Security Patterns

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

Abstract

Design patterns propose generic solutions to recurring design problems. Commonly, they present a solution in a well-structured form that facilitates its reuse in a different context. Recently, there has been growing interest in identifying pattern-based designs 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 system 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 aspects of design. Furthermore, we investigate how verification can be enabled by adding formal constraints to the patterns.

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

Contents

List of Figures

List of Tables

1 Introduction

During the process of software development, expert knowledge in the domain of the implemented system is required. Furthermore, in today's systems with various communication features, security considerations are of greater interest than ever. Therefore, a fair amount of additional security expertise is needed to meet non-functional security requirements. A common approach to overcoming knowledge gaps among developers is to use patterns (analysis patterns [12], design patterns [13], specification patterns [10], etc.). In recent years, pattern-based approaches to software development, applied to different domains, have received significant attention in the software engineering community. In the security domain, it is challenging to capture and convey information in order to facilitate security, which is a very abstract goal. In this report we give a collection of security patterns that have been identified by the community. We use a variation of the design pattern template [13] that better suits the presentation of security-specific 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 is difficult to measure and highly dependent on the environment. Other pattern-based approaches like the well-known *Design Patterns* from Gamma et al. [13] are believed to greatly enhance productivity of the software development process by conveying expertise. Unfortunately, the structure provided by various pattern templates is not sufficient to portray all security relevant aspects. Some approaches [11][27][34] that apply patterns to the field of security use the regular 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 design of security critical systems.

The security needs of a system depend highly on the environment in which the system is deployed. As the pattern approach is not capable of fully covering all possible constellations of security, it is crucial that a developer is provided with information that enables an evaluation of the situation that will lead to the selection of appropriate patterns. By introducing and connecting general security principles with a pattern's substance, the developer gains security insight by reading and applying the pattern. Furthermore, behavioral information and security-related constraints are added in our pattern template. The developer can use this information to check if a specific 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, it promotes the verification of security-relevant properties. Hence, overall security can be improved during the design stages of development by applying this pattern-based approach.

The remainder of this document is organized as follows. Section 2 gives background information for this work including general security principles, a brief description of some UML diagrams, and an overview of pattern-based approaches. The Section 3 defines Security Patterns and introduces our template. Section 4 illustrates in several examples how the pattern template is applied. In Section 5 we show how formal verification is done using two example systems. Finally, Section 6 draws a conclusion and mentions further research areas.

2 Background

This section describes previous work and fundamentals that are relevant to this report. We start with an introduction of ten principles [31] that provide guidelines to more secure system development. The descriptions of *Security Patterns* reference those principles. Next, the selected UML notations that are used in the *Security Patterns* section are briefly overviewed. Finally, we provide a historical perspective of pattern-based approaches that elucidate the pattern approach, especially *Design Patterns*, and explain its application to this work.

2.1 Viega's and McGraw's ten principles

To improve development of secure software Viega and McGraw [31] point out ten guiding principles to achieve better security. They state, in contrast to checklist based approaches, that the use of guiding principles can help to cope with unknown attacks. Although guidelines do not guarantee security, their application can help to prevent common errors during the software development process. Some of the principles exhibited by Viega and McGraw are based on Saltzer's and

Schroeder's [26] eight design principles, which were published in 1975.

The ten principles, outlined in the following section, can facilitate the understanding of particular *Security Patterns* and give security insight. Therefore, they are used in our pattern-based approach and will be referred to as the *Ten Principles*. In accordance with Viega and McGraw [31] we assume that a set of guiding principles, including the following one, cannot be complete in terms of ensuring security. In this context the authors [31] estimate that the *Ten Principles* cover about 90 percent of all potential problems. Further relevant issues are addressed in Section 2.1.11, which describes tradeoffs.

2.1.1 Principle 1: Secure the weakest link.

Intruders usually attack parts of a system that are likely to break. Thus, the level of security in a software system is determined by its weakest components. In order to improve system security, possible weaknesses must be identified and strengthened until the risk of violations can be considered acceptable. Viega and McGraw [31] indicate that a system's users and administrators can also be a major vulnerability. They might easily become victims of social engineering[1] if the current security policy does not take into account those attacks.

2.1.2 Principle 2: Practice defense in depth.

An exception to the first principle are overlapping security mechanisms. If more than one protective measure exists in a system, then the level of security is not necessarily determined by the weakest part. Every additional protection layer may contribute to system's security. Viega and McGraw [31] propose in their second principle the use of several security layers to create a more effective defense against attacks.

2.1.3 Principle 3: Fail securely.

Security flaws are often inherent to system failures. Unfortunately, failures cannot be avoided completely in complex software systems. Thus, Viega and McGraw [31] point out that it is even more important to plan failure modes and assure that a system's security is not compromised by

[1] Social engineering is a non-technical kind of attack that exploits human weaknesses. An exemplary scenario consists of an intruder contacting authorized users in order to make them reveal information or take actions that compromise the security [32].

exceptional behavior. Some systems perform insecure operations during failure modes in order to provide certain functionality or to maintain compatibility with old standards. This common practice of code violates the 'fail securely'-principle. An attacker might find a way to trigger the insecure system failure and take advantage of its behavior.

2.1.4 Principle 4: Follow the principle of least privilege.

In 1975 Saltzer and Schroeder [26] published the principle of least privilege in a paper on protection of information in computer systems. Their work states that every entity in a system should be granted only the minimum set of permissions needed to perform its designated tasks. In complex systems, especially if many privileged users exist, we can assume that misuse of rights may occur. Thus, it is reasonable to apply this guideline in order to limit the impact of unintended use of rights. When Viega and McGraw [31] adapted this principle, they complained about software developers that violate principle 4 just for the reason of laziness. Granting full access to an entity is much more convenient than to determine exactly which interaction with other system parts is permissible. Thus, some programmers violate principle 4 and grant more rights than necessary.

2.1.5 Principle 5: Compartmentalize.

The principle of compartmentalization aims for a similar goal as the principle of least privilege: it tries to minimize the damage an attack can cause. Viega and McGraw [31] recommend segmenting a system into several components that can be protected independently. Thus, a security breach in a smaller entity would not affect other parts of the system. Unfortunately, those systems with independent secured parts are more difficult to program and often result in increased administrative work.

According to principle four: *Follow the principle of least privilege*, compartmentalization can provide a finer grain for granting access rights and allows a more restrictive access model to be implemented.

2.1.6 Principle 6: Keep it simple.

Viega and McGraw [31] promote the idea that one should keep a system as simple as possible in order to avoid unnecessary complexity. This objective is fairly simple and not new to the field of computer science. The required effort to understand a system grows with a system's complexity. Saltzer and Schroeder [26] named this principle *economy of mechanism.* Furthermore, they point out that undesirable access paths can be discovered more easily in a plain, simple system.

Viega and McGraw state that usability is a significant part of simple design. With respect to security, they consider it even more important than in any other domain because many security-relevant actions depend on user decisions. Saltzer and Schroeder take into account this aspect in their principle of *psychological acceptability.* They state that the human interface has to support the application of protection mechanisms.

Some principles exhibited here contradict each other if they are applied to the greatest possible extend. Thus, it is necessary to evaluate to which extend the application of a principle is reasonable according to your specific system. If the ultimate goal is simplification then defense cannot be practiced in depth, because it would increase the system's complexity. Finding an appropriate combination of the principles can be a challenging task if a system of good security is to be designed.

2.1.7 Principle 7: Promote privacy.

Promoting privacy consists of two main objectives. First, the amount of information that can be gathered about a system and its users should be minimized. This objective will make the system a harder target and protects the users' privacy. Second, misinformation can be used to lead attackers to wrong assumptions and complicate malicious efforts.

Viega and McGraw [31] point out that there is a tradeoff between holding information back and usability. For security purposes users might need to reenter data a system could provide by itself.

2.1.8 Principle 8: Remember that hiding secrets is hard.

Usually a system's security depends on certain secrets being kept. Once private keys, passphrases, or secret algorithms are revealed, security can easily be compromised. Protecting such secrets is a difficult task. Even storing critical information in binary form does not prevent being uncovered. Some attackers have sufficient reverse engineering capabilities to analyze and understand machine-readable code. Viega and McGraw [31] suggest being suspicious even of known entities and to take into account insider attacks.

2.1.9 Principle 9: Be reluctant to trust.

Some security violations are possible because system developers extend trust unnecessarily. A more pessimistic point of view during system design can help to recognize weak points. Viega and McGraw [31] advise programmers to design systems whose parts mistrust each other. Furthermore, one should not rely on the security of off-the-shelf solutions, but rather practice defense in depth. Hiding secrets in client code is not desirable either because skilled users might be able to extract information and abuse that knowledge.

2.1.10 Principle 10: Use your community resources.

According to Saltzer's and Schroeder's [26] principle of open design, Viega and McGraw [31] state that community resources usually are more secure than routines written by individuals. Widely distributed programs have been executed and tested many times and therefore it is very unlikely that unknown errors and security weaknesses exist. Saltzer and Schroeder point out that decoupling the protection algorithms from keys enable public scrutiny of protection mechanisms, thereby potentially improving their overall capability. Furthermore, the protection of a key is more practicable than keeping an algorithm secret.

2.1.11 Tradeoffs

The preceding principles contain certain contradictions. Thus, it is important to take into account a principle's context of application. The remainder of this section outlines some tradeoffs that are likely to occur.

Principle 5 suggests compartmentalization as a means of minimizing possible damage. Frequently additional code is necessary to manage and protect many compartments in a system. This increase in effort and code is contradictory to principle 5, which states that redundancy should be avoided to keep a system simple. Practicing defense in depth (principle 2) diverges from the *Keep it simple* principle in a similar fashion [31]. Another tradeoff exists between *Keep it simple* (principle 6) and *Promote privacy* (principle 7) since usability as a part of principle 6 is constricted by promoting privacy. Making user information available in the internet/intranet may be convenient for some kind of application and improve usability but it contradicts principle 7.

The principles in this section may not be helpful at all times nor do they give answers to concrete design problems. Nevertheless they can further the understanding which aspects are relevant to secure software development. Additionally they convey the message that security is an important factor of today's software development.

2.2 The Unified Modeling Language (UML)

This section gives a brief introduction of the Unified Modeling Language (UML) covering notations used in this paper. With respect to the intent of this paper, we do not give a detailed description of UML or object-oriented design approaches (a variety of books cover those topics, including [5, 25]).

UML is a registered trademark of the Object Management Group (OMG). Currently, OMG publishes the UML specification as of version 1.5 [24]. UML is a widely used language to express and capture structural and behavioral aspects of software systems. During the last years it evolved to a powerful and broadly known set of notations, that is capable of supporting software engineers during various steps of software development. In the UML User Guide [5] Booch, Rumbaugh, and Jacobson point out that the language is appropriate for visualizing, specifying, constructing and documenting artifacts of a software-intensive system.

The Unified Modeling Language consists of three elementary building blocks: things, relationships and diagrams. Basic elements are called *Things.* They can be connected by relationships that convey a certain coherence, depending on the type of relationship. Diagrams embody the third building block and provide means for grouping useful elements of UML together. The remainder of this section presents three different diagrams: First, we outline class diagrams that are used to model structural aspects. Afterwards, we briefly explain sequence and state diagrams that encompass behavioral aspects. All presented types of diagrams show different views of a system. In combination with each other they can further the understanding of a system as whole.

2.2.1 UML class diagrams

The UML class diagram groups things and relations that depict the static structure of a software system. In this graphic notation classes represent abstractions of structural elements of a system. They encompass the things that can be tied together by different types of relations. A class defines a group of objects that have similar properties and behavior [25]. In terms of object-oriented design, properties are called *attributes* and the behavior is determined by *operations.* Figure 1 depicts the graphical representation of a class including attributes and operations.

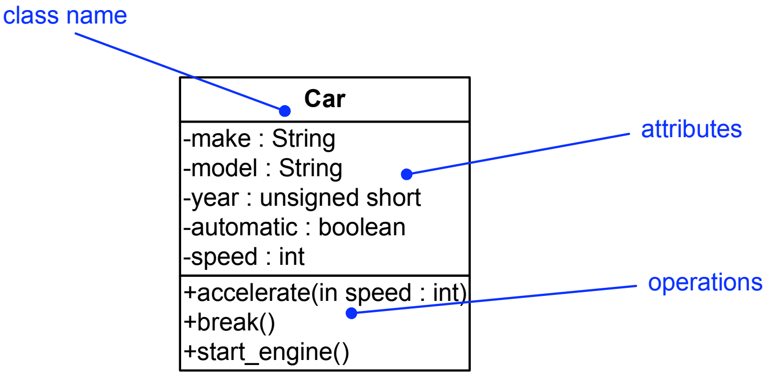


Figure 1: UML Notation: Class representation

Besides classes, the class diagram contains several types of relationships that can be used to connect classes and convey information about their structural relation. We distinguish among three types of relationships: dependencies, generalizations, and associations. Dependencies denote one class might be affected by changes in an other class. Dependencies are directed and do not necessarily imply the opposite direction. They are displayed by a dashed arrow that points to the class that the other depends on. A generalization relationship expresses that one class is a special case of a general class. The generalization is rendered as arrow with a white head pointing to the general class. Associations are the third type of relationships and indicate that instances of involved classes are connected in some way. An association is depicted by a plain line. Frequently a text tag is added to explain the relationship's meaning. Furthermore, the UML notation includes aggregation and composition as a special associations that express "whole/part" relationships [5]. They are rendered by a line with a diamond-shaped head. The composition expresses a stronger relationship than the aggregation and has a solid-filled head. In a composition, parts depend on the existence of the whole (their composite). Figure 2 shows an example UML class diagram illustrating the different types of relationships.

Aggregation and generalization significantly differ from each other. The aggregation involves two different objects and states information about the relationship of instances, whereas the generalization addresses class level and specifies the relation of classes. At the instance level the generalization relation refers to only one object that has the characteristics of all classes in its

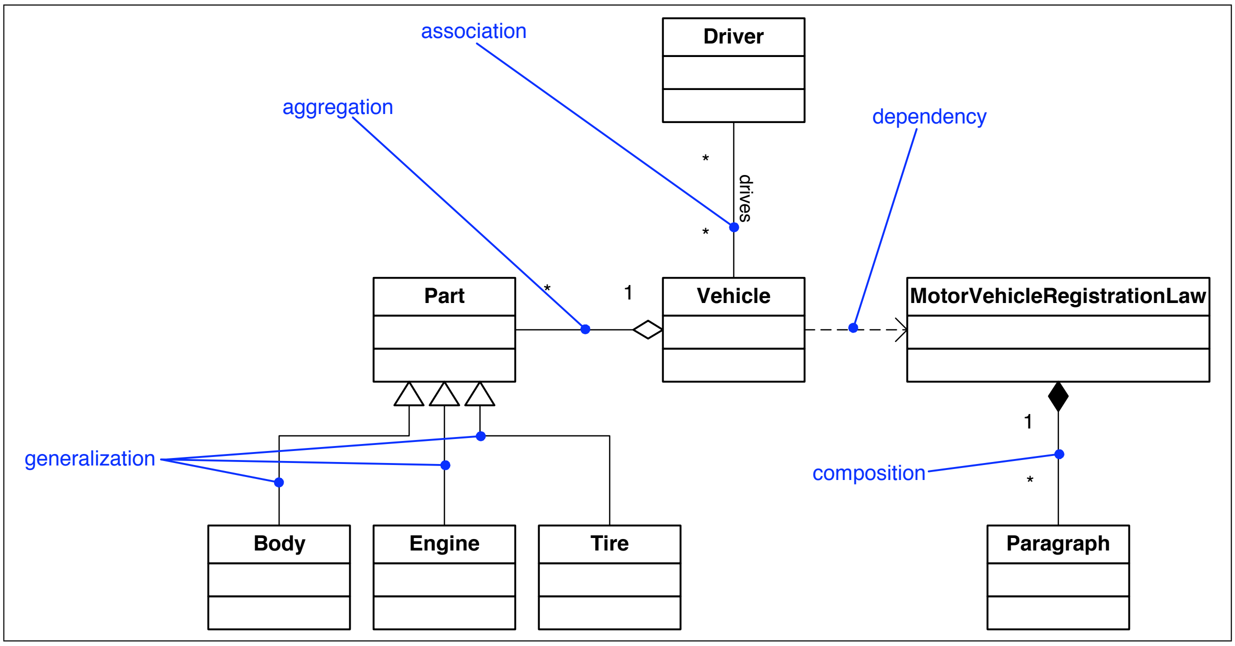


Figure 2: UML Notation: Relationships in class diagrams

inheritance hierarchy.

2.2.2 UML sequence diagrams

Sequence diagrams depict behavioral aspects of a system. They are useful to model or visualize the control flow among objects. Those objects are rendered as rectangles and each object is connected to a vertical dashed line that represents its lifetime. The object name and class are separated by a colon and placed inside the rectangle. Labelled arrows between the different lifelines depict messages that are arranged in order of their time sequence. Figure 3 visualizes an example scenario in an UML sequence diagram.

The example shows the message sequence that is initiated by a student's application for a study abroad. The diagram visualizes the interaction of the involved entities.

