more comprehensive description of the patterns, the information in our template can be used for requirements analysis as well as design-level development, thus potentially increasing the scope of applicability when compared to the original descriptions of the patterns.

Similar to design patterns [6], we can classify our security patterns into the categories *Creational*, *Structural*, and *Behavioral*. Table 2 shows this classification, as well as each pattern's focus for addressing security, defined in terms of *application-level*, *host-level*, and *network-level*. In order to give a high-level perspective on the purpose of each pattern, Table 2 illustrates each pattern's relationship to the ten security principles [12].

3 Security Pattern-Driven Modeling and Analysis

In this section, we briefly present a process for modeling and analyzing secure systems using security patterns. We demonstrate how this process can be applied to a simple e-commerce system.

Process for Using Security Patterns

Figure 1 overviews the process for using security patterns that enables simulation and model checking of the resulting system models.

1. Use the security patterns to construct a basic foundation

[1]These principles are similar to the security principles previously identified by Saltzer and Schroeder [10].

Single Access Point:	Improve control and monitoring by defining a single interface for communication with system external entities.
Check Point:	Check incoming requests for violations and take appropriate actions.
Roles:	Abstract rights from specific subjects, thereby facilitating maintenance of complex privilege structures.
Session:	Provide different parts of a system with global information about a user currently interacting with internal components.
Full View with Errors:	Prevent users from performing illegal operations while failing securely.
Limited View:	Avoid illegal operations by offering users only valid operations.
Authorization:	Facilitate access control of resources.
Multi-level Security:	Handle access in a system with various security classification levels. (Imposes Bell-Lapadula model [1] for preserving confidentiality among different levels.)

Table 1: Current list of security patterns

Patterns	***Purpose[1]***	***Abstraction level[2]***	***1. Secure weakest link***	***2. Practice defense in depth***	***3. Fail securely***	***4. Principle of least privilege***	***5. Compartmentalize***	***6. Keep it simple***	***7. Promote privacy***	***8. Hiding secrets is hard***	***9. Be reluctant to trust***	***10. Use community resources***
Single Access Point	S	AHN	x					x			x	
Check Point	S	AHN	x	x		x					x	
Roles/RBAC	S	AHN				x		x	x			
Session	C	AHN		x				x	x			
Limited View	B	A		x		x		x	x	x	x	
Full View with Errors	B	A			x			x				
Authorization	S	AHN				x	x		x			
Multilevel Security Pattern	S	AHN				x	x		x	x	x	
			Viega's and McGraw's 10 principles									

[1]Purpose:
C: Creational
S: Structural
B: Behavioral

[2]Abstraction levels:
A: Application-level
H: Host-level
N: Network-level

Table 2: Security pattern overview

for the system and refine the UML high-level structural and behavioral models of the system (Figure 1, Step A).

2. Use *Hydra* [2, 9] to perform consistency checks (Figure 1, Step B) and generate a *Promela* model of the system from the UML diagrams (Figure 1, Step C).
3. Using MINERVA's [2] visualization utilities to display simulator output within the context of the UML diagrams, simulate the model from the first step for validation purposes (Figure 1, Step D), and then refine the system model if errors are discovered.
4. Specify properties for the system by instantiating the specification patterns from the *Constraints* field of security patterns (Figure 1, Step E). Determine which parts of the system are relevant to a given property to be checked, and introduce abstractions for the remaining parts of the system where possible. To simplify analysis, create equivalence classes for attribute values when possible.
5. Use a model checker (e.g., SPIN [7]) to determine if the properties are satisfied. If errors are detected, then use MINERVA's [2] visualization utilities to display the counterexample in terms of the UML diagrams, and then refine the system model. (Figure 1, Step F)

Construction of UML Models

We use a small e-commerce system (**eBiz**) as a working example to demonstrate how security patterns can be used to construct UML models for a secure system. For the **eBiz** example, we applied the *Authorization*, *Check Point*, and *Single Access Point* security patterns to construct a basic foundation for the system model defined in terms of UML structural (class) and behavioral (state) diagrams. Then, using our formalization work and tool suite, we analyzed security-relevant properties using model checking. We focus in the subsequent security analysis of the system on constraints related to the *Check Point* Pattern.