2.2.3 UML state diagrams

Unlike sequence diagrams, state diagrams do not depict just scenarios of a software system. A state diagram is capable of completely specifying the behavior of a class, represented by states and transitions. A state embodies a situation or condition during an object's lifetime [5] whereas

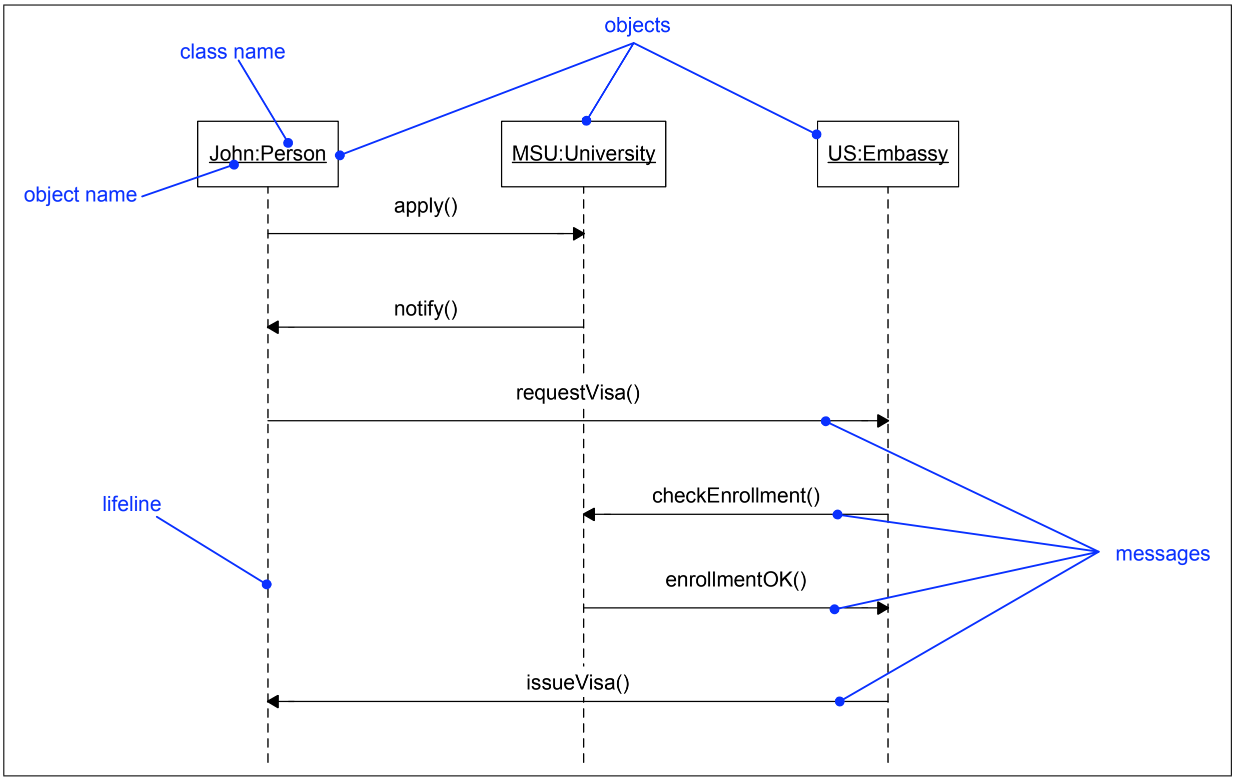


Figure 3: UML Notation: Sequence diagram example

transitions define the conditions of state changes. A state's graphic representation is a rounded rectangle, containing its name. Furthermore, a state is capable of triggering actions, which are denoted in the lower part of its rectangle. According to the UML specification [24], a transition is determined by a *source state*, an *event trigger*, a *guard conditions*, *actions*, and a *target state*. A transition and thereby a state change is only executed if its source state receives the event that corresponds with the event trigger and guarding condition is met. Before the target state is activated, the transition triggers its defined actions. Each transition is depicted by an arrow that connects source and target state. Figure 4 gives an example of a state diagram and its elements.

2.3 The Pattern Approach

This section clarifies what patterns are and why they are useful. After a brief presentation of the pattern approach, we focus on design patterns and the different types of templates that are mainly used. Finally, we briefly describe *Anti-Patterns*.

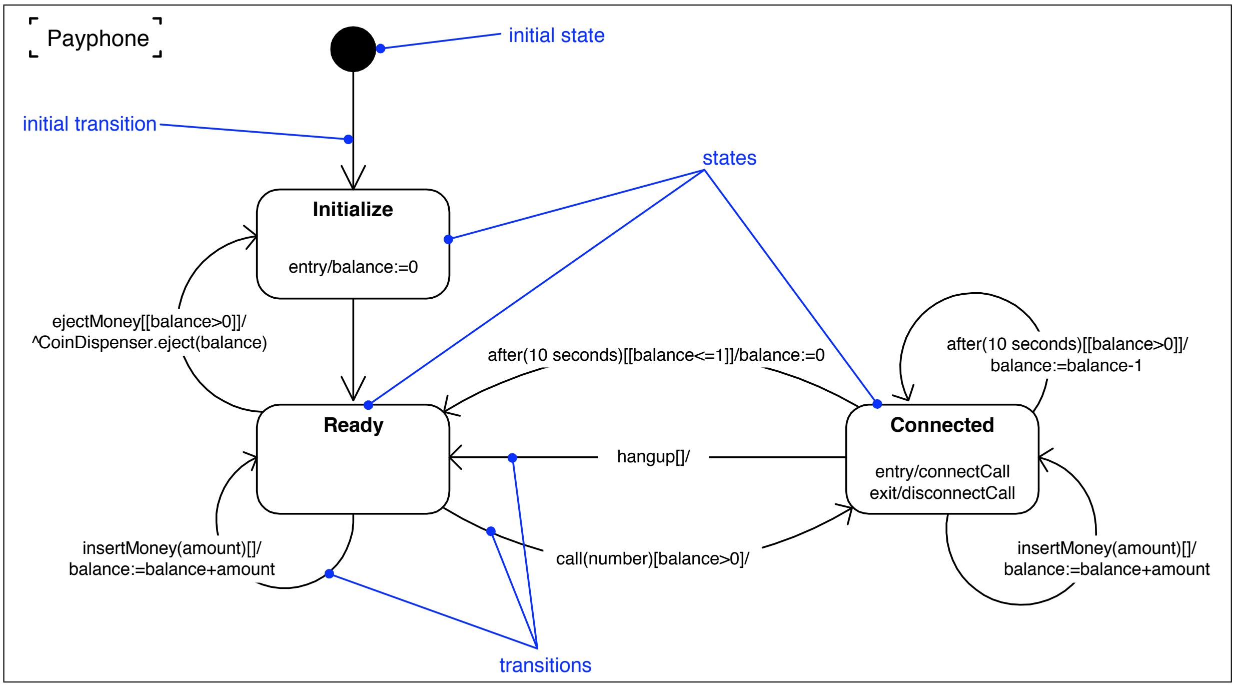


Figure 4: UML Notation: State diagram example

The first person who used the pattern approach was Christopher Alexander [2]. In his book: *A pattern language: towns, buildings, construction*, he gave a succinct definition of what a pattern is:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
>
> *Christopher Alexander [2], Page x*

Even if Alexander meant to describe the architectural problems that accompany the planning and construction process of towns and buildings, his brief definition is applicable to many different areas. The computer science community finally became aware of patterns as means of coping with recurring problems when Gamma et al. (who are also referred to as the *Gang of Four (GoF)*) [13] published their book *Design Patterns: Elements of Reusable Object-Oriented Software*, in 1994. Since then, the pattern approach has been applied in many different realms. Fowler [12] stated, in his book on *Analysis Patterns*, that the GoF had a greater impact on software patterns than Alexander.

In recent years, the pattern approach obtained increasing popularity. Several factors account for the acceptance among members of the computer science community. Patterns improve and simplify communication as they extend the existing terminology. A pattern name is associated with a specific problem and its solution. Furthermore, expert knowledge of a specific domain can be captured in a pattern. Structures that experienced designers intuitively apply can be conveyed to novices. Additionally, the use of patterns unifies design and thus improves comprehensibility.

Gamma et al. transferred Alexander's patterns to describe solutions to software design problems in terms of objects and coined them *Design Patterns*. The GoF point out four mandatory elements of a *Design Pattern*: *pattern name*, *problem*, *solution*, and *consequences.* Although those sections embody the core of a pattern the captured information can be structured in various ways. Thus, several *Design Pattern* formats can be found. The most common templates are the one presented by Gamma et al. [13] and the *canonical* template, which is currently used by a large software company (AG Communication Systems (AGCS) [29]). Table 1 gives an overview of the different sections of those formats and how they relate to each other. It is based on information that was presented by Brad Appleton [3] in his report on *Patterns and Software: Essential Concepts and Terminology.* The different elements that we adapt into our *Security Pattern Template* will be explained when used in Section 3.2.

Unlike regular patterns that point out desirable solutions to recurring problems, *Anti-Patterns* depict common design errors and pitfalls. They describe structures that are unfavorable for certain reasons. Besides just indicating unsuitable structures, *Anti-Patterns* can also give advice how to avoid or correct design shortcomings. Similar to regular patterns, *Anti-Patterns* capture experience.

3 Security Patterns

This section is organized in two subsections: Firstly, Section 3.1 outlines terminology and previous work that is inherent to *Security Patterns* and relevant to this paper. Secondly, Section 3.2 presents our *Security Pattern Template*.

3.1 Security Patterns and previous work

In recent years the pattern approach evolved to a widely-used means of solving generic problems and facilitating reuse. Thus it is not surprising that pattern research has been done in many domains

Canonical Template	**GoF Template**
Name	Name, Classification
Aliases	Also Known As
Problem	Intent
Context	Applicability
Forces	Motivation
Solution	Structure, Participants, Collaborations, Implementation, Sample Code
Resulting Context	Consequences
Rationale	-
Known Uses	Known Uses
Related Patterns	Related Patterns
Sketch	-

Table 1: Canonical Pattern Template versus GoF Template

including the security domain.

In 1997 one of the first papers addressing security patterns was published by Yoder and Baralow [34] *Architectural Patterns for Enabling Application Security.* Till today many other papers on that topic are available. Braga, Rubira, and Dahab [6] developed in 1998 a pattern language for cryptographic software. Fernandez [9, 11] posted two papers in 2000 and 2001 presenting some security patterns. In 2001 Schumacher and Roedig [27] publish a security pattern template in their work *Security Engineering with Patterns.* In 2002 Kienzle et al. [16] present a security pattern template.

We quote Schumacher and Roedig [27] for definitions what security patterns and security pattern systems are. Their understanding is similar to the definitions found at www.security-patterns.org [33] and reflects our comprehension of *Security Patterns*:

> "A *Security Pattern* describes a particular recurring security problem that arises in specific contexts and presents a well-proven generic scheme for its solution. A *Security Pattern System* is a collection of security patterns, together with guidelines for their implementation, combination and practical use in security engineering."[27]

With all the previous work in the domain of *Security Patterns* we still think that improvements and extensions to the current security pattern template are possible. Some approaches [34, 9, 11, 27] use the regular or an only slightly modified Design Pattern [13] template. A specifically tailored pattern template for the security domain is necessary. Our template adds structural and behavioral aspects to the template using UML class- and state-diagrams. Furthermore, developers are provided with security insight using links to the Viega's and McGraw's [31] ten principles. Finally, adding security-related constraints enables the use of formal verification techniques. A similar approach in the domain of requirements engineering lead to valuable results [20, 17, 19].

3.2 Security Pattern Template

The following template for Security Patterns is derived from the template of the well-known Design Patterns[2] from Gamma et al. [13]. Some sections have been altered to convey more security-relevant information than the original template. Other parts are completely new (Behavior, Constraints, Supported Principles, Community Resources) and provide additional information that has not been captured or been relevant to common design patterns. The approach depicted below facilitates reuse of security specific knowledge.

- **Pattern Name and Classification**

  In accordance to Gamma's template, the Name is a *primary key*[3] to the pattern. It should be self-explanatory and intuitive in order to improve communication among designers.

- **Intent**

  The intent section describes briefly for what this pattern is used. It names security problems that can be solved by the application of this pattern. It may also introduce the pattern as an elegant way of improving a system according to the *Ten Principles*.

- **Also Known As**

  If synonyms exist for this pattern, then they should be named in this section.

- **Motivation**

  This part describes security problems (or violations of the *Ten Principles*) that are addressed

[2] See Section 2.3 for more information on Design Patterns

[3] Usually *primary key* is a term used in database design. In a relational table the *primary key* unambiguously identifies each record (row) of a table. We use this term in a similar fashion to identify patterns.

by the application of this pattern. Furthermore, the motivation section should show the basic ideas of the pattern in an illustrative way and convey how the mentioned problems are solved. A good practice is to quote a fitting example of the patterns application.

- **Applicability**

  Describes the context in which the pattern can be used. Under which circumstances should the pattern be applied? Does it address application-level, host-level or network-level security?

- **Structure**

  The structure section uses UML class diagrams to give an overview of the static components used in this pattern.

- **Participants**

  Describes the different classes or objects depicted in the structure section.

- **Collaborations**

  A textual description of the interaction among the participants and how they perform their different tasks.

- **Behavior**

  To illustrate the pattern's behavior and interaction more formally, UML state and sequence diagrams are used to depict the dynamic aspects of this pattern.

- **Constraints**

  The constraints section describes properties that must be fulfilled at all times by the implementation of the pattern. We enrich the template with constraints in order to facilitate verification tasks. The goal of this approach is to provide an efficient way to rigorously check system design and code during the process of software development. Our idea is that the application of the pattern leads to a formal model that can be used to automatically check your system against the properties presented herein.

  In order to categorize the constraints, their function may be named along with them. Frequently occurring security-related classes of constraints are:

  - Availability,
  - Authenticity,

- Confidentiality,
- Integrity.

- **Consequences**

  How does the pattern cope with the outlined problems. What further information becomes relevant upon the application of the given pattern. How is the current system and its security affected? Which side-effects arise from the pattern's use? Kienzle et al. [16] propose the following template as a means to categorize consequences. The *Security Patterns* outlined herein make use of this criteria to depict tradeoffs and outcomes systematically.

Accountability:	According to the National Security Telecommunications and Information Systems Security Committee, accountability describes a system's ability to determine the responsible source for activities [30]. How does the application of the pattern impact the system's capability of accomplishing those tasks? How is, for example, logging or authentication affected?
Confidentiality:	The assurance that information is not accessed by unauthorized parties is termed Confidentiality [30].
Integrity:	Which impact has the application of the pattern on the protection of information against malicious modification or destruction.
Availability:	Availability assures that authorized users can use a system's resources when required [30]. How does the pattern support or limit availability?
Performance:	Which impact has the pattern on a system's performance? Does it slow it down or improve working speed?
Cost:	Which expenses accompany the application of the pattern?
Manageability:	How is the management and maintenance of the system affected?
Usability:	Which changes coming along with the implementation of the pattern might be perceived by an user?

- **Implementation**

  What are issues of concern, when the pattern is implemented?

- **Known Uses**

  Which systems have implemented this pattern?

- **Related Security Patterns**

  Are there any other *Security Patterns* that may be used with this one? Is it part of a *Security Pattern System*[4]?

- **Related Design Patterns**

  Which *Design Patterns* can be used to realize the pattern? How can they contribute to system security? Which of the *Ten Principles* do they support?

- **Supported Principles**

  Which of the *Ten Principles* are supported by this pattern and which are violated (if any).

- **Community Resources**

  Names of libraries, exemplary implementations or other resources that support the application and implementation of the pattern, if any.

4 Examples of Security Patterns

This section contains examples of *Security Patterns.* Table 2 outlines the patterns that are described in this section. Similar to *Design Patterns* [13] we can classify them into the categories *Creational*, *Structural* and *Behavioral* patterns. Furthermore, their abstraction level defined in terms of *Application-level*, *Host-level*, and *Network-level* are shown. In order to give a high-level perspective on the purpose of each pattern, their relation to the *Ten Principles* is pointed out.

The patterns *Single Access Point*, *Check Point*, *Roles*, *Session*, *Full View With Errors*, *Limited View* are based in part on patterns by Yoder and Barcalow [34] on *Architectural Patterns for Enabling Application Security.* Yoder and Barcalow focused on creating a framework that improves application development. Therefore, they presented the named patterns in *canonical form*[5].

[4] The term *Security Pattern System* is defined in Section 3.1

[5] pattern format, explained in Section 2.3

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

Our approach shows that some of their patterns may be applicable beyond the scope of application security. To convey more security-related information we use our augmented *Security Pattern* template introduced in Section 3.2 on page 14.

Fernandez [11] also introduced a few security-related patterns. His *Authorization*, *Role-Based Access Control (RBAC)* and *Multilevel Security Pattern* refer in part to Yoder's and Barcalow's work, mentioned above. As *Security Patterns* on a relatively high abstraction level, Fernandez' patterns are a good addition to our overview.

Note that Fernandez' *RBAC* and Yoder's and Barcalow's *Roles* describe the same pattern.

4.1 Single Access Point

The Single Access Point (SAP) pattern was first used by Yoder and Barcalow in 1997 [34]. While they concentrated their work on a framework for application development, the basic idea for SAP is also appropriate beyond that scope. The concept of limiting extraneous access to a single channel in order to facilitate control may be used in any self-contained system that communicates with others.

- **Pattern Name and Classification**

  The following *Security Pattern* is commonly known as *Single Access Point.* It provides a scheme for the static design of a system. Hence, it can be considered a structural *Security*

Pattern.

- **Intent**

  The Single Access Point pattern defines one single interface for all communication with system external entities in order to improve control and monitoring.

- **Also Known As**

  The Single Access Point pattern is also referred to as [34]

  - Guard Door,
  - Login Window,
  - One Way In or
  - Validation Screen.

- **Motivation**

  Due to various access points, many systems cannot be protected effectively against attacks from the outside. Hidden back doors and different (inconsistent) implementations of security policies aggravate protection. The application of the Single Access Point pattern prevents external entities from communicating directly with components in the system. All inbound traffic is routed through one channel, where monitoring can be performed easily. Additionally, the Single Access Point is an appropriate place for capturing an information log on the parties currently accessing the system. This data may be useful inside the system to verify certain access requests and to determine their rights.

- **Applicability**

  The Single Access Point may be applicable to self-contained systems that need to communicate with external entities. It can be used at the application-level as well as at the host or network-levels, even though implementation at the abstraction-level of application development is more apparent at first glance. Application at the network-level implies that all sub-nets inside the system's boundaries are isolated from other nets. The only connection to the outside is a Single Access Point. While we assume virtual isolation of system internal entities in high abstraction levels, lower design levels have to include this goal in their models (for example by adding encryption, signing of messages, and tokens that guarantee freshness). The system's deployment structure determines where further securing effort is necessary.

Yoder and Barcalow [34] describe the following forces that result from the implementation of a Single Access Point. It is obvious that this pattern cannot be used with systems that need several entry points in order to provide greater flexibility. Furthermore, multiple access points alleviate adaptation in different environments. Each of them could be customized to request only the authorization information required to execute the current operation - a Single Access Point that grants access to several parts of the system might ask for information that will not be used during the session.