General security requirements depict that an unauthorized access to the system should not be possible and triggers a countermeasure. In order to determine whether an access is authorized, the `SecurityPolicy` class is used. (In our example system, the security policy is modeled as an access matrix in the `Authorization` class that deter-

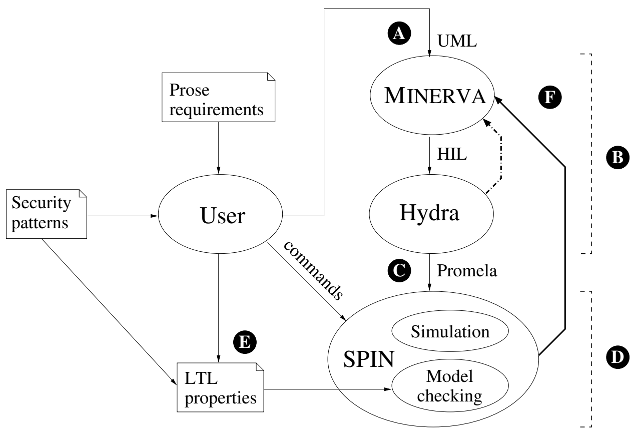

Figure 1: Process for using security patterns

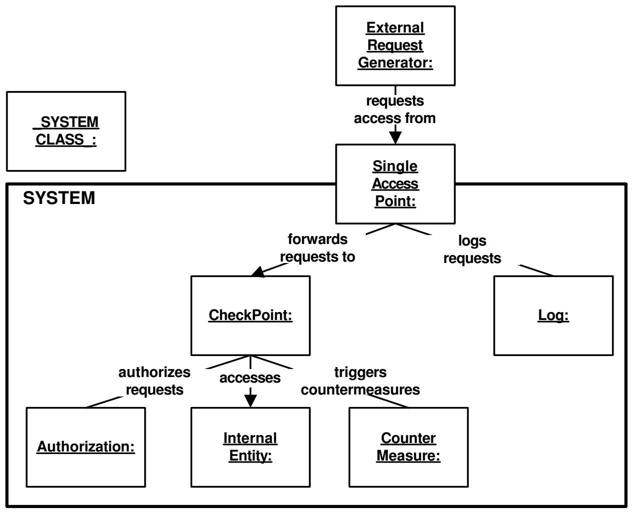

Figure 2: UML object diagram for **eBiz**

mines, based on the sender and receiver identity of a message, whether access is granted. A different name is chosen because a security policy usually contains more information than an access matrix.)

Figure 2 depicts the UML object diagram of the **eBiz** system. The diagram contains eight objects, where the _SYSTEMCLASS_ is used only to define the initialization and instantiation process of the system. (The bold rectangle labeled *SYSTEM* illustrates the boundary between the **eBiz** system and external entities making requests; it is not a UML construct.) The nondeterministic ExternalRequestGenerator randomly creates messages that simulate external access requests to the system and are sent to the SingleAccessPoint. Before a message is forwarded to the CheckPoint, a logging request is sent to the Log object. Upon arrival of a message, the CheckPoint consults the Authorization object to determine whether the access is permissible or not. In case of an access violation, the CheckPoint triggers the execution of a CounterMeasure. If the message is consistent with the privilege structure represented by Authorization, then the request is forwarded to the InternalEntity. Success or refusal of access is finally reported to the external entity that is represented by the ExternalRequestGenerator.

The CheckPoint state machine is depicted in Figure 3 and is based on the high-level state diagram of the *Check Point* Pattern depicted in Figure 7. Directly after the starting-transition, the CheckPoint sends a *ready()* notification to the _SYSTEMCLASS_ and enters its MainIdle state. This state can only be left by one transition that is triggered by a *checkMsg(sender,recipient,accessType)* event. Once all information is available in the CheckPoint class, a confirmation of the Authorization class is requested and the state waitAuth is activated. The CheckPoint class remains in this state until the Authorization responds with its *authOK(grantAccess)* event. Then the ProcessRequest state is reached. The grantAccess attribute contains the evaluation of the Authorization class. The variables sender, recipient, and accessType are reset to *zero* after the processing of a request is completed. Depending on the value of the grantAccess variable, the state machine enters either the AccessDenied or the AccessGranted state. Both states lead to a return to the MainIdle state. Yet, the transition leaving the AccessDenied state triggers a counter measure whereas the transition from the AccessGranted state to the MainIdle state initiates the access to the InternalEntity.

As an abstraction to simplify analysis, we created *equivalence classes* for several system attributes representing request status, communication status, and sender/receiver identity. Figure 4 lists the **eBiz** system's attributes and their possible values. The attributes (variables) with value *undetermined* indicate that the corresponding object is in between completing a request and the arrival of a new request.