- **Structure**

  The Single Access Point represents the system's only connection to the outside. All incoming communication requests are passed to the Single Access Point instance. From there, they will be directed to the intended recipient, if all security-relevant requirements are met. The UML class-diagrams depicting those structural relationships are shown in Figures 5 and 6. The rectangular box depicts the system's boundaries. The class *Single Access Point (SAP)* is the only one that interacts with external entities. The first example (Figure 5) is a simplified model of an online store. Requests from outside (e.g. from known customers, a store manager or anonymous guests browsing the online store) are directed to the SAP and then forwarded to the internal class. An example message sequence could be a query that returns a set of available products in the store. The second example (Figure 6) depicts a medication information system that might be used in a hospital. Both application domains show how external entities have to request information from the system via the SAP.

- **Participants**

  - External Entities

    - ∗ are components located outside the system's boundaries. They contact the SAP in order to communicate with internal entities.

  - Internal Entities

    - ∗ are all components located inside the system's boundaries.

  - Single Access Point

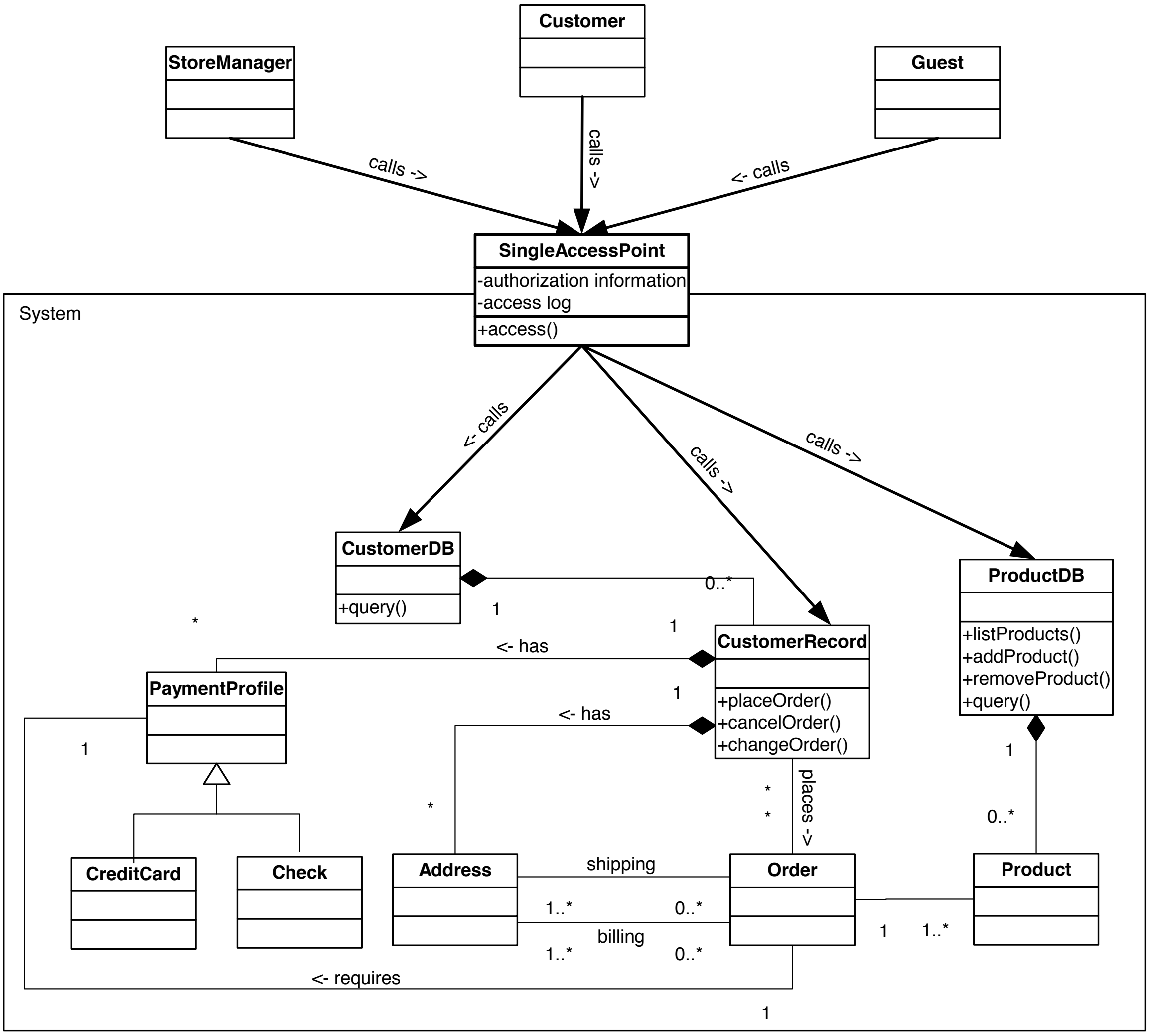


Figure 5: UML class-diagram: SAP Example: Online store

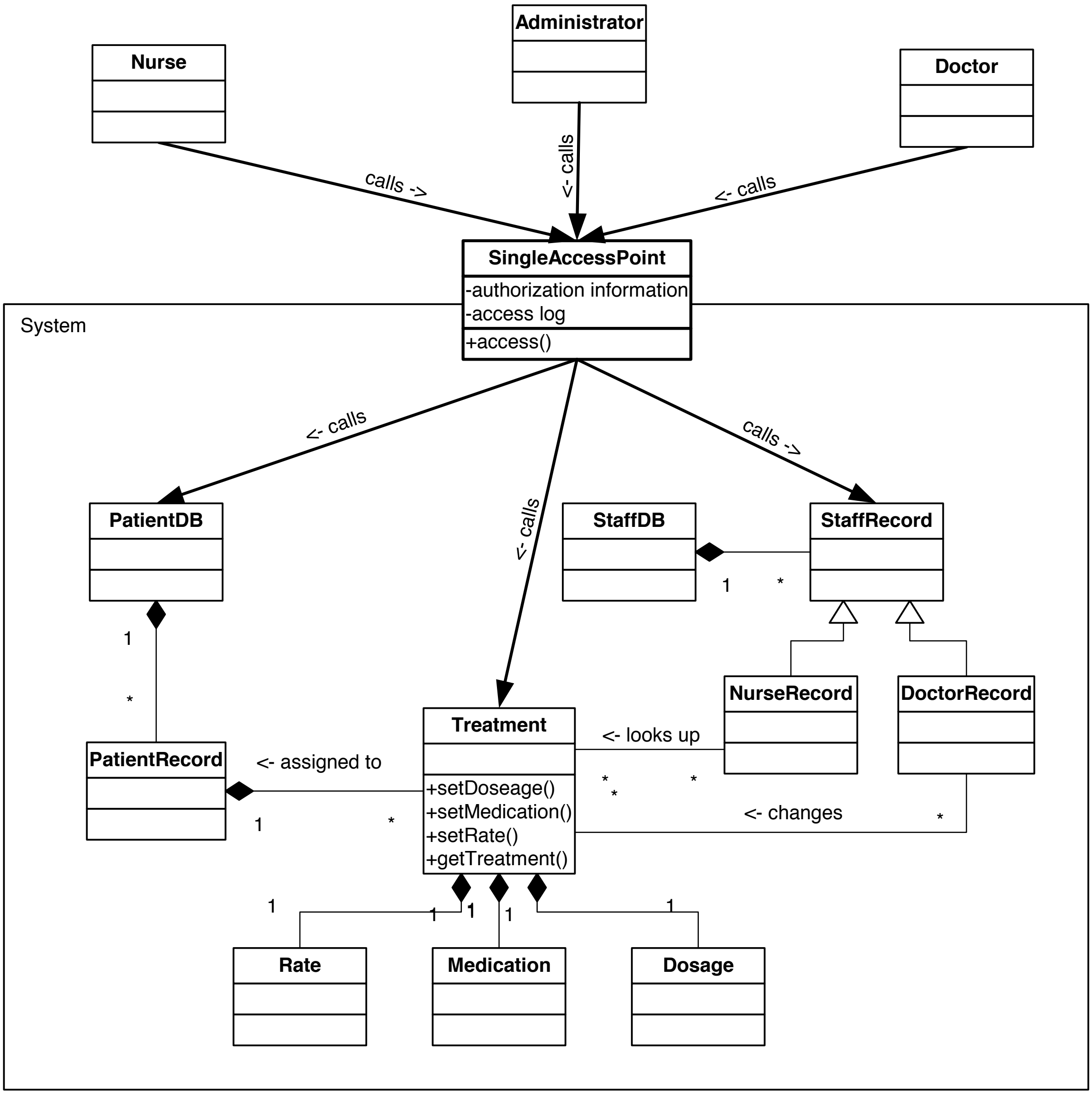


Figure 6: UML class-diagram: SAP Example: Medication system

  - ∗ provides an interface that allows external parties to communicate with system internal components.
  - ∗ gathers information about the occurring access requests, their origin and authorization information.
  - ∗ triggers actions or forwards data to parties inside the system.

- **Collaborations**

  The SAP class interacts with any class that needs to communicate with system external parties. It works as a mediator. If certain security policies need to be enforced, all requests might be sent to a Check Point[6] class before they are forwarded to their intended recipients.

- **Behavior**

  The following examples relate to the online store class diagram in Figure 5. The first UML sequence-diagram in Figure 7 shows how an external request is routed to an internal component by the Single Access Point. If the recipient is unknown or not available the SAP can be implemented to send an error message back to the calling instance. This sequence of events is depicted in Figure 8.

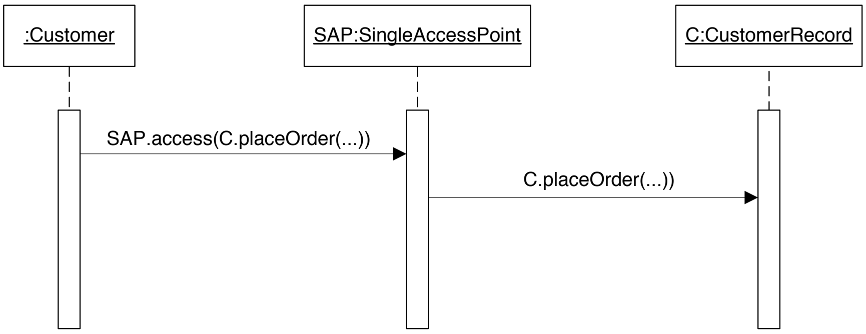


Figure 7: UML sequence-diagram: Single Access Point: Recipient available

[6] compare *Check Point Security Pattern* in section 4.2 at page 26

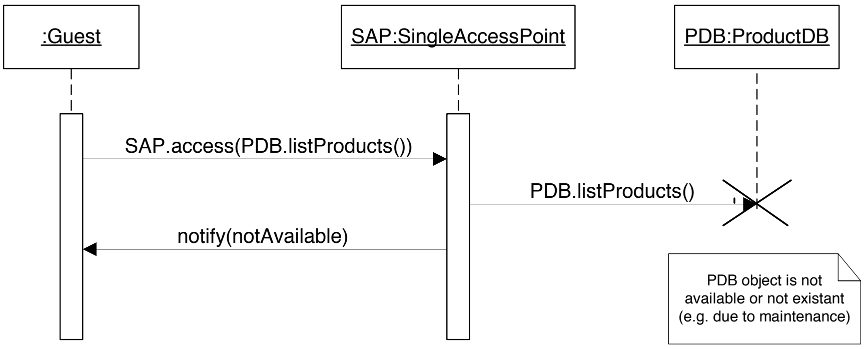

:Guest
SAP:SingleAccessPoint
PDB:ProductDB
SAP.access(PDB.listProducts())
PDB.listProducts()
notify(notAvailable)
PDB object is not available or not existant (e.g. due to maintenance)

Figure 8: UML sequence-diagram: Single Access Point: Recipient not available

- **Constraints**

Authenticity	A message that is directly sent to an internal component originates either from a system internal component or the SAP.
Availability	If the internal recipient to an external request is not available an error message is sent to the calling external entity.
Availability	If external entities send more requests to the SAP than usual their forwarding can be delayed or skipped in order to maintain availability of service for other entities.
Confidentiality	Communication between internal components is not disclosed to outside entities.
Integrity	Messages inside the system or from the SAP to an internal component cannot be modified by external entities.
Internal Existence	If the internal recipient to an external request is known it is forwarded.
Internal Existence	If the internal recipient to an external request is unknown an error message is sent to the calling external entity.

- **Consequences**

Accountability:	The accountability is improved by application of this pattern. Single Access Point provides a good place to capture logging information and perform authentication tasks.
Confidentiality:	Confidentiality can also be improved as disclosure of information to unauthorized parties is more unlikely. Every access will pass the Single Access Point and can be monitored.
Integrity:	Undesirable modification of data can be prevented better by efficient checks who is allowed to access the system.
Availability:	Availability may be reduced if the Single Access Point cannot handle all accesses concurrently. Denial of Service (DoS) Attacks can be prevented more efficiently. All information that is necessary for their detection can be gathered at the Single Access Point.
Performance:	If substantial checking and logging tasks are conducted at the Single Access Point, system performance can be affected. Multiple Access Points would provide more flexible and easier ways to communicate.
Cost:	Depending on the extent of communication with external parties, the development can be more difficult and expensive.
Manageability:	Maintenance of security-relevant code will be easier as it is not scattered over the entire system.
Usability:	In order to access the system, an user might have to enter more information than the authentication of the specific request would require. Since the interface, provided by the SAP, has to conduct all possible communication with external entities, it may be unified. This unification may lead to a more inconvenient login process.

- **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 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**

  Singleton, Facade, Wrappers

- **Supported Principles**

  The main idea of the Single Access Point pattern is the principle *keep it simple* (Principle 6). Security-relevant code for the examination of system accesses is not any longer distributed over the whole system. If interfaces to external components are considered potential weaknesses, then the pattern also *secures weak links* (Principle 1). Furthermore, the implementation of a central monitoring instance indicates a certain *reluctance to trust* incoming messages and queries (Principle 9).

- **Community Resources**

  To be determined.

4.2 Check Point

The Check Point pattern that is explained herein is also part of Yoder's and Barcalow's framework for application security [34]. The Parts *Structure*, *Participants*, *Collaborations*, *Constraints*, *Consequences* and *Supported Principles* contain new information. The other sections were based in part on 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 [34]

  - 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 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 performed in a specifically designated component. It is called Check Point.

- **Applicability**

  Check Points are applicable in 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 checkpoint 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 at a place where it can intercept a major part of the traffic. Hence, a Single Access Point is predestinated to be combined with a Check Point - all messages will be monitored. This structure is depicted in Figure 9.

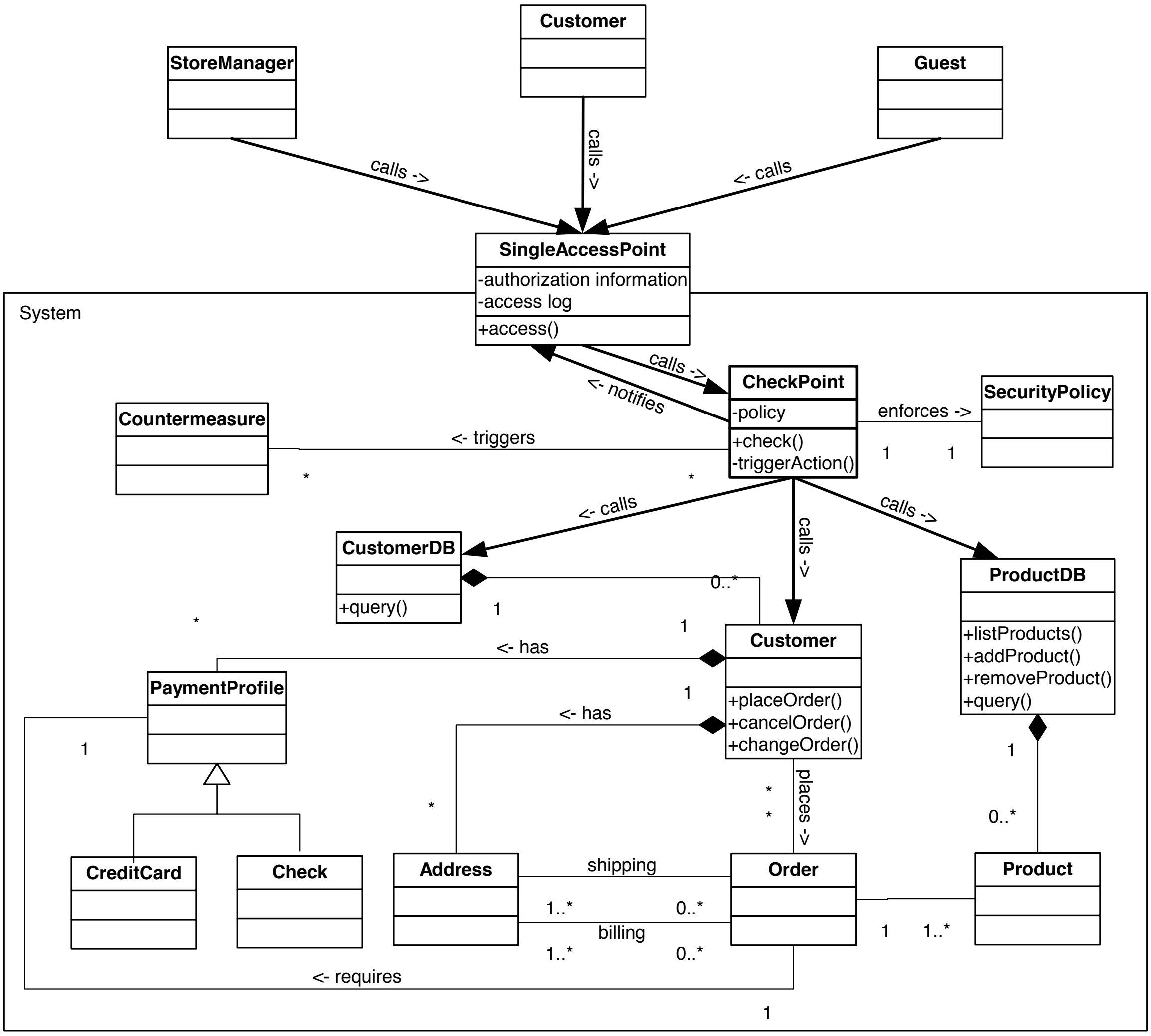


Figure 9: UML class-diagram: Check Point

- **Participants**
  - Check Point
    - ∗ 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 are considered innocuous.
  - Countermeasure

    - provides a set of actions that can be triggered in order to react to an access violation.