Analysis of Security Properties

Using our approach, we checked specification-pattern-influenced [3] constraints from the *Authorization*, *Check Point*, and *Single Access Point* security patterns against the Promela specifications of the UML models of the **eBiz** system. We instantiated these constraints in terms of linear time temporal logic (LTL) to enable analysis with the Spin model checker. Due to space constraints, we present analysis of properties from only the *Check Point* security pattern. All of the following properties were generated by instantiating the

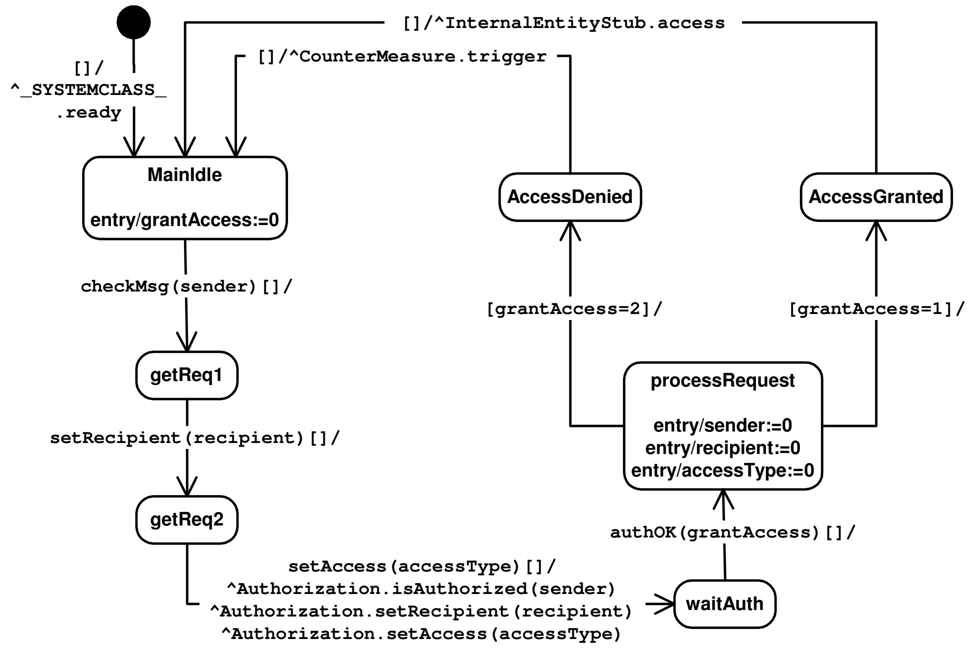

Figure 3: UML state diagram of the `CheckPoint` class

LTL pattern templates that can be found in the description of the *Check Point* Security Pattern.

Property *CHK1*, a general security constraint, states that an unauthorized access leads to the activation of a countermeasure. This constraint can be expressed in LTL as $\Box(p \rightarrow (\Diamond q))$ (it is always the case that if p, then eventually q). In terms of the **eBiz** system, the *CHK1* property can be expressed as shown in Row 1 of Table 3, meaning that it is always the case that if the `CheckPoint` (`CP`) denies access to a request (`grantAccess==2`), then eventually a countermeasure (`CM`) will become triggered (`triggered==1`). (Note that the names of the objects have been abbreviated in Table 3 for ease in presentation purposes.)

Property *CHK2* checks whether integrity or confidentiality is compromised after the `CheckPoint` has denied access to a request. In order to verify that a request remains unsuccessful after its refusal, we use the property: $\Box(p \rightarrow (q \; \mathcal{U} \; r))$ (it is globally true that p infers q until r). Instantiated for the **eBiz** system, the *CHK2* property can be expressed as shown in Row 2 of Table 3, meaning that it is always the case that if the `CheckPoint` (`CP`) denies access to a request, then the `ExternalRequestGenerator` (`ERG`), and hence, the current request, is not successful (`success==0`) until the `ExternalRequestGenerator` sends the next request (`waiting==0`).

Property *CHK3* embodies an availability constraint. It verifies that when a `CheckPoint` grants access to a request, the internal entity notifies the external entity of the access' successful completion. The property has the structure: $\Box(p \rightarrow ((\Diamond q) \; \mathcal{U} \; r))$ (it is globally true that p infers eventually q until r). In terms of the **eBiz** system, the *CHK3* property can be expressed as shown in Row 3 of Table 3, meaning that it is always the case that if the `CheckPoint` (`CP`) grants access to a request (`grantAccess==1`), then eventually the `ExternalRequestGenerator` (`ERG`) is successful (`success==1`) until the `ExternalRequestGenerator` sends the next request (the *eventually* operator $\Diamond$ is needed because notification of success does not occur immediately).

All three `CheckPoint`-related properties (*CHK1*, *CHK2*, and *CHK3*) shown in Table 3 were eventually successfully verified using the Spin model-checker. During the iterative verification process, the model checking was able to detect various errors that were visualized by MINERVA in terms of the UML diagrams, thus facilitating the model refinement process. For example, one error was uncovered in the state diagram of the *Authorization* class. Figure 5 shows an excerpt from the faulty UML state diagram, including the faulty transitions. We inadvertently defined the wrong recipient in the highlighted transitions.

Instead of *recipient=2* on the left transition and *recipient!=2* on the right transition from Sender 3, we specified *recipient=1* (left) and *recipient!=1* (right). This error did not cause a deadlock, because the two transitions were still disjunct and complete, but the logic did not correctly represent the access structure of our system. The error caused the system to grant unauthorized access, if sender 3 requested read access to recipient 1. Furthermore, sender 3 was not able to read from recipient 2 although it had authorization.

4 Conclusions

This paper has presented a template for security patterns that addresses difficulties inherent to the development

Request Status

$$\texttt{accessType} = \begin{cases} 0 & \text{(Undetermined)} \\ 1 & \text{(Read-access)} \\ 2 & \text{(Write-access)} \end{cases} \tag{1}$$

$$\texttt{grantAccess} = \begin{cases} 0 & \text{(Undetermined)} \\ 1 & \text{(Access granted)} \\ 2 & \text{(Access denied)} \end{cases} \tag{2}$$

$$\texttt{success} = \begin{cases} 0 & \text{(Request successful)} \\ 1 & \text{(Request unsuccessful)} \end{cases} \tag{3}$$

Communication Status

$$\texttt{logged} = \begin{cases} 0 & \text{(Logging idle)} \\ 1 & \text{(Logging complete)} \end{cases} \tag{4}$$

$$\texttt{triggered} = \begin{cases} 0 & \text{(CounterMeasure idle)} \\ 1 & \text{(CounterMeasure triggered)} \end{cases} \tag{5}$$

$$\texttt{waiting} = \begin{cases} 0 & \text{(Requestor idle)} \\ 1 & \text{(Requestor awaiting response)} \end{cases} \tag{6}$$

Sender/Receiver Identity

$$\texttt{sender} = \begin{cases} 0 & \text{(Undetermined)} \\ 1 & (\texttt{ExternalEntity1}) \\ 2 & (\texttt{ExternalEntity2}) \\ 3 & (\texttt{ExternalEntity3}) \\ 4 & (\texttt{ExternalEntity4}) \end{cases} \tag{7}$$

$$\texttt{receiver} = \begin{cases} 0 & \text{(Undetermined)} \\ 1 & (\texttt{InternalEntity1}) \\ 2 & (\texttt{InternalEntity2}) \end{cases} \tag{8}$$

Figure 4: Possible values for **eBiz** system attributes

ID	*LTL property*	*Expression p*	*Expression q*	*Expression r*
CHK1	$\Box(p \rightarrow (\Diamond q))$	(CP.grantAccess==2)	(CM.triggered==1)	
CHK2	$\Box(p \rightarrow (q\ \mathcal{U}\ r))$	(CP.grantAccess==2)	(ERG.success==0)	(ERG.waiting==0)
CHK3	$\Box(p \rightarrow ((\Diamond q)\ \mathcal{U}\ r))$	(CP.grantAccess==1)	(ERG.success==1)	(ERG.waiting==0)

Table 3: *Check Point* security properties instantiated for **eBiz**

of security-critical systems, including identifying security-specific consequences of applying a pattern, formally specifying security-oriented constraints, and explicitly identifying which security-specific development principles are being addressed when applying a given pattern. Using these patterns, developers new to addressing security issues during systems development can gain insight into how security concerns can be modeled and analyzed starting from the requirements engineering phase. It is important to understand that security patterns are not a "silver bullet" that can make any system secure if applied. The patterns have to be analyzed for applicability, and the consequences, liabilities, and constraints of a pattern have to be understood for its application to be successful.