  - Security Policy

    - object that implements the rules that are applied to determine wether an access or condition is allowed or not.

- **Collaborations**

  The Check Point class monitors the message flow between other components. The calling class requests the Check Point to scrutinize a certain message. To determine wether a request is entitled or not, the *Security Policy* 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 revealed then the Check Point triggers appropriate counteractions (implemented in a *Countermeasure* object).

- **Behavior**

  The UML sequence-diagrams in figure 10 and 11 show two possible scenarios. The first one depicts the message flow, if access is granted to an external request. The second one shows a message being rejected by the Check Point. Concrete examples of message interaction are shown in Figures 12 and 13.

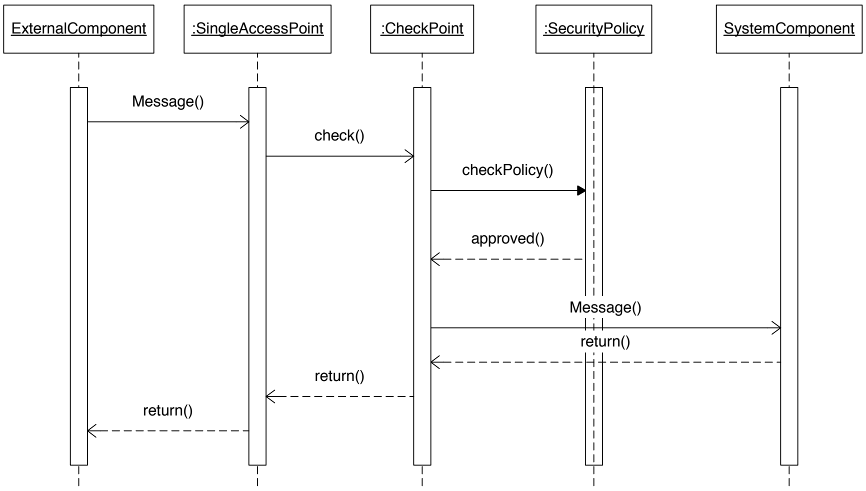


Figure 10: UML sequence-diagram: Check Point: access granted

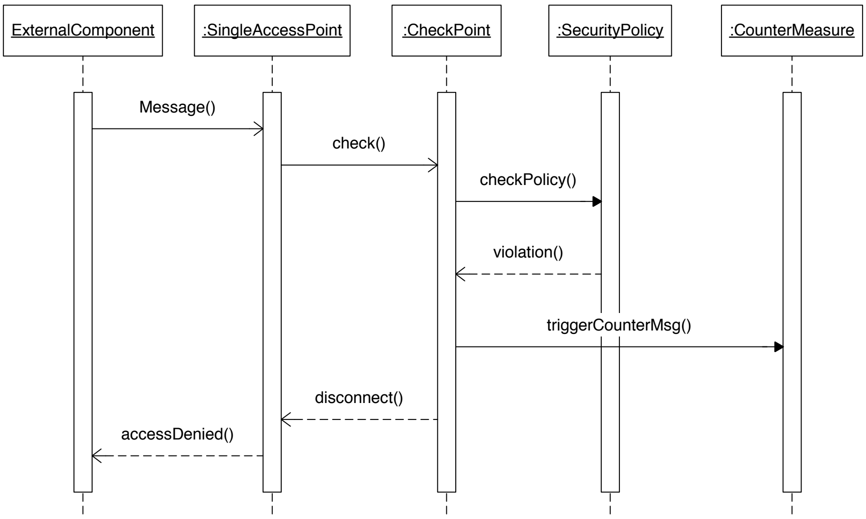


Figure 11: UML sequence-diagram: Check Point: access denied

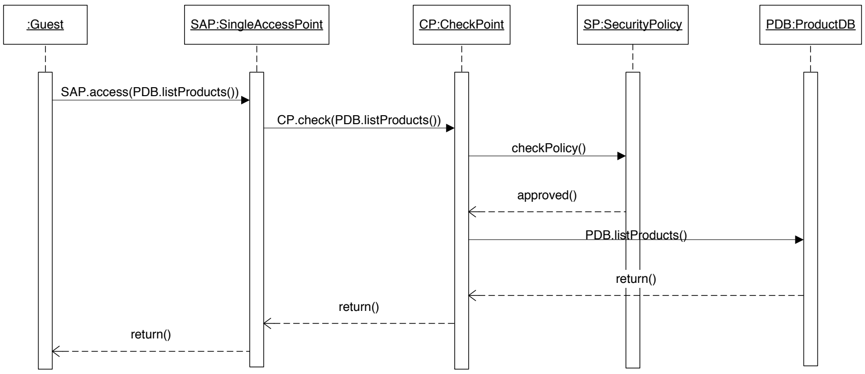


Figure 12: UML sequence-diagram: Check Point: Example 01

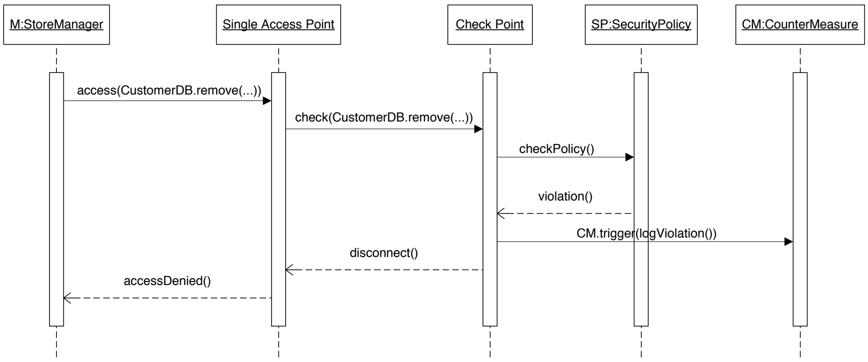


Figure 13: UML sequence-diagram: Check Point: Example 02

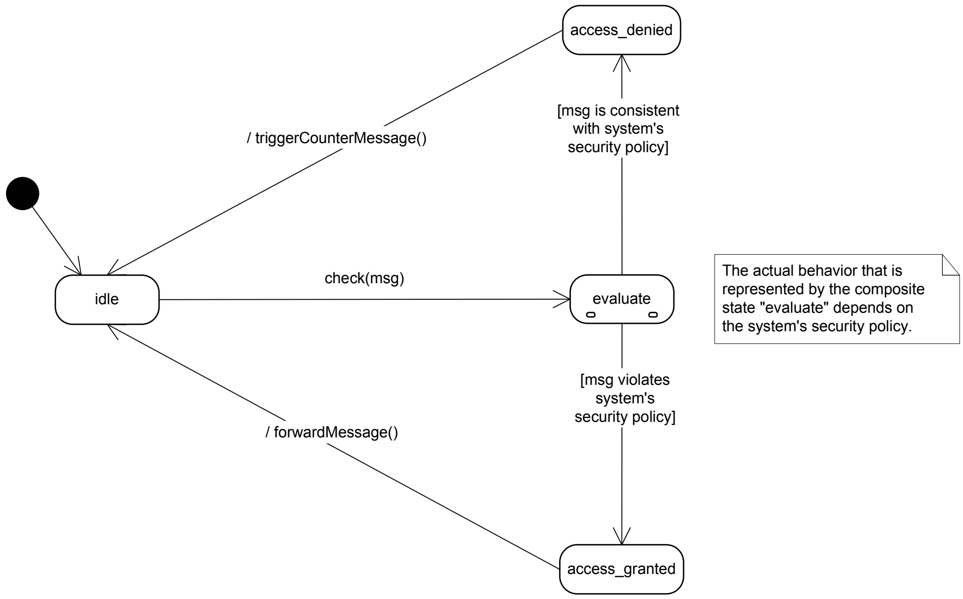

access_denied
/ triggerCounterMessage()
[msg is consistent with system's security policy]
idle
check(msg)
evaluate
The actual behavior that is represented by the composite state "evaluate" depends on the system's security policy.
[msg violates system's security policy]
/ forwardMessage()
access_granted

Figure 14: UML state-diagram: Check Point

- **Constraints**

Authenticity	Check Point's Requests concerning the current Security Policy are only answered by the authentic Security Policy Object.
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 to the implemented security policy, 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, the appropriate actions are triggered.
General Security	If a checked message is potentially harmful according to the implemented security policy, the message is not forwarded to its intended recipient.
Integrity	Messages sent between Check Point and Security Policy cannot be modified.

- **Consequences**

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 therefore costly
Manageability:	Maintenance of security-relevant code will be easier if it is located at one position. Though, complexity of the check algorithm is - depending on the implemented policy - high.
Usability:	Some communication activity might be prevented even if it is not harmful. A high-quality check algorithm is vital.

- **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 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 [13] can be used to decouple 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 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).

- **Community Resources**

  To be determined.

4.3 Roles

The following *Security Pattern* is part of Yoder's and Barcalow's [34] framework of architectural patterns for application security. Furthermore it corresponds with Fernandez' *Role-Based-Access-Control (RBAC)* [11] pattern. We present the *Roles* pattern using our template[7] that is tailored for the security domain. Behavioral and structural information, UML diagrams and a *Constraints* section were added.

- **Pattern Name and Classification**

  The following pattern is termed *Roles*. In accordance to the classification approach chosen by Gamma et al. [13], it belongs to the class of structural patterns as it proposes an infrastructure that is based on the composition of classes and objects.

- **Intent**

  The *Roles* pattern aims for better maintainability of privileges in a system. Thus, it follows Principle 6 (Keep it simple) of the *Ten Principles* by abstracting rights from specific subjects and thereby facilitating maintenance of complex privilege structures.

[7] The mentioned Security Pattern Template is outlined in Section 3.2.

- **Also Known As**

  In accordance to Yoder and Barcalow [34] the *Roles* pattern is also known as

  - Actors,
  - Groups,
  - Projects,
  - Profiles,
  - Jobs and
  - User Types.

- **Motivation**

  Various multiuser-systems provide functionality or resources that may not be used by every user. In order to control access to those restricted parts the system has to keep track of the privileges an user has. Commonly privileges are not derived from an individual's needs but from the tasks that a subject is supposed to perform in the system. Thus, abstracting privileges from persons and assigning them to profiles termed *Roles* can be useful.

  Given some business application (e.g. Enterprise Resource Planning (ERP) Software), certain tasks and queries relate to a specific job positions. For example, the right of employee A to query all corporate cost centers is derived from his job position as Manager of Finance not from the person itself. If A leaves the company all his rights need to be revoked and transferred to B the new Manager of Finance, who will now perform A's tasks. Unfortunately A had a great amount of privileges and the administrative effort caused by A's cancellation is enormous. In some constellations with individuals having more than one position and quitting just a part of them, it is even harder to maintain their privileges.

  The application of the *Roles* pattern simplifies maintenance in such scenarios a lot. As all privileges needed for a certain task are assigned to its respective role, it is sufficient to match users with roles according to their tasks. This approach is by far more efficient while dealing with complex privilege structures than the plain assignment of rights to users. Once all needed roles are created and rights are assigned to them, they are usually not subject to frequent changes.

  The *Roles* pattern belongs to the field of *Security Patterns*, because it addresses problems that threaten system security. Its main contribution is the simplification of the security-critical administration process of privileges.

- **Applicability**

  The *Roles* pattern is applicable to any system that needs to restrict access of participating subjects (i.e. users or processes) to its resources. Especially in large systems that have many users with similar privilege constellations, the implementation of *Roles* can reduce organizational redundancy.

  In systems where it is hard to identify common privilege groups and nearly every user needs a role of his own the implementation of this pattern generates no additional benefit. Single-user systems or systems that have static privileges structures are not subject to the application of the *Roles* pattern at all.

- **Structure**

  The *Roles* pattern is realized by constructing an User-Role-Privilege relationship. Instead of assigning privileges to users, Role objects are created. These objects contain a set of privileges. Several Role objects can be assigned to each user. According to the *Principle of Least Privilege* (Principle 4 of the *Ten Principles*), an user has no privileges unless they are granted to him by an assigned Role object. The different Role objects may be aggregated in a collection object (in our example termed: Roles). This structure is reflected in the UML class diagram in Figure 15. We used the previous example of an online store to show how the *Roles* pattern works in combination with the other *Security Patterns* presented earlier. Those dependencies are subject to further elucidation in Section *Related Security Patterns* of this pattern.

  Note: The information that connects users and privileges is stored inside the Role object. We made this design decision in order to keep security code together. An implementation in which all privileges are checked for certain roles upon a call, would be slow and scatter security code throughout the system [34].

- **Participants**

  – Privilege

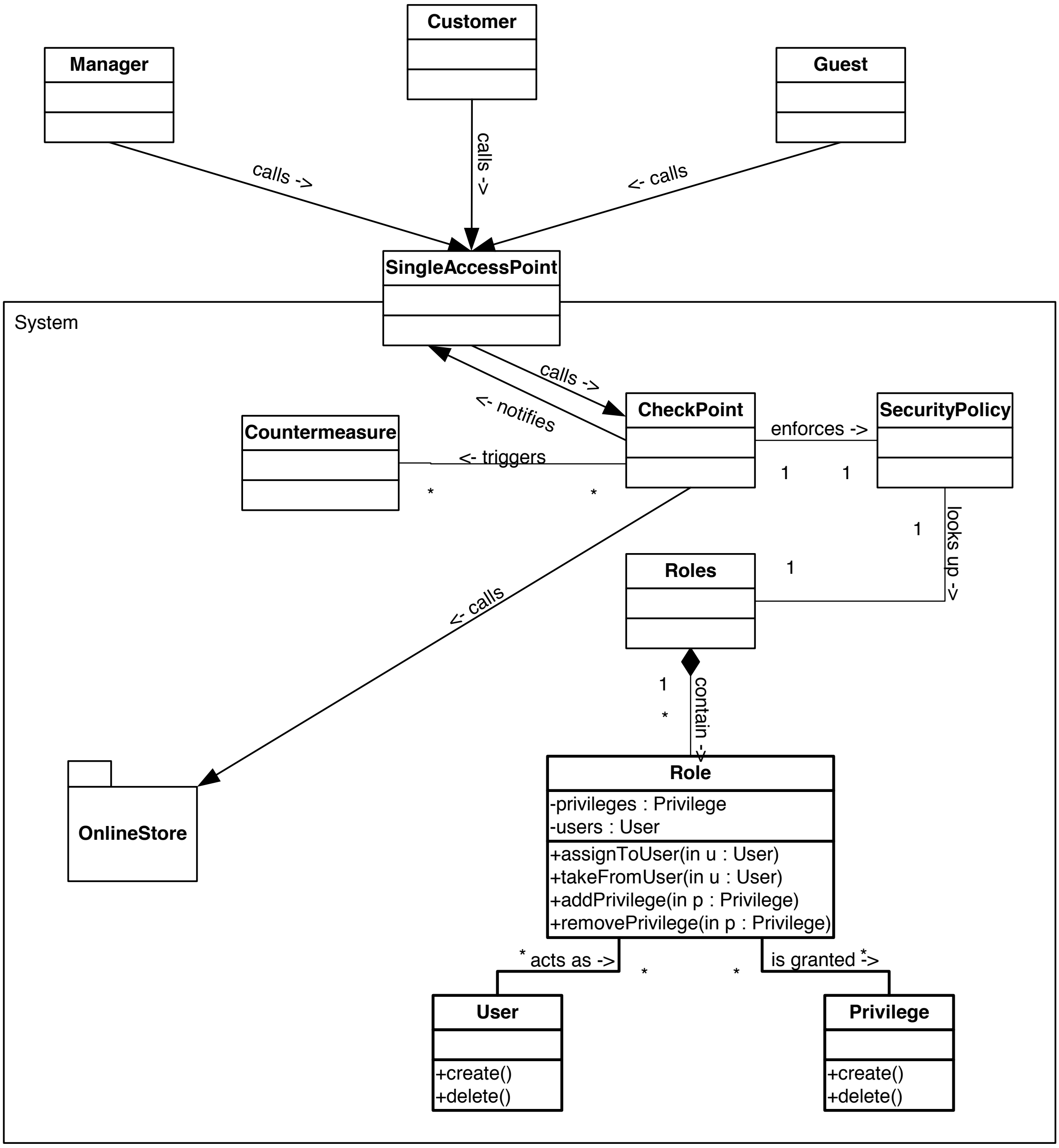


Figure 15: UML class-diagram: Roles

        - defines activities or resource use that is permissable to subjects that where granted this privilege.
        - if violations should be checked automatically, the definition of the privilege needs to be formal.
    - Role
        - holds a set of privileges that are associated to that specific role object.
        - saves the users to which this role is assigned.
        - provides information about its users and privileges.
    - Roles
        - is a collection of all roles in the system.
    - User
        - stores information about an user.
        - exists for each user of the system.

- **Collaborations**

  Role objects contain links to Privilege and User objects. As the User-Role-Privilege relationship determines the rights of certain users it is critical to maintain consistency. Thus, upon deletion of User or Privilege objects, the affected Role objects have to be updated.

- **Behavior**

  Not applicable.

- **Constraints**

General Security	Only authorized subjects may assign roles to users.
General Security	Only authorized subjects may assign privileges to roles.
General Security	The mapping between roles, users, and privileges adheres to the system's access matrix.

- **Consequences**

Accountability:	is not affected
Confidentiality:	The *Roles* pattern can contribute to confidentiality by improving the administration of access rights.
Integrity:	Integrity is affected in a similar way as Confidentiality. The likelihood of unintentionally granting privileges that harm system's integrity is reduced
Availability:	Restriction of resources is used to maintain availability. The *Roles* pattern also simplifies the administration of privileges that deal with resource management.
Performance:	A great amount of Role objects can slow down the process of checking whether a subject has the needed privileges or not. Given an efficient implementation and assuming reasonable use of *Roles*, overall performance can be improved.
Cost:	Maintenance cost can be reduced in complex multiuser systems. Development is more difficult and more expensive on account of the higher level of complexity. Privilege lookups are more complicated and may take longer.
Manageability:	The administrator's possibilities of managing privileges are enhanced, as User-Role and Role-Privilege relationships can be managed separately. This enables an administrator to make use of the additional level of indirection to group common privileges [34]. If numerous privilege combinations exist, a significant amount of roles is necessary. Manageability will be improved if the overall amount of relationships is reduced by the application of *Roles*.
Usability:	In complex systems an user might perceive slightly longer delays in case of extensive privilege checks.