When used with our previous UML formalization work [9] and tool suite [2], security patterns can facilitate rigorous analysis of security-relevant properties. Using the pattern-based approach, security aspects can be addressed during early phases of software development, thereby reducing error-prone post-design security modifications.

We are continuing to expand this work in several directions. First, we are exploring how constraints that capture various *beliefs* about security can be incorporated into the security pattern template. Second, we are also investigating whether the domain of an application plays a role in specifying security patterns. Finally, we are exploring the use of timing information when specifying security properties.

A Check Point

The *Check Point* Pattern that is explained herein is also part of a variation of Yoder's and Barcalow's framework for application security [14]. The fields *Structure*, *Participants*, *Collaborations*, *Constraints*, *Consequences*, and *Supported Principles* contain new information, where the emphasis is on requirements-level information. The other sections were based in part on information from Yoder's and Barcalow's pattern.

- **Pattern Name and Classification**
  The *Check Point* Pattern is a structural pattern.
- **Intent**
  Check Point proposes a structure that checks incoming requests. In case of violations, the Check Point is responsible for taking appropriate countermeasures.
- **Also Known As**
  The *Check Point* Pattern is also referred to as [14]:
  - Access Verification,
  - Authentication and Authorization,
  - Holding off hackers,
  - Validation and Penalization, or
  - Make the punishment fit the crime.
- **Motivation**
  In order to prevent unauthorized access, it is vital to

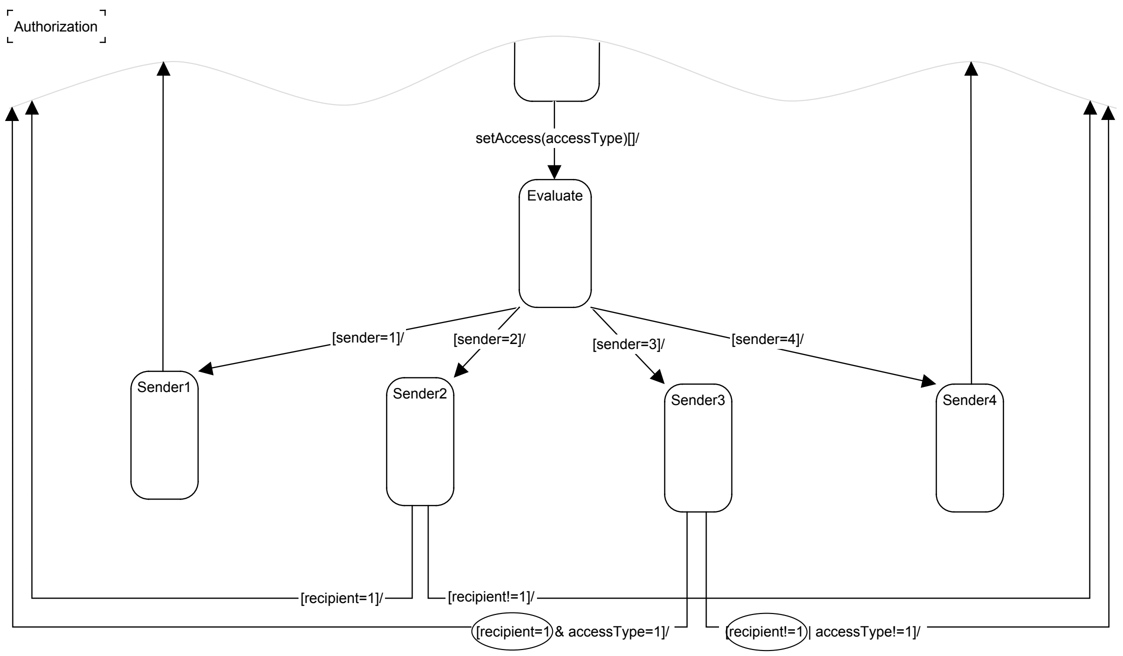

Figure 5: Faulty UML state diagram of the `Authorization` class

check who interacts in which manner with a system. It can be a difficult task to determine whether a given access should be granted or not. Any secure system needs a component that monitors the current communication and takes measures if necessary. Furthermore, usability should not be affected by denying innocuous actions. These tasks are specified in a component called a Check Point.

- **Applicability**
  Check Points are applicable for any security-relevant communication. It can be used at each abstraction-level from inside-application-level to network-level.
  In order to perform a check, the system needs to have a policy that will be enforced. The Check Point implementing that policy should be able to distinguish between user mistakes and malicious attacks.

- **Structure**
  A Check Point is a component that analyzes requests and messages. In order to monitor message flow it needs to be placed between different parties that exchange data. A good position to install a Check Point is wherever it can intercept a major part of the traffic. Hence, a Single Access Point is most likely to be combined with a Check Point - all messages will be monitored. This structure is depicted in Figure 6.

- **Participants**
  - `CheckPoint`
    - implements a method to check messages according to the current security policy.
    - triggers actions that might be necessary to protect the system against attackers.
    - grants messages access, if they are considered innocuous.
  - `Countermeasure`
    - provides a set of actions that can be triggered in order to react to an access violation.
  - `SecurityPolicy`
    - object that implements the rules that are applied to determine whether an access or condition is allowed.

- **Collaborations**
  The `CheckPoint` class monitors the message flow between other components. The calling class requests the `CheckPoint` to evaluate a certain message. To determine whether a request is entitled or not, the `SecurityPolicy` object is consulted. Once it is approved, according to the policy, the request will be forwarded to its intended recipient. If illegal request attempts or attacks are uncovered, then the `CheckPoint` triggers appropriate counteractions (implemented in a `CounterMeasure` object).

- **Behavior**
  Figure 7 shows the state diagram of the *Check Point* Pattern. The `CheckPoint` evaluates a request and, based on this evaluation, it either forwards the message or triggers a countermeasure. The UML sequence diagrams in Figure 8 shows a possible scenario where a message is rejected by the `CheckPoint`. Three sequence diagrams have been elided due to space constraints and can be found in [13].