- **Implementation**

  To be determined.

- **Known Uses**

  To be determined.

- **Related Security Patterns**

  The *Roles* pattern can be combined with several other *Security Patterns*. A *Check Point* can use the *Roles* to determine whether operations may be permitted or not. *Session* objects can provide links to the *Roles* that were assigned to the current user. The privilege structure of the *Roles* pattern can be used by the *Limited View* pattern to define what is visible to the user.

- **Related Design Patterns**

  An elegant way of maintaining consistency in the User-Role-Privilege structure is to use the *Observer* [13] pattern. In this way, Role objects will be notified upon deletion of either Privilege or User objects and can take appropriate actions to restore consistency.

  If the system needs to behave differently, depending on the current user's Role, Yoder and Barcalow [34] propose the use of the *Strategy* [13] pattern.

- **Supported Principles**

  Given appropriate preconditions[8], the *Roles* pattern simplifies the administration and helps to keep track of the privileges. Hence, it embodies Principle 6: *Keep it simple.* Furthermore, the organization and enforcement of privileges should follow Principle 4 (*Principle of least privilege*) and grant a subject only those rights it subject is explicitly authorized for.

- **Community Resources**

  To be determined.

4.4 Session

Alike the other patterns presented so far, the *Session Security Pattern* belongs to Yoder's and Barcalow's [34] framework of architectural patterns for application security. Below we transferred it to our security-specific template and enriched it with additional structural and behavioral information. The *Session* pattern presented herein may also be used for non-security purposes. We focus in our presentation only on its utilization for security improvement.

[8] outlined in Section *Applicabiliy* of this pattern

- **Pattern Name and Classification**

  The *Session* pattern deals with object creation (of *Session* objects), accordingly we classify it as creational pattern.

- **Intent**

  The intention of the *Session Security Pattern* is to provide different parts of a system with global information about an user. This may be used to facilitate accountability and to enforce privilege violations.

- **Also Known As** Other names used to address the *Session* pattern are

  - User's Environment,
  - Namespace,
  - Threaded-based Singleton and
  - Localized Globals.

- **Motivation**

  In many systems global information is needed at various positions. Especially security relevant code in multiuser systems needs information on the current users. Usually the *Singleton* pattern would be used to instantiate one global object that holds and provides all required data. Unfortunately, this approach is not applicable in a distributed, multi-user or multi-threaded environment. Several instances might be required, for example one for each user or process.

  To overcome this problem *Session* objects are used. They provide the needed information. In terms of security, *Session* objects are very useful to keep security-relevant information like roles, privileges or authentication data. This information enables components inside the system to conduct further checks based on the provided information (Principle 2: *Practice defense in depth*). A *Session* object can be created after an user has successfully logged into the system. Typically, this would be done at the *Check Point*[9] of a system. In this way, the system can identify a specific user to be authorized for certain operations by checking the respective *Session* object.

[9] see Section 4.2 for more information

- **Applicability**

  The *Session* approach can be used if different parts of a system need access to global information that cannot be provided by a *Singleton.* Furthermore, *Session* objects are not shared by different instances of an application or different users.

- **Structure**

  A *Session* object is created during runtime and shares global data with other system components. In order to facilitate access to a *Session* object from various positions in the system, a mechanism for distributing its instance needs to be implemented. Figure 16 depicts how the *Session* pattern could be integrated in our example system.

- **Participants**

  – *Session*

    ∗ stores information that will be provided by *Session.*

    ∗ the stored information is initialized on creation.

    ∗ provides methods for sharing the information.

  – System components that use *Session*

    ∗ need to know the instance of the *Session* object they use.

    ∗ call methods of session to retrieve information.

- **Collaborations**

  System internal components that require global information need to have a pointer to the instance of their *Session* object. Depending on the design of the system, they might be given the respective *Session* instance with an user's request to perform some action. Once they have access to the instance, it can be used to query the desired information.

- **Behavior**

  Figure 17 depicts the message flow upon an user's access request. After the *Check Point* receives a message from an user it demands the corresponding privilege information from a *Session* object. In the example we outlined here, the access request is permissable according to the privilege structure and the message is forwarded to its intended recipient.

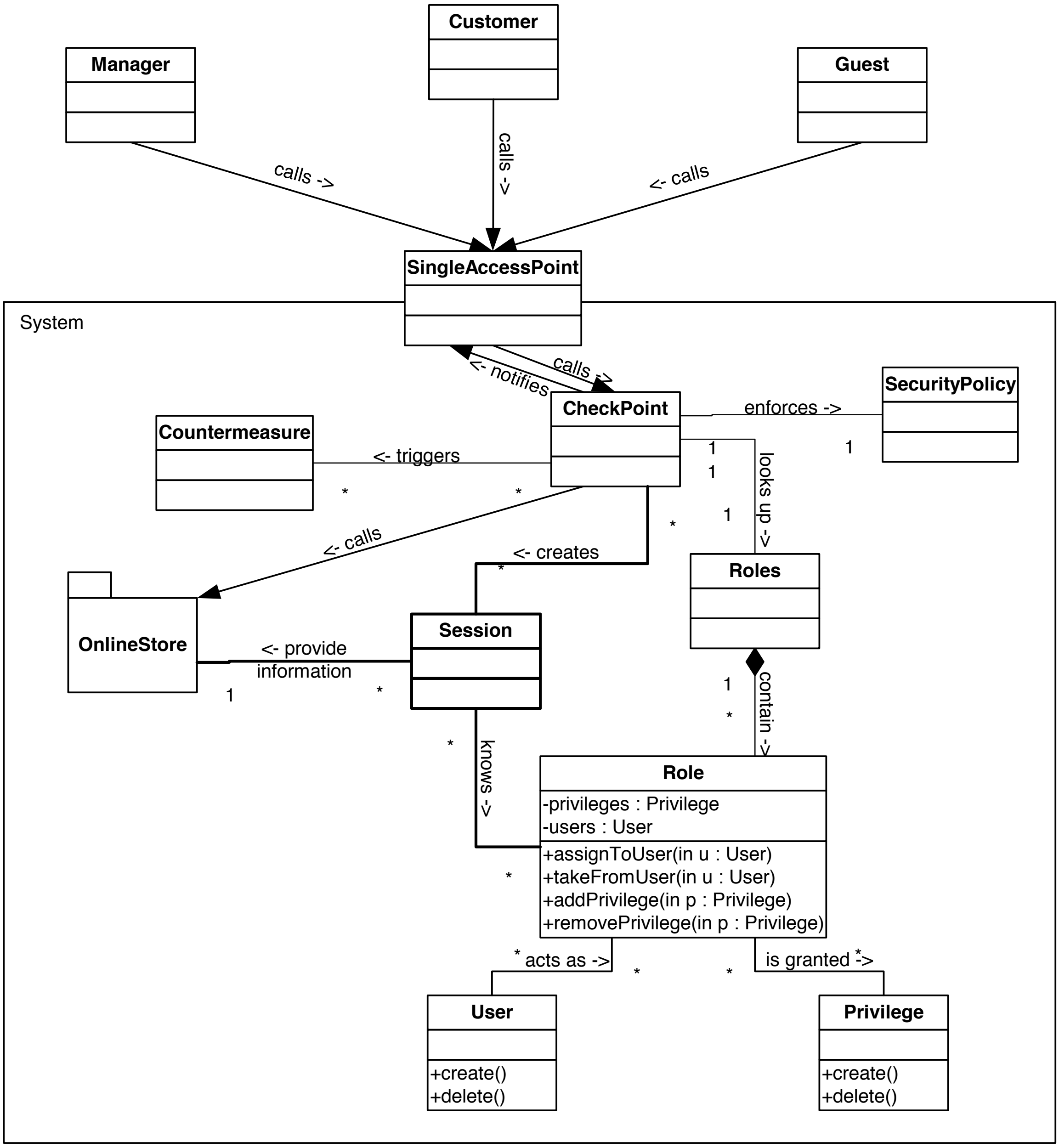


Figure 16: UML class-diagram: Session

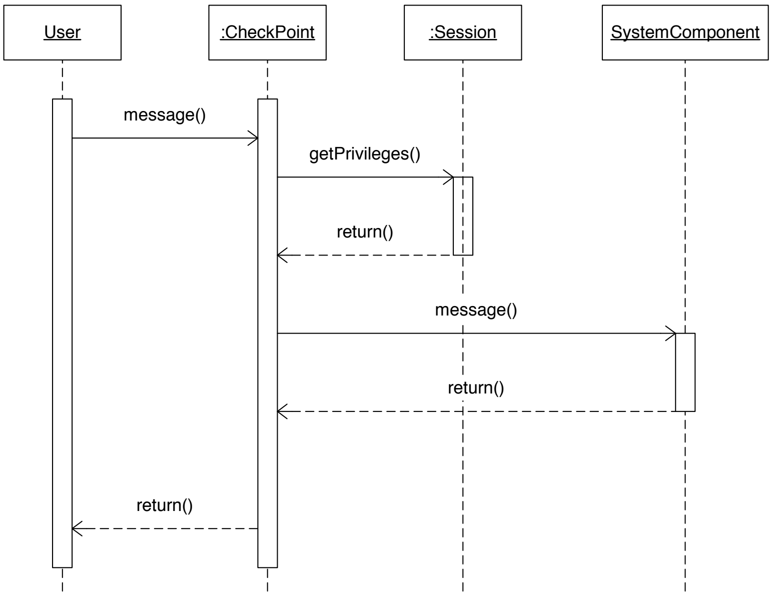


Figure 17: UML sequence-diagram: Session: Example

- **Constraints**

Integrity	Only authorized subjects or functions may change information stored in the *Session*
Confidentiality	Only authorized subjects or functions may access information stored in the *Session*.
General Security	Information is kept secure in the *Session*.

- **Consequences**

Accountability:	Using a *Session* object an user's actions can be traced and logged.
Confidentiality:	By providing role and privilege information in an user's *Session* object, unauthorized access can be recognized and stopped by internal components.
Integrity:	In a similar way as confidentiality, integrity of an user's operations can be checked against his privilege structure in order to identify actions that compromise integrity.
Availability:	Alike Confidentiality and Integrity, Availability can be improved by applying the *Session* pattern. Resource restrictions might be checked according to an user's privileges.
Performance:	Depends on implementation.
Cost:	Development can be difficult. All system components that need global information need to have an instance of *Session*. With increasing amount of information stored in a *Session* object more organization is necessary. Those circumstances may influence development cost.
Manageability:	Manageability, especially during system development is enhanced by the *Session* pattern. The pattern establishes a common mechanism to handle important global variables. Adding a variable to the *Session* object will make it available to all components. Instead of exchanging various variables only the instance of *Session* needs to be transferred. [34]
Usability:	Not affected.

- **Implementation**

  To be determined.

- **Known Uses**

  To be determined.

- **Related Security Patterns**

  The *Session* pattern can be easily integrated with the other *Security Patterns.* As mentioned above, the *Check Point* provides a convenient place to create an instance of *Session*, once an user is authenticated successfully. Furthermore, the *Session* can keep the current users *Roles* and make them accessible to system internal components. Finally, the *Limited View* pattern can query the *Session* object in order to determine which data the current user sees.

- **Related Design Patterns**

  From its tasks the *Singleton* [13] is closely related to the *Session* pattern, which takes into account multi-threaded, multi-user or distributed environments [34].

- **Supported Principles**

  Principle 6 (*Keep it simple*) is supported by this pattern: Access to global variables and the passing of global data is simplified and thus less error-prone. It is crucial to keep in mind that *privacy* should be *promoted* (Principle 7): The *Session* object should not provide security-critical information (like passwords) to system components that may leak or compromise this information.

- **Community Resources**

  To be determined.

4.5 Full View With Errors

Yoder and Barcalow [34] introduced the *Full View with Errors Security Pattern* in their work. The following section adapts the pattern to our security-tailored template.

- **Pattern Name and Classification**

  The *Full View with Errors* pattern is a behavioral pattern. It describes how various parts of the system that deal with user interaction could behave.

- **Intent**

  The *Full View with Errors* pattern depicts an approach to prevented users from performing illegal operations by *failing* them *securely* (Principle 3).

- **Also Known As**

  The pattern *Full View with Errors* is also known as

  - Full View with Exceptions,
  - Reveal All and Handle Exceptions and
  - Notified View.

- **Motivation**

  Systems with user interaction need to concern about how they prevent users from performing restricted operations. Furthermore, the system needs to notify a subject if it performs invalid or forbidden actions.

  The *Full View with Errors* pattern proposes to show an user all options, no matter whether his current *Roles* or privileges allow him to conduct all of them. If an illegal operation is performed the system will notify the user and explain why the chosen action is not permitted.

- **Applicability**

  The *Full View with Errors* pattern is applicable to systems that have user interaction. Furthermore, it is not reasonable to use *Full View with Errors* if there are many operations an user is not able to perform. In such cases where many meaningless options exist the user is distracted from the main functionality (intended for that specific user).

- **Structure**

  Not applicable.

- **Participants**

  Not applicable.

- **Collaborations**

  Not applicable.

- **Behavior**

  The *Full View with Errors* pattern displays all available options. Upon execution of an operation a check is performed that determines whether the action is legal or not. In case of an illegal call the operation is stopped and an error message is displayed.

- **Constraints**

  To be determined.

- **Consequences**

Accountability:	Not affected.
Confidentiality:	Not affected.
Integrity:	Not affected.
Availability:	Not Affected.
Performance:	On the one hand the approach of the *Full View with Errors* pattern does not require checks before a view is updated. Hence, it is usually faster as the *Limited View* pattern. On the other hand the user may choose illegal operations that are subject to costly checking.
Cost:	Cost is not significantly affected by the application of the *Full View with Errors* pattern.
Manageability:	Implementation of this pattern is fairly easy as one has not to concern about different views.
Usability:	By displaying all options - no matter whether they are available at the moment or not - the user does not get confused by missing items. Unfortunately, the user might be tempted to choose an illegal operation and will evoke an error message. Frequent security notifications can annoy the user.

- **Implementation**

  To be determined.

- **Known Uses**

  To be determined.

- **Related Security Patterns**

  The *Full View with Errors* pattern competes with the *Limited View* pattern. They have an opposing strategy. The error-check upon a call of an operation usually makes use of the structure provided by the *Roles* pattern. This structure can be accessed via a *Session* object.

- **Related Design Patterns**

  Not applicable.

- **Supported Principles**

  The main principle underlying the *Full View with Errors* pattern is *Fail Securely* (Principle 3). Furthermore, development can be simplified by applying this pattern. Thus, Principle 6: *Keep it simple* is also subject to this pattern.

- **Community Resources**

  To be determined.

4.6 Limited View

The *Limited View* pattern belongs to Yoder's and Barcalow's [34] framework. In the following section it is presented in terms of our *Security Pattern* template.

- **Pattern Name and Classification**

  The *Limited View* is a behavioral pattern, as it proposes a certain behavior for visualizing components of a system.

- **Intent**

  The *Limited View* intends to prevented users from performing illegal operations by offering only valid operations to them. According to Principle 2 (*Practice defense in depth*) a *Limited View* can be an additional security measure.

- **Also Known As**

  Other known names for the *Limited View* pattern are

  - Blinders,
  - Child Proving,
  - Invisible Road Blocks,
  - Hiding the Cookie Jars and
  - Early Authorization

- **Motivation**

  Users should be deterred from performing restricted operations. The *Limited View* pattern prevents an user to choose illegal actions by not visualizing those options.

  In order to archive this goal, the view has to be customized according to the current users access privileges. This implies that an user's privileges are checked in advance. Thus, the *Limited View* pattern may access *Roles* via a *Session* object.

- **Applicability**

  Alike *Full View with Errors*, the *Limited View* pattern is applicable to systems that interact with users. A visualization based on the patterns concept of hiding irrelevant information is reasonable if there are many options an user is not allowed to access anyway. In some cases the *Limited View* pattern may be applied in order to concisely display the available functionality.

- **Structure**

  Not applicable.

- **Participants**

  Not applicable.

- **Collaborations**

  Not applicable.

- **Behavior**

  The *Limited View* pattern only displays options that the current user is entitled to access. Thus, a check is performed in advance to the visualization of the different options. Hence, any operation that the user may chooses is consistent with his privileges.

- **Constraints**

  To be determined.

- **Consequences**

Accountability:	Not affected.
Confidentiality:	Not affected.
Integrity:	Not affected.
Availability:	Not Affected.
Performance:	In advance to the visualization each option is checked for consistency with the current user's rights. Those tasks can derogate system's performance. Once, all options are displayed some systems do not conduct further checks to improve performance.
Cost:	Cost is not significantly affected by the application of the *Limited View* pattern.
Manageability:	The implementation of this pattern may be more challenging, as the complete right structure needs to be checked simultaneous. However, a combined security check could be more efficient and simplified.
Usability:	The changing options in the interface can confuse the user if the meaning of appearing and disappearing options is not evident to him. It is less likely that an user sees error screens and gets frustrated by them.

- **Implementation**

  To be determined.

- **Known Uses**

  To be determined.

- **Related Security Patterns**

  The *Limited View* pattern embodies the converse approach to the *Full View with Errors* pattern. The structure needed to preselect the options that qualify for a specific user, might be provided by a*Session* object. To map privileges and users the *Roles* pattern could be used.

- **Related Design Patterns**

  Yoder and Barcalow [34] propose to make use of the *State*, *Strategy*, *Composite* and *Builder*

Design Patterns during the implementation of the *Limited View* pattern.

- **Supported Principles**

  The *Limited View* pattern embodies several of Viega's and McGraw's [31] *Ten Principles.* Principle 2 (*Practice defense in depth*) can be accomplished if the preselection of what the user can do is used additionally to other security checks. According to Principle 4 (*Principle of least privilege*) and Principle 7 (*Promote privacy*) this pattern restricts the user to acquire more information than necessary. This is also consistent with the idea of Principle 8, which states that *hiding a secret is hard* and a secure system has to consider insider attacks. Hence, the *Limited View* pattern realizes Principle 9 (*Reluctance to trust*).

- **Community Resources**

  To be determined.

4.7 Authorization

The *Authorization* pattern was presented in Fernandez' work *A pattern language for security models* [11]. It is a *Security Pattern* of high abstraction that has been used in various systems. We present it here in order to show the use of our security-specific template.

- **Pattern Name and Classification**

  The *Authorization* pattern is a structural *Security Pattern.*

- **Intent**

  Authorization provides a structure that facilitates access control of resources.

- **Also Known As**

  Not applicable.

- **Motivation**

  Many systems need to restrict access to its resources according to certain criteria (e.g. a security policy). To structure the different possibilities of access, we distinguish between active entities and passive resources [11]. Below, active entities are also referred to as subjects, who access passive resources (protection objects) of a system. An uniform way that abstracts from the type of subjects and protection objects is highly desirable.

The *Authorization* pattern takes account of these goals and provides an abstract structure that is suitable for representing access conditions in a computational environment. Furthermore, the concept presented in the *Authorization* pattern offers features to facilitate and model the transfer of rights as well as a restriction of the conditions in which authorization rules apply.

- **Applicability**

  The *Authorization* pattern is applicable in any system that requires supervision of subjects, who access resources.

- **Structure**

  The authorization structure of a system can be captured in classes and relationships. Active entities are represented by instances of a *Subject* class and and passive resources by instances of a *Protection Object* class. Between those main participants exists a relation that portrays which subject is entitled to access certain objects. The properties of this relationship are organized in the association class *Rights*. The objects of this class define the type of access, transfer conditions and constraints that restrict the use of *Rights*.

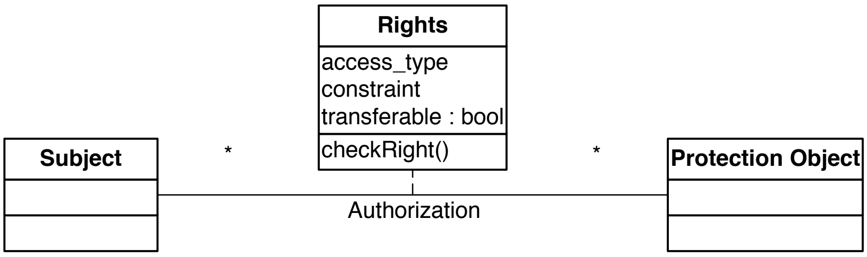


Figure 18: UML class-diagram: Authorization (In accordance to Fernandez [11])

- **Participants**
  - Protection Object
    - represents passive resources of the system that are accessed by subjects.
  - Rights
    - defines the properties of the authorization rule between subject and protection object.
    - The *access_type* property describes which kind of access of the current *Rights* object grants. Commonly this property holds values in the style of read, write, execute, ...

- ∗ The *constraint* property is a predicate that describes under which circumstances the current *Rights* object is valid and may grant a certain privilege.
    - ∗ The *transferable* property determines whether a right is transitive and its connected subject may grant the right to other active entities.
  - – Subject
    - ∗ represents active entities that need to access protected objects in accordance to their rights.

- **Collaborations** Different *Subjects* in the system want to access *Protection Objects.* In order to make use of a certain resource they need to request access to it from the responsible controlling instance. This instance will check if an association class between the subject and the object exists that justifies the required access request. Depending on this examination access is granted or not.

- **Behavior** The behavior of the *Authorization* pattern is depicted in Figure 19 and Figure 20. The first diagram shows a subject that is successfully authorized using the herein presented structure. The second sequence shows how the request of an active entity is rejected.

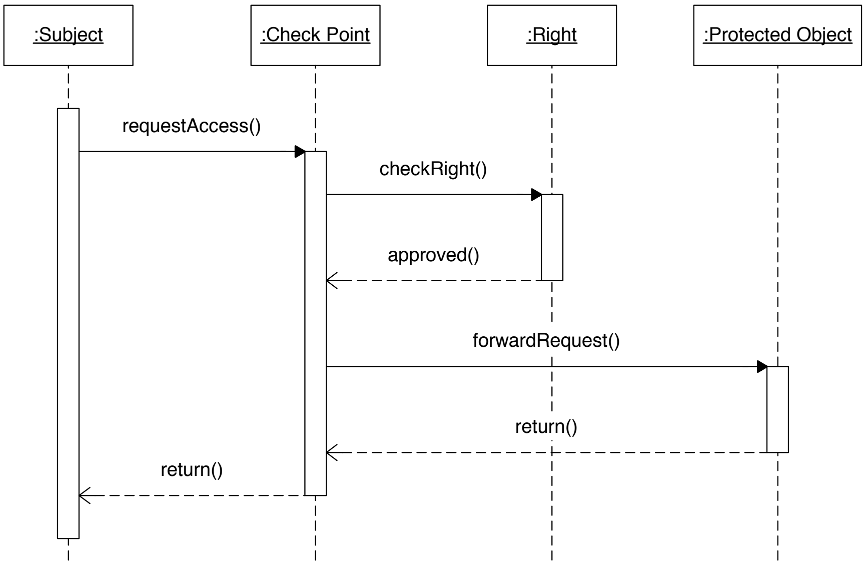


Figure 19: UML sequence-diagram: Authorization approved

- **Constraints**

  To be determined.

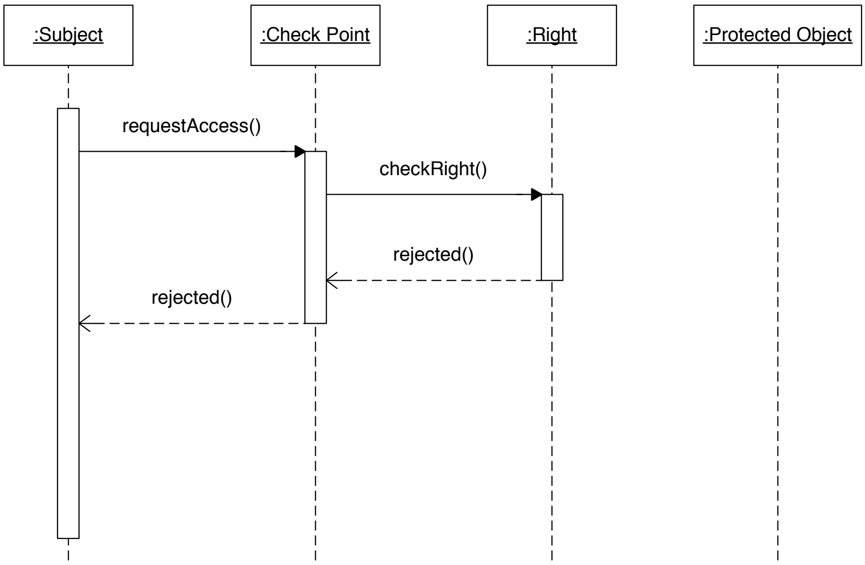


Figure 20: UML sequence-diagram: Authorization rejected

- **Consequences**

Accountability:	Not affected.
Confidentiality:	Can be improved by specification and rigorous enforcement of rights.
Integrity:	Can be improved by specification and rigorous enforcement of rights.
Availability:	Can be improved by specification and rigorous enforcement of rights.
Performance:	The system's performance might be derogated by extensive right checks and the evaluation of the rights' constraint predicates.
Cost:	Cost is not significantly affected by the application of this pattern.
Manageability:	Thanks to the high level of abstraction, the *Authorization* pattern may be applied to manage any type of subject (processes, users, roles, groups, ...) resource (transactions, files, devices, ...) and access (read, write, execute, ...) [11].
Usability:	If the access rights are checked extensively the user might recognize a loss of performance.

- **Implementation**

To be determined.

- **Known Uses**

  To be determined.

- **Related Security Patterns**

  The *Check Point* pattern can be used to examine requests by using the structure of the *Authorization* pattern. The application of the *Session* approach can provide the system with the subject-privilege information. The *RBAC* or *Roles* pattern is an extended version of this pattern.

- **Related Design Patterns**

  Not available.

- **Supported Principles**

  The *Authorization* pattern should be used in combination with Principle 4 (*Principle of least privilege*) to assign the user's rights. Furthermore, *Authorization* can be used to *compartmentalize* (Principle 5) the system and reduce the impact of a security breach. For example, an attacker that illegally manages to authenticate as user A, has only the rights user A would have. Given a restrictive security policy, the enforcement of access rights *promotes privacy* (Principle 7).

- **Community Resources**

  To be determined.

4.8 Multilevel Security

The *Multilevel Security* pattern is part of Fernandez' *Pattern language for security models* [11]. It belongs to the high abstraction level *Security Patterns* and is of special interest for the development of high-security systems (like military applications). The following section adapts this pattern to our *Security Pattern Template.*

- **Pattern Name and Classification**

  The *Multilevel Security* pattern is a structural pattern, as it establishes a structure for systems that partition subjects and objects in different security levels.

- **Intent**

  This pattern tries to provide a mechanism for handling access in a system with various security classification levels.

- **Also Known As**

  Not applicable.

- **Motivation**

  In many systems integrity and confidentiality of data need to be guaranteed. For example in the domain of military intelligence, the classification of data and the clearance of users need to be considered to accomplish these goals.

  The *Multilevel Security Security Pattern* addresses these concerns and provides a structure that allows to have different security levels for subjects and objects. Furthermore, the presented approach facilitates the enforcement of access restrictions in an environment of the mentioned characteristics.

- **Applicability**

  The *Multilevel Security* pattern is applicable to systems that need to provide several security levels for subjects and objects. Furthermore, a hierarchical structure that reflects the subjects and objects sensitivity level has to exist. According to Fernandez [11] this precondition is violated in most commercial environments.

- **Structure**

  To represent the *Multilevel Security* structure for each subject exists an instance of the *Subject Classification* class and for each object an instance of the *Object Classification* class. These instances are used to aggregate a subject's respective an object's security levels and categories. Given the domain of military intelligence, examples of security levels could be: top secret, secret, confidential and unclassified. Categories are predefined compartments that represent object's matter respective the subject's clearance to that compartment.

- **Participants**

  - Object Category

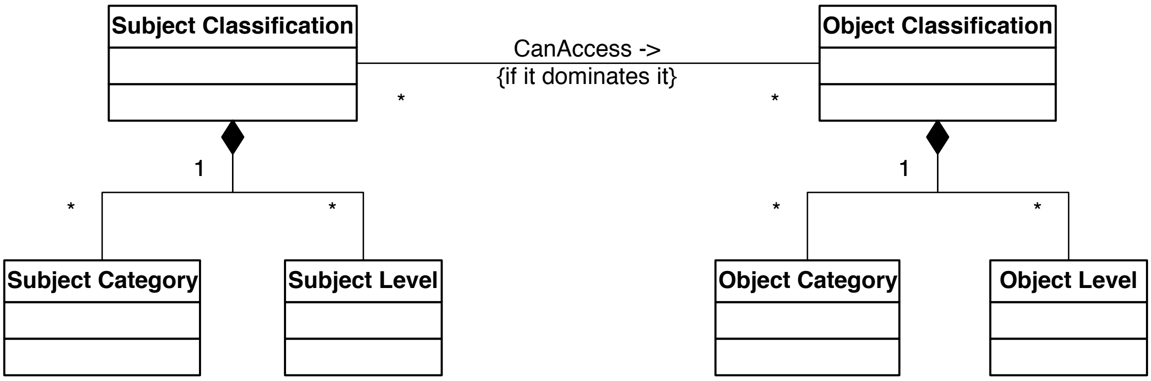


Figure 21: UML class-diagram: Multilevel Security (In accordance to Fernandez [11])

    - defines a category that an object belongs to.

- Object Classification
    - determines the security classification of an object (passive resource that is accessed by subjects).
    - The object's security classification is represented by a set of object categories and object levels.
- Object Level
    - defines the security level of an object.
- Subject Category
    - defines a category a subject has access to.
- Subject Classification
    - determines the security classification of an subject (active entity that accesses objects).
    - The subject's security classification is represented by a set of object categories and object levels.
- Subject Level
    - defines the security level of the subject.

- **Collaborations**

  The classes *Subject Classification* and *Object Classification* contain a set of category and level classes. This set determines the security classification of the respective object. Access will be granted if the classification of the requesting subject dominates the classification of the protected object. The *Behavior* section explains in detail how to determine whether one classification dominates a second one.

- **Behavior**

  The UML sequence diagrams in Figure 22 and Figure 23 depict the message flow upon a subject's access request in a multilevel security system. The first scenario shows a successful authorization. The second scenario portrays the a subject's request being rejected. The decision on whether to grant or to deny access is based on the participating subject's and object's security classification. Below we describe the underlying rules for evaluation of access.

  Access is granted if a subject's classification Cs *dominates* an object's classification Co. To preserve confidentiality the rules of the Bell-LaPadula [4] model are applied.

  Bell's and LaPadula's *simple security* property (also known as *no read up* property) of the model prevents subjects without appropriate clearance to access higher classified data. According to Bell-LaPadula, the different security levels are ordered between a highest and lowest level. An examplary order might be:

  $$\text{unclassified} < \text{confidential} < \text{secret} < \text{top secret}$$

  where *unclassified* states the lowest possible sensitivity level and *top secret* the highest. Given the ordered level structure, Cs dominates Co if the classification level of Cs > the classification level of Co and Co's categories are a subset of Cs's clearance categories. Integrity is addressed by similar rules that are stated in Biba's model, which is outlined in Summer's book on *Secure Computing* [28]. Biba defines rules that prevent subjects of high integrity levels to observe data of lower integrity in order to prevent corrupted data of being taken to higher integrity levels. Furthermore, it is not permissable for subjects to modify objects of higher integrity levels.

- **Constraints**

  To be determined.

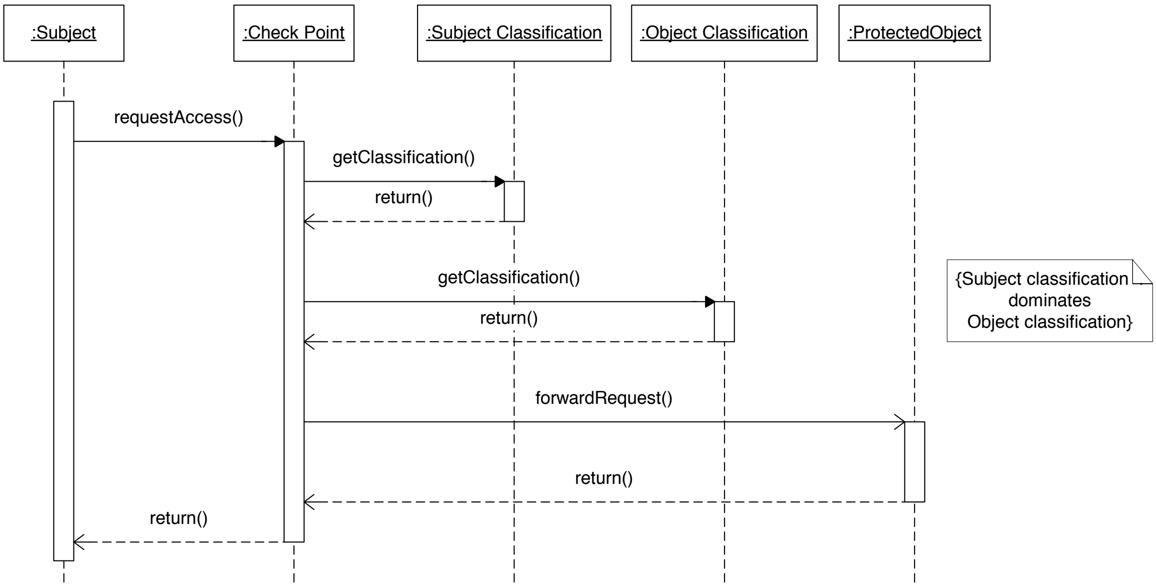


Figure 22: UML sequence-diagram: Multilevel Security: Access granted

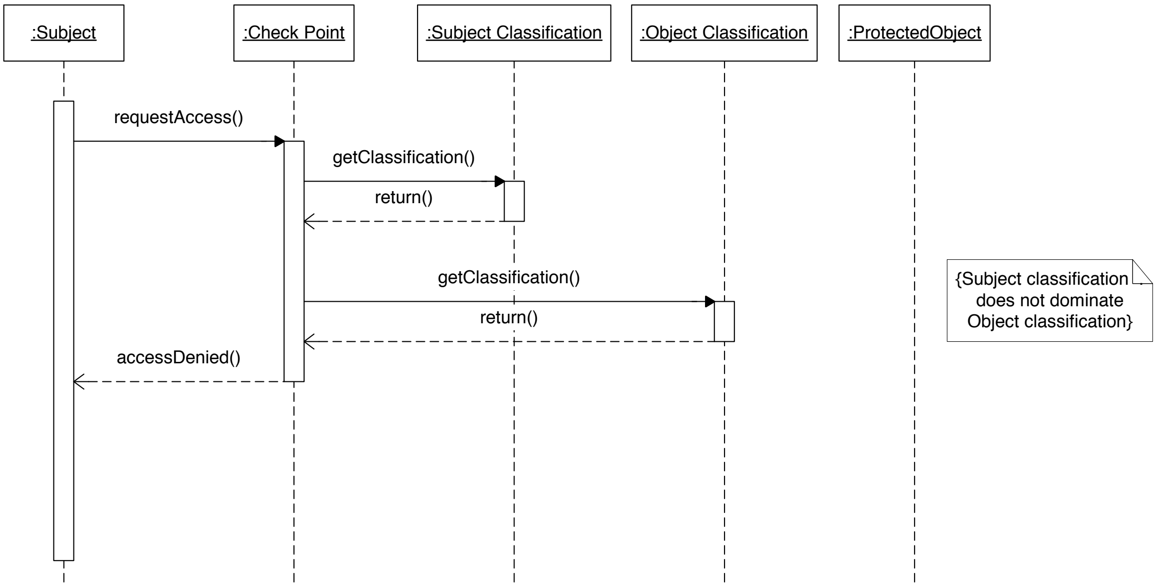


Figure 23: UML sequence-diagram: Multilevel Security: Access denied

- **Consequences**

Accountability:	Not affected.
Confidentiality:	User access is controlled according to the rules that are stated in the Bell-LaPadula [4] model in order to guarantee confidentiality of data.
Integrity:	Biba's model [28] and its rules establish integrity.
Availability:	Not affected.
Performance:	The evaluation of access rights can derogate performance.
Cost:	Establishing a multilevel security system can be costly. All subjects and objects need to be classified in certain sensitivity levels.
Manageability:	The *Multilevel Security* pattern can ease administrative work in an environment that requires classification of subjects and objects.
Usability:	The users range of action might be limited by restrictive rules.

- **Implementation**

  To be determined.

- **Known Uses**

  To be determined.

- **Related Security Patterns**

  The *Check Point* pattern may be used to enforce the *Multilevel Security* structure, which can be provided in *Session* object.

- **Related Design Patterns**

  Not available.

- **Supported Principles**

  The *Multilevel Security* pattern and its mechanism of granting access to subjects implements the *Principle of least privilege* (Principle 4). Furthermore, it facilitates *Compartmentalization* (Principle 5) and reduces the impact of a security breach. *Promoting privacy* (Principle 7), *Hiding secrets is hard* (Principle 8) and *Reluctance to trust* (Principle 9) are main ideas of the *Multilevel Security* pattern.

- **Community Resources**

  To be determined.

5 Formal Verification

This section describes research into investigating the use of formal analysis techniques for elements of security patterns.

In Section 5.1 we outline techniques and tools that are used to verify constraints for security patterns. We describe two systems that each include several security patterns (in Section 5.2 and Section 5.3 respectively). Furthermore, verification results related to each system model are exhibited.

Section 5.2 includes the security patterns *Authorization*, *Check Point*, and *Single Access Point*. Section 5.3 contains constraints related to the patterns *Roles*, *Session*, and *Multilevel Security*. *Full View with Errors* and *Limited View* are not covered by our example systems as their patterns do not specify any structure or behavior that could be checked.

Note, the models that we present in the remainder of this section underwent several revisions before they finally evolved to their current version. This model evolution was influenced by several factors including: revealed errors, previous work into formalizing UML diagrams [10, 23, 18], as well as restrictions imposed by the model checking approach and its tool support.

5.1 Formal analysis techniques and tools

This section provides insight into formal analysis techniques and tools that can be used to verify constraints for security patterns. In prior work [19] this approach was developed and used to formally analyze requirements patterns.

In order to perform model checking we transfer a system, specified in terms of a formalized UML notation, into a formal specification. This specification can be verified against formal requirements using a model checker. During the process of model checking either unknown design errors are revealed or the system adheres to the specified properties. An overview of the model checking process is given in Figure 24. The remainder of this section describes the six basic steps denoted in the figure and presents involved tools.

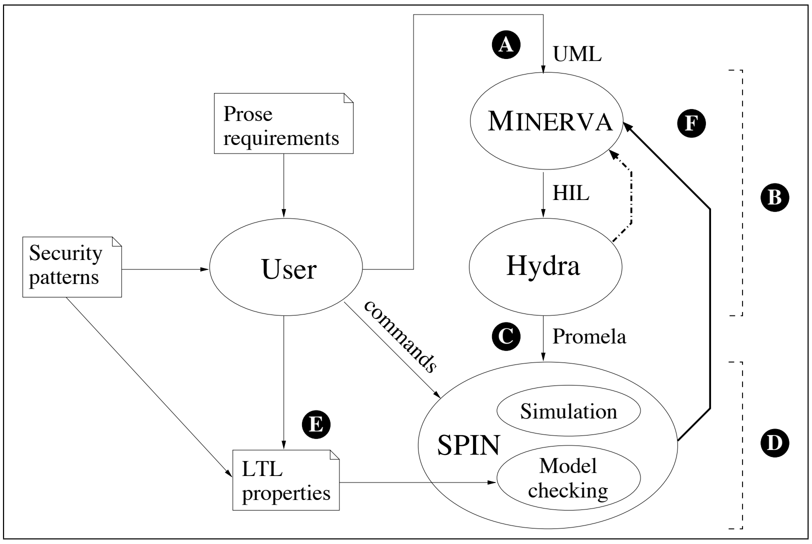


Figure 24: Formal analysis process[19]

It is indispensable to have a formal specification of a system in order to perform automated exhaustive model checking. Thus, the first steps of our formal analysis process address the transformation of our UML models into a formalized syntax. Later steps describe the process of simulating and analyzing the system.

McUmber et al. [22, 23] developed a framework and tools for generating formal specifications from UML diagrams. Their framework is based on homomorphic mappings between metamodels of UML and a target specification language. In order to meet the requirements for the transformation the UML high-level structural and behavioral models of our system are refined in Step A. For all modelling purposes we use the DOmain Modelling Environment (DOME) by Honeywell [15]. DOME is a highly extendable CASE tool that was developed to support model-based development. Thus, it offers extension mechanisms that enable automatic transformation of the models. Minerva [8] makes use of the extensibility and provides support for graphical modelling. In Step B the refined UML models are checked for consistency by Minerva. Afterwards, in Step C the models are transformed to a formal language using McUmber's Hydra [22, 23]. Following a defined set of transformation rules, Hydra automates the generation of a formal specification in a target language. For our purpose we configured Hydra to use the PROcess MEta LAnguage (Promela) as target. That enables us to use Spin [14] a software tool for formal verification, which accepts Promela-specified systems. Spin was developed at Bell Labs and is capable of simulating models and exhaustive checking of properties. When used as exhaustive verifier, Spin successively generates the state space imposed by the Promela system specification. Then all possible states are rigorously checked for compliance with the properties. Unfortunately, the limitations of computation are reached quickly during the verification of complex systems. Especially nondeterminism and concurrency in the model significantly add to the size of the state space. Abstraction, reduction and compression techniques are implemented in Spin to cope with exponential growing state space. Nevertheless, optimization of the model still is a relevant issue. In Step D the newly generated Promela specification is simulated in Spin. If this first validation process reveals errors then they can be corrected using Minerva's visualization support [21]. In Step E the prose requirements are transformed into LTL (linear time temporal logic). LTL is an extension of propositional logic. In order to formulate properties on sequences of states the temporal operators $\Box$ (always), $\Diamond$ (eventually), $\cup$ (until), and $\circ$ (next) are added. Dwyer et al. [10] developed specification patterns that support the process of property generation. Furthermore, a specification pattern repository [1], containing several templates for LTL claims that are frequently used in the software engineering domain, is available on the Internet. After we expressed our properties in LTL, they are entered in Spin. In order to conduct the verification Spin translates the LTL claims into a Büchi automaton [7] that only accepts system executions satisfying the LTL formula [14]. Once a violation of the LTL property is found the verification process aborts and prints out a trace to the error. By means of Minerva the error can be fixed in Step F. Then, the verification is resumed until the model is proven to adhere to the specified LTL property.

After presenting our model checking approach, the following sections demonstrate two example systems.

5.2 Exemplary verification 01

In order to demonstrate the capability of constraint verification, we designed exemplary systems that we check against our constraints from the pattern templates.

This section is organized in three parts. First, we give a detailed overview of the structure and collaborations of our system. Second, we describe the evolution of this system, including changes that were made due to errors or environmental restrictions. Third, we show the results of analyzing the security constraint properties for the patterns *Authorization*, *Single Access Point*, and *Check Point*.

5.2.1 System design

This section elucidates the design of our first example system using a subset of UML, including formal specifications generated from the UML models. This section starts with an overview of our system's structure and its class diagram. Then, we describe our model's state diagrams in alphabetical order. The Promela code, which was automatically derived from those diagrams, is given in Appendix A.1. This Promela specification represents the model checker's input containing the formal system definition.

Class structure

The class structure of our first system is depicted in Figure 25. It contains eight classes, where the *_SYSTEMCLASS_* has only an organizational purpose. This class defines the initialization and instantiation process of the system. The nondeterministic *ExternalRequestGenerator* randomly creates messages that simulate external access requests to the system. Those messages are received by the *SingleAccessPoint*. Before the 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 permissable 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 *InternalEntityStub*, which substitutes all internal entities. Success or refusal of access is finally reported to the external entity that is represented by the *ExternalRequestGenerator*.

Attributes and ranges

Throughout the system, we use a limited number of attributes to save status information and to transfer data between classes. Figure 26 gives an overview of the system's variables and their possible values. For better understanding we arrange the attributes in three groups that reflect their purpose in the system.

The remainder of this section describes the system's classes in alphabetical order and depicts their UML state-diagrams.

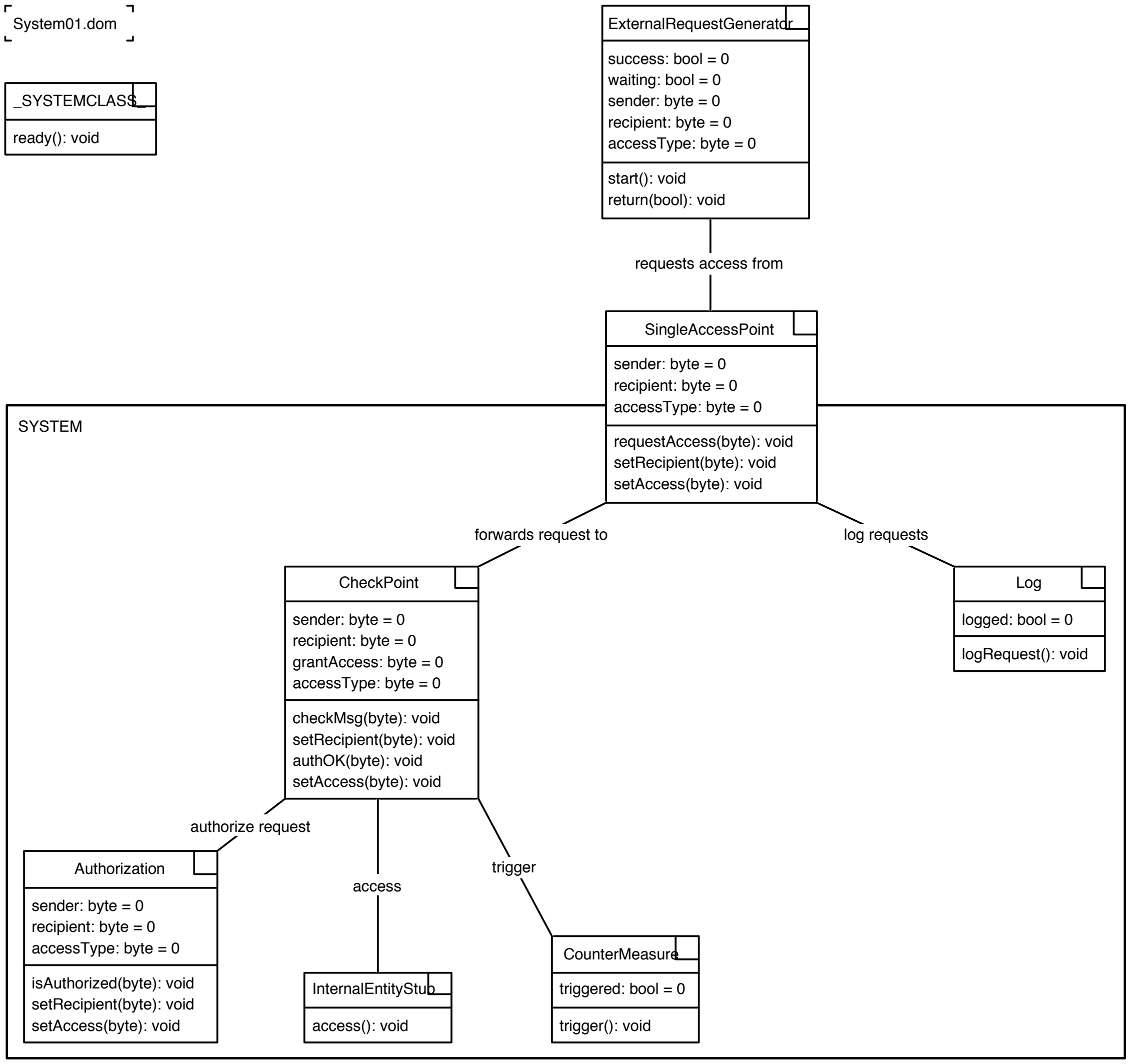
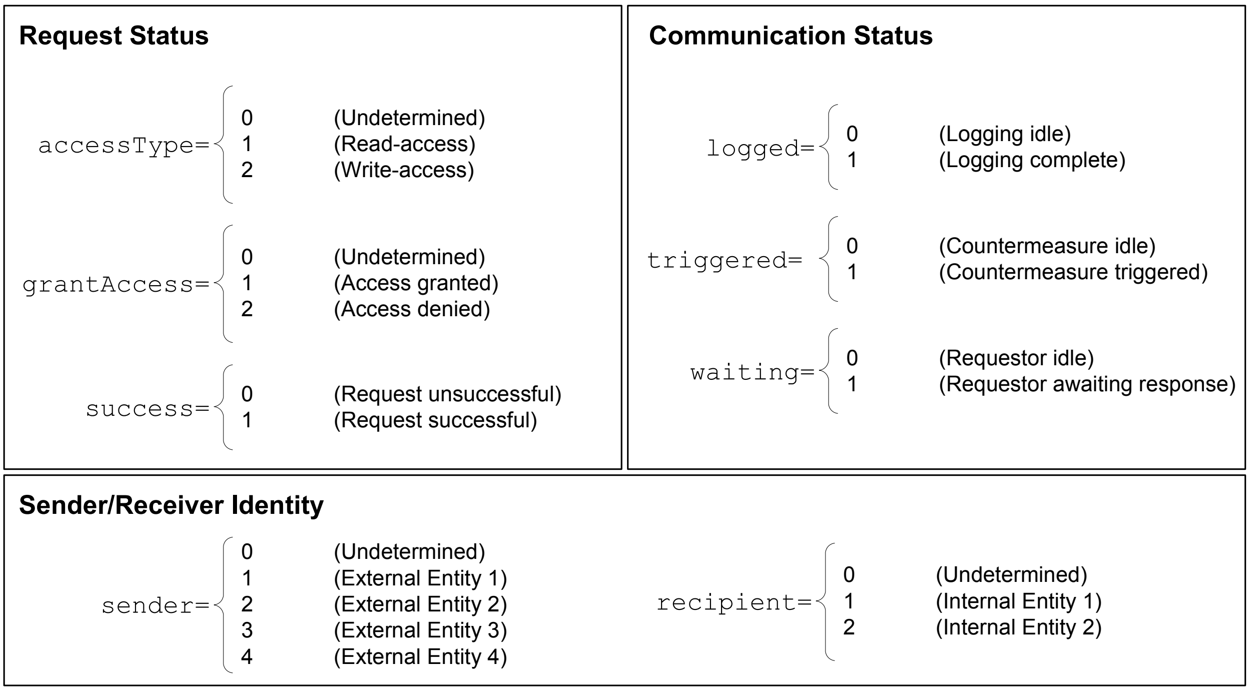


Figure 25: UML class-diagram: System 01

SYSTEMCLASS state machine

The _SYSTEMCLASS_, depicted in Figure 27, is the only class that is instantiated and run automatically. Hence, all other classes are instantiated on the transitions in the _SYSTEMCLASS_ state machine. In order to synchronize the message-flow and to guarantee that the classes are instantiated in the defined sequence, all but the first transitions wait for a *ready()* event that confirms the completion of the preceding step. To delay the system's regular activities until the initialization is completed, we add wait states to all state machines, whose start states have transitions ready to fire. After all initializations are completed in the _SYSTEMCLASS_, *start()* events are triggered to release the wait states. In our model it is sufficient to block only the *ExternalRequestGenerator*



Request Status

accessType= 0 (Undetermined)
1 (Read-access)
2 (Write-access)

grantAccess= 0 (Undetermined)
1 (Access granted)
2 (Access denied)

success= 0 (Request unsuccessful)
1 (Request successful)

Communication Status

logged= 0 (Logging idle)
1 (Logging complete)

triggered= 0 (Countermeasure idle)
1 (Countermeasure triggered)

waiting= 0 (Requestor idle)
1 (Requestor awaiting response)

Sender/Receiver Identity

sender= 0 (Undetermined)
1 (External Entity 1)
2 (External Entity 2)
3 (External Entity 3)
4 (External Entity 4)

recipient= 0 (Undetermined)
1 (Internal Entity 1)
2 (Internal Entity 2)

Figure 26: System 01: Attributes and ranges

class since all other state machines wait for requests.

Authorization state machine

The state machine of the *Authorization* class is shown in Figure 28. Once it is started and the first transition is executed, the state machine reaches the *Idle* state. During this starting-transition a *ready()* message is sent to the *_SYSTEMCLASS_* to notify it of *Authorization's* successful creation. In the *Idle* state the machine waits for an *isAuthorized(sender, recipient, accessType)* request from the *CheckPoint*. Due to the limited capability of the current tools, each message can transfer only a maximum of one parameter. Hence, we had to convey the information on multiple messages. Besides the regular *Idle* state, two additional states (*getParam1* and *getParam2*) are required to accept the three messages *isAuthorized(sender)*, *setRecipient(recipient)*, and *setAccess(accessType)*. Upon receival of the last parameter the request transfer is completed and the state machine enters the *Evaluation* state.

The subsequent states and transitions implement the privilege-structure of the system, which is shown in the Access Matrix in Table 3.