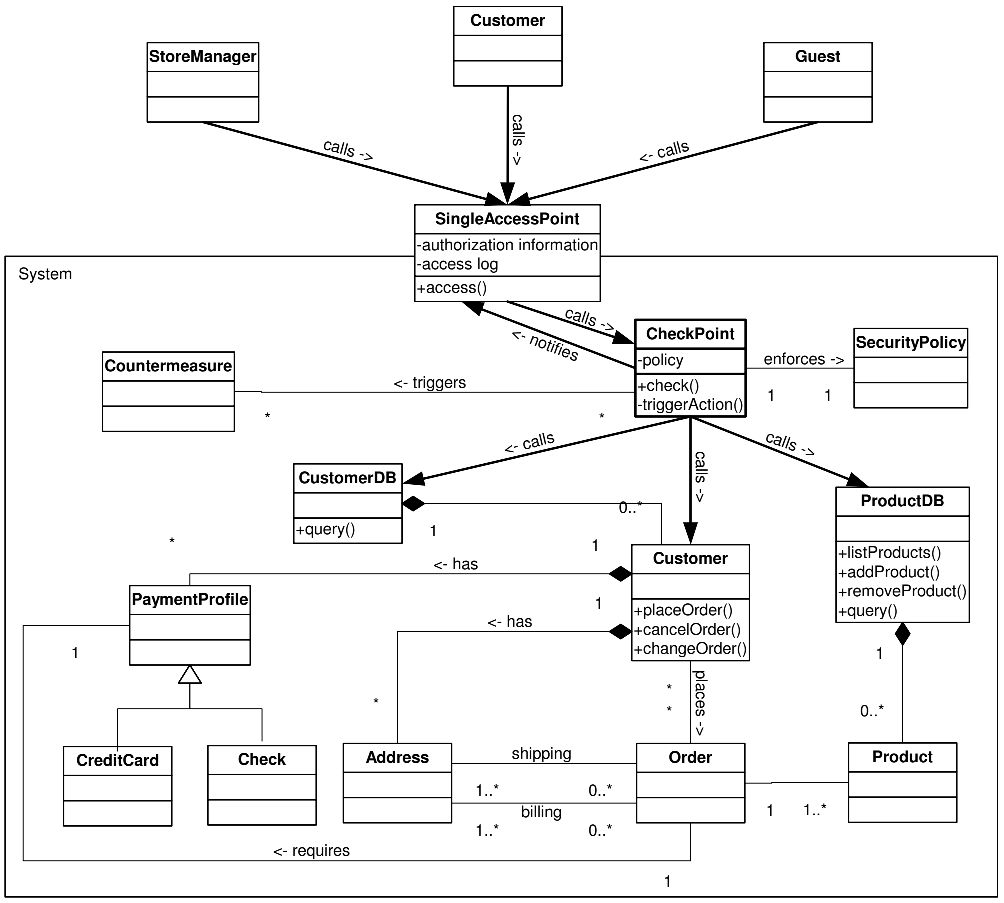

Figure 6: UML class diagram of the *Check Point* Pattern

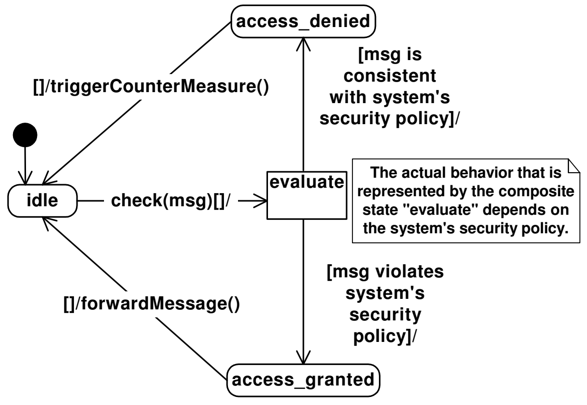
access_denied
[]/triggerCounterMeasure()
[msg is consistent with system's security policy]/
idle
check(msg)[]/
evaluate
The actual behavior that is represented by the composite state "evaluate" depends on the system's security policy.
[]/forwardMessage()
[msg violates system's security policy]/
access_granted

Figure 7: UML state diagram of the *Check Point* Pattern

- **Constraints:**

  Figure 9 gives assumptions about global conditions that must be applicable when applying the *Check Point* Pattern. In contrast, the following properties should be satisfied in the context of the pattern's application:

  - **Security Property 1:**
    An unauthorized access leads to the activation of a countermeasure.
    $\Box$(``Unauthorized Access'' $\rightarrow$ $\Diamond$(``Countermeasure taken''))
  - **Security Property 2:**
    When the *Check Point* Pattern denies a request, it is important that the current request remains unsuccessful until a new request is received. This ensures that integrity and confidentiality is not compromised after a request was denied.
    $\Box$(``Access denied'' $\rightarrow$ (``Request unsuccessful'' $\mathcal{U}$ ``Next request''))
  - **Security Property 3:**
    When an access is granted by the check point, then the external entity should eventually be able to access the system successfully until the next request is received.
    $\Box$(``Access granted'' $\rightarrow$ $\Diamond$(``Request successful''))

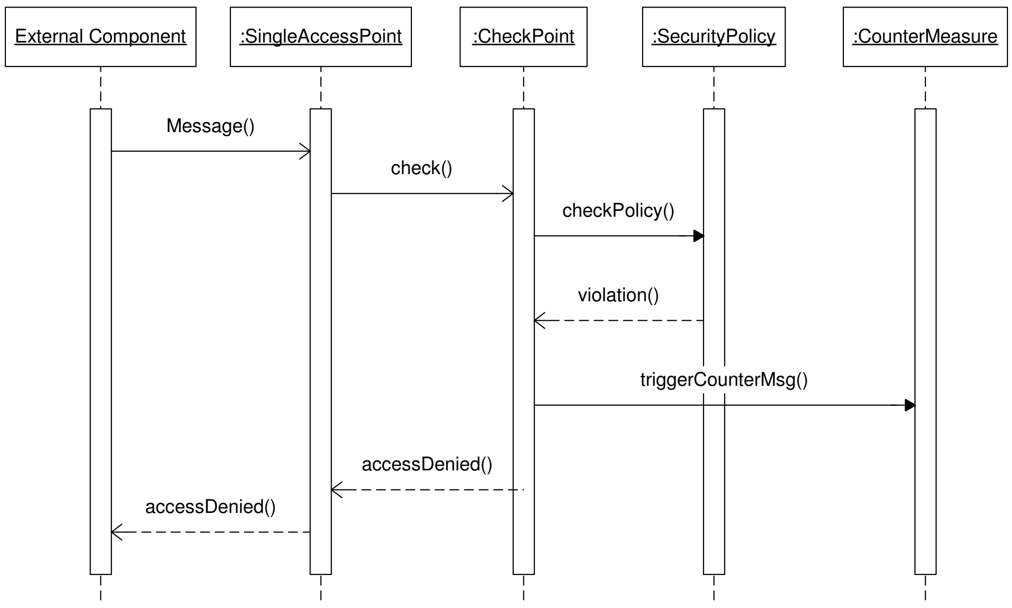

Figure 8: UML sequence diagram of the *Check Point* Pattern: access denied

```
    𝒰 ``Next request''))
```