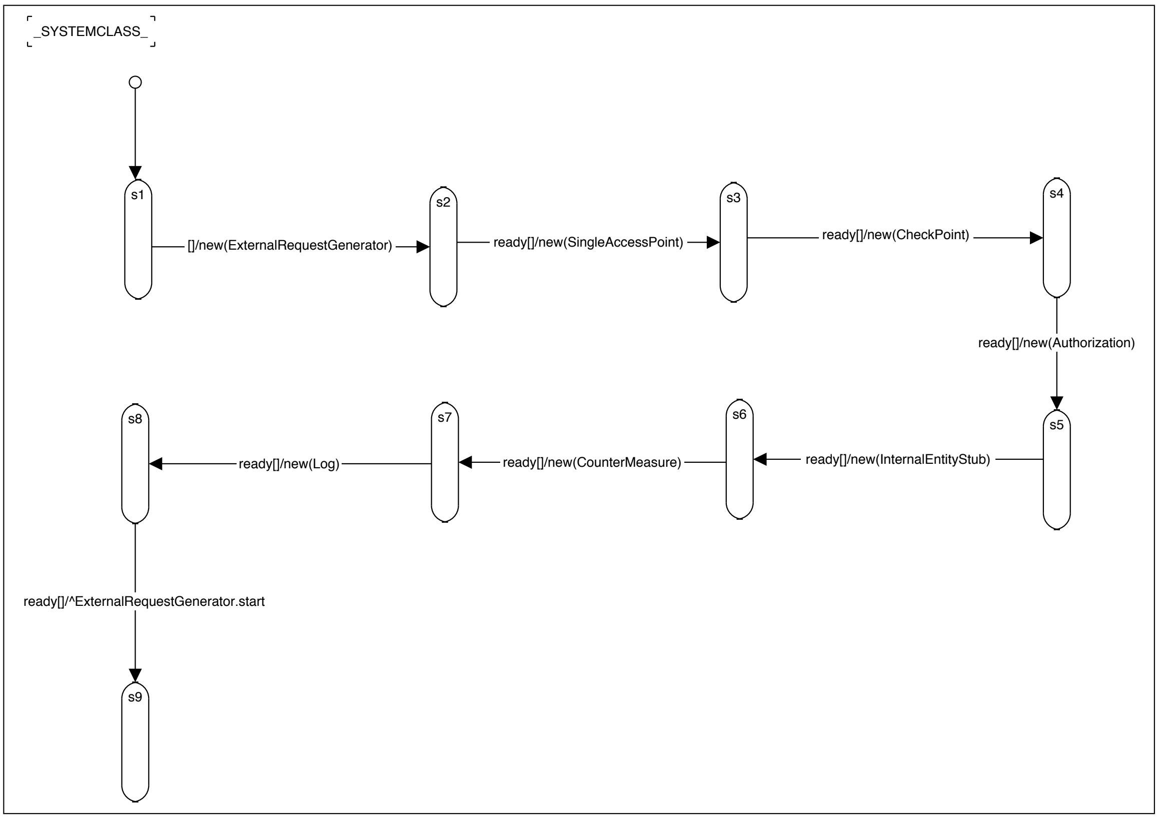

SYSTEMCLASS
s1
[]/new(ExternalRequestGenerator)
s2
ready[]/new(SingleAccessPoint)
s3
ready[]/new(CheckPoint)
s4
ready[]/new(Authorization)
s5
ready[]/new(InternalEntityStub)
s6
ready[]/new(CounterMeasure)
s7
ready[]/new(Log)
s8
ready[]/^ExternalRequestGenerator.start
s9

Figure 27: UML state-diagram: System 01: _SYSTEMCLASS_ (for initialization)

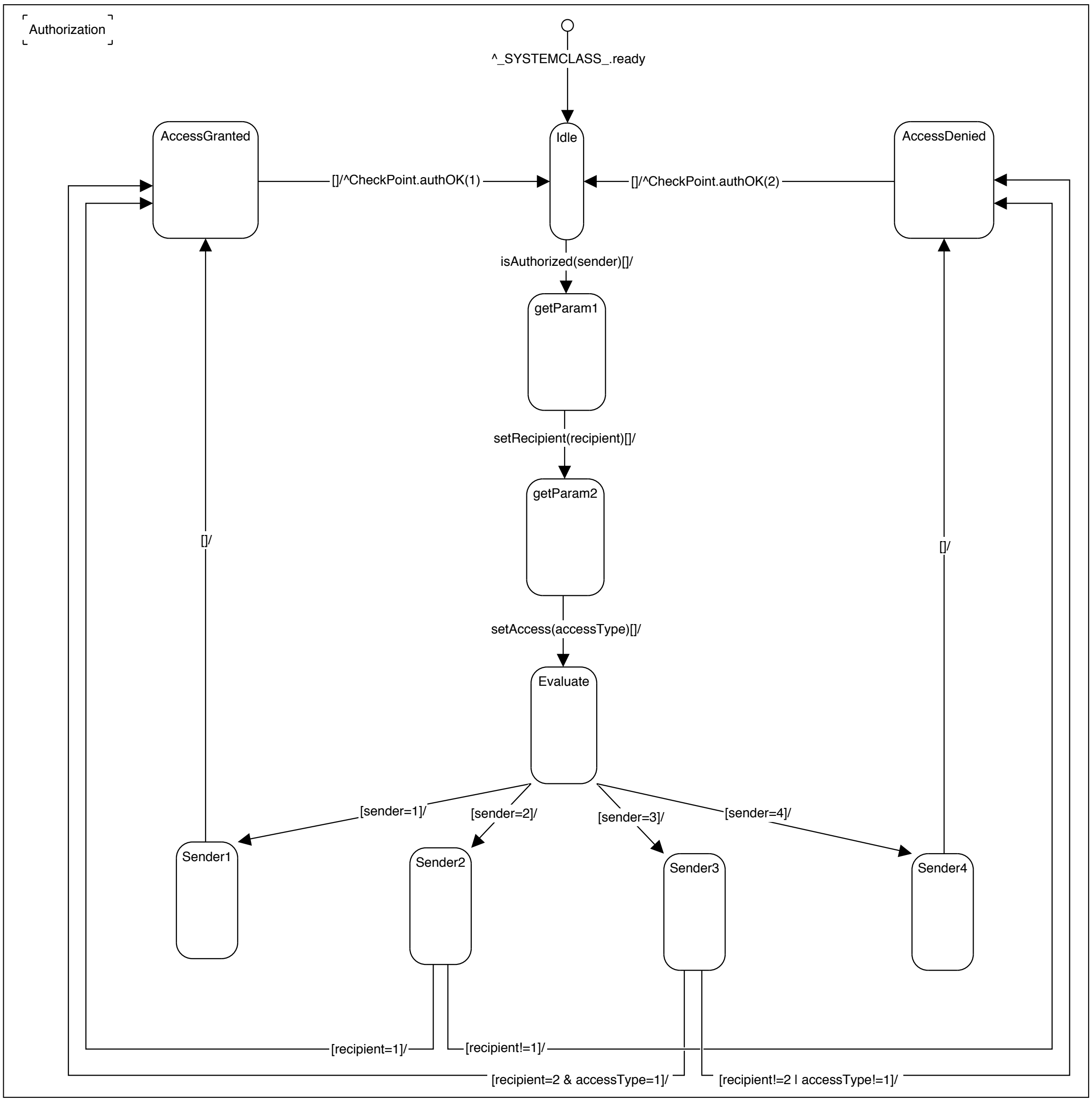


Figure 28: UML state-diagram: System 01: Authorization class

		Internal Entities				Example
		Recipient 1		Recipient 2		
		read	*write*	*read*	*write*	
External Entities	Sender 1	yes	yes	yes	yes	Administrator
	Sender 2	yes	yes	no	no	Owner of Internal Entity 1
	Sender 3	no	no	yes	no	Reader of Internal Entity 2
	Sender 4	no	no	no	no	Unauthorized user

Table 3: System 01: Access matrix implemented in the Authorization class

In the *Evaluation* state, the state machine switches to either one of the four sender-states, depending on the origin of the access request. Those sender-states have transitions to *AccessGranted* and *AccessDenied* states. Guards on the transitions implement the actual access logic and account for compliance of the requests that reach the *AccessGranted* state. Requests that are not consistent according to the current access policy lead to an activation of the *AccessDenied* state. Finally, the state machine returns to its *Idle* state while sending the result of its evaluation to the *CheckPoint* class.

CheckPoint state machine

The *CheckPoint* state machine is depicted in Figure 29. 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. As multi-parameter messages are not supported the we use a similar work-around as in the *Authorization* class. The information is decomposed and transferred using multiple messages (*checkMsg(sender)*, *setRecipient(recipient)*, *setAccess(accessType)*) and states *MainIdle*, *getReq1*, *getReq2*). 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 stays 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. For initialization and state space reduction purposes we reset the request-related variables *sender, recipient, and accessType.* Depending on 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 CounterMeasure whereas the transi-

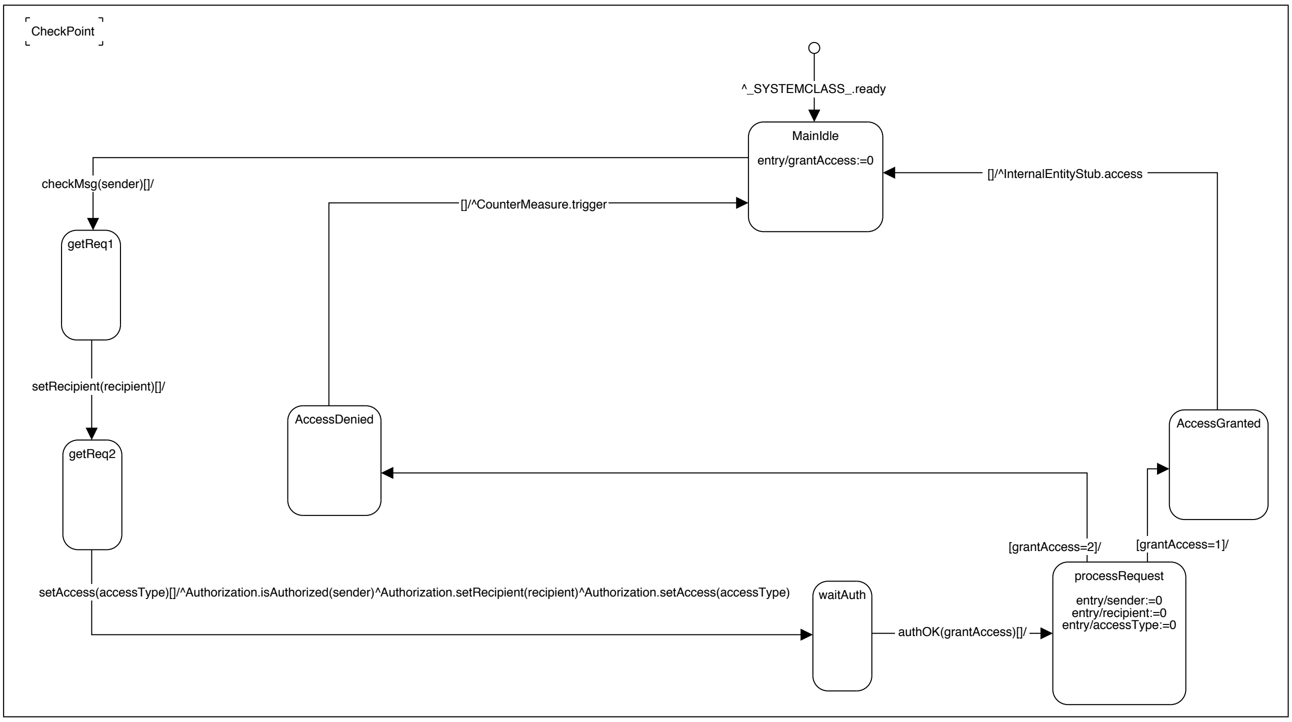


Figure 29: UML state-diagram: System 01: CheckPoint class

tion from the *AccessGranted* state to the *MainIdle* state initiates the access to the InternalEntity (denoted as the *InternalEntityStub* in our system).

CounterMeasure state machine

Directly after the start of the *CounterMeasure* state machine that is shown in Figure 30, it notifies the _SYSTEMCLASS_ of its creation and enters the *Idle* state. A *trigger()* event from the *CheckPoint* class enables a change to the *CMTriggered* state. Simultaneously the internal attribute *triggered* changes its value to *one*. Subsequent to the *CMTriggered* state the *Idle* state is activated again while resetting the *triggered* attribute to *zero*.

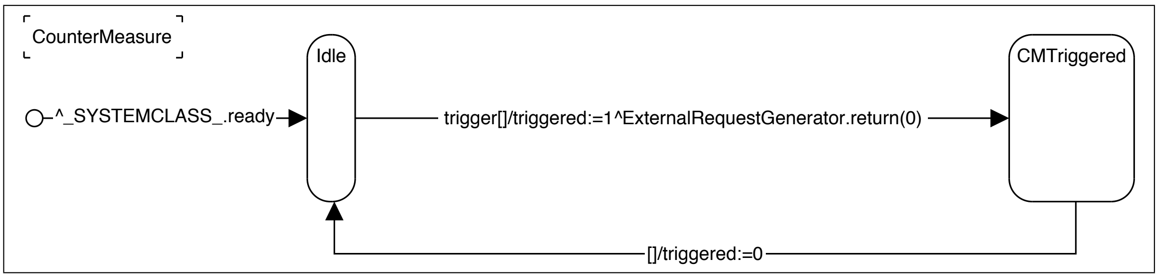


Figure 30: UML state-diagram: System 01: CounterMeasure class

ExternalRequestGenerator **state machine**

The *ExternalRequestGenerator* class that is presented in Figure 31 creates access requests and thereby initiates any activity of the system. In order to guarantee that the system's processing of messages is deferred until all classes are instantiated, the starting transition results in a wait state after notifying the _SYSTEMCLASS_ of its creation. The *ExternalRequestGenerator* machine remains in this state until the _SYSTEMCLASS_ releases it by dispatching a *start()* event. Upon consumption of this *start()* event, the _SYSTEMCLASS_ is informed of the receipt by a *ready()* message and the *genSender* state is entered. The states *genSender*, *genRecipient*, *genAccess*, and *sendReq*, are now activated in succession. They are connected by more than one fireable transition. This intentional nondeterminism is used to randomly create the parameters of a request. The

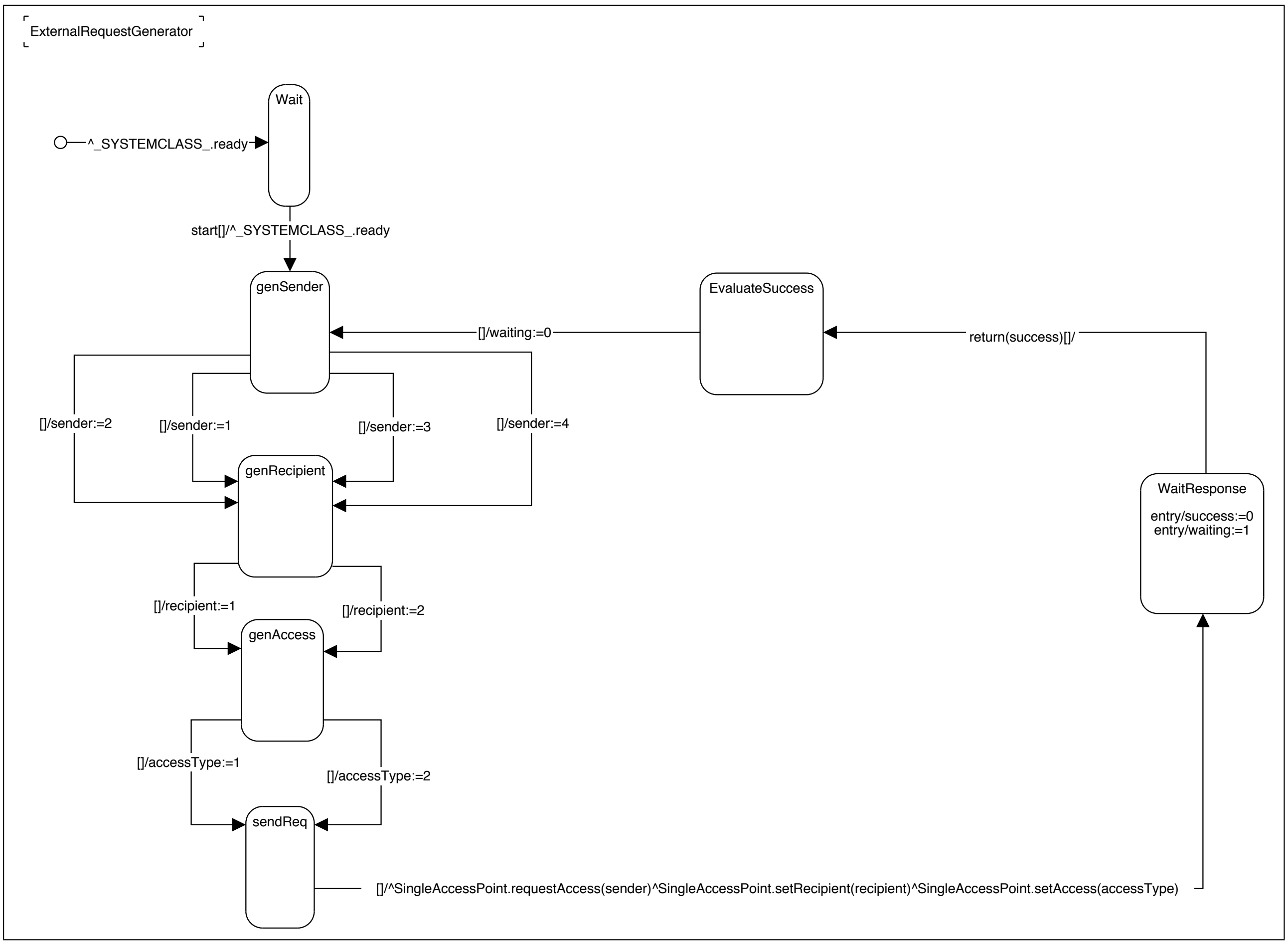


Figure 31: UML state-diagram: System 01: ExternalRequestgenerator class

transitions that connect the states *genSender* and *genRecipient* determine which external entity is simulated during this request. Upon execution of the transition, the local attribute *sender* is set to a value. Similarly, the current request's recipient and access type are determined in the following transitions between *genSender* and *genAccess*, respectively, between *genAccess* and *sendReq*, thereby setting the attributes *recipient* and *accessType*. During the model checking of our system, using exhaustive state space creation, all possible combinations of parameters are tested. Once, the state machine reaches the *sendReq* state, all parameters are determined and stored in local attributes. The next transition, reaching the *WaitResponse* state, sends the request including its parameters to the *SingleAccessPoint*. Due to the limitations of our tools, the request is split into three separate messages, containing one parameter