- **Consequences:**
  Figure 10 enumerates the consequences of the *Check Point* Pattern.
- **Implementation**
  To be determined.
- **Known Uses**
  - UNIX telnet application
  - Windows NT login application
- **Related Security Patterns**
  The *Check Point* Security Pattern enforces the current security policy, by monitoring the traffic passing through the Single Access Point. Once a request is verified positively, the *Session* Pattern can be used to grant an external party access rights that exceed the current request. The *Role* Security Pattern facilitates the management of access rights when assigned to a Session.
- **Related Design Patterns**
  The Strategy Design Pattern [6] can be used to decouple the Check Point from the actual implementation of the security policy. Changes can be adopted more easily.
- **Supported Principles**
  The *Check Point* Pattern can be used to *secure the weakest link* (Principle 1). Applied in combination with other security mechanisms it *practices defense in depth* (Principle 2). If the *principle of least privilege* (Principle 4) is part of the current security policy, then it will be implemented in the check algorithm, used by Check Point. Monitoring messages and other data shows *reluctance to trust* that they are harmless (Principle 9).

REFERENCES

[1] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report Mitre TR-2997, Mitre Corporation, Bedford, MA, March 1976.

[2] L. A. Campbell, B. H. Cheng, W. E. McUmber, and R. Stirewalt. Automatically detecting and visualizing errors in UML diagrams. *Requirements Engineering Journal*, December 2002.

[3] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *Proceedings 2nd Workshop on Formal Methods in Software Engineering*, pages 7–16, Clearwater Beach, FL, Mar. 1998.

[4] E. B. Fernandez and R. Pan. A pattern language for security models. In *8th Conference on Pattern Languages of Programs*, September 2001.

[5] M. Fowler. *Analysis Patterns: reusable object models*. Addison Wesley Longman, Inc., 1997.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[8] D. M. Kienzle, M. C. Elder, D. S. Tyree, and J. Edwards-Hewitt. Security patterns template and tutorial, June 2002.

Authenticity	Check Point's requests concerning the current Security Policy are only answered by the real Security Policy object. No malicious agent in the system can replace the Security Policy.
Confidentiality	The communication between Check Point and the Security Policy may not be overheard by external entities.
General Security	If a checked message is consistent with the implemented security policy, then the request may be forwarded to its intended recipient.
General Security	If a checked message is potentially harmful and further action is required according to the implemented security policy, then the appropriate actions are triggered.
General Security	If a checked message is potentially harmful according to the implemented security policy, then the message is not forwarded to its intended recipient.
Integrity	Messages sent between Check Point and Security Policy cannot be modified.

Figure 9: Assumptions for global conditions when applying the *Check Point* Pattern

Accountability:	Accountability is not affected.
Confidentiality:	The application of this pattern can contribute to the confidentiality of a system if the check algorithm is good (i.e. it detects and stops unauthorized use).
Integrity:	Undesirable modification can be filtered if the check algorithm is capable of detecting those attacks.
Availability:	Denial of Service (DoS) attacks can be prevented, if the Check Point algorithm takes appropriate actions (e.g. delaying or ignoring messages if they match a certain pattern).
Performance:	Complex check routines may slow down the system and its message exchange.
Cost:	The implementation of a security policy is difficult and costly.
Manageability:	Maintenance of security-relevant code will be easier if it is clustered in one location. But the complexity of the checking algorithm could be greater.
Usability:	Some communication activity might be prevented even if it is not harmful. Therefore, the checking algorithm has to be designed careful to not misclassify permissible actions.

Figure 10: Consequences of the *Check Point* Pattern

[9] W. E. McUmber and B. H. C. Cheng. A general framework for formalizing UML with formal languages. In *Proceedings of IEEE International Conference on Software Engineering (ICSE01)*, Toronto, Canada, May 2001.

[10] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63(9), pages 1278–1308. IEEE, September 1975.

[11] M. Schumacher and U. Roedig. Security Engineering with Patterns. In *8th Conference on Pattern Languages of Programs*, July 2001.

[12] J. Viega and G. McGraw. *Building Secure Software - How to Avoid Security Problems the Right Way*. Addison-Wesley, September 2002.

[13] R. Wassermann and B. H. Cheng. Security patterns. Technical Report MSU-CSE-03-23, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, August 2003.

[14] J. Yoder and J. Barcalow. Architectural Patterns for Enabling Application Security, 1997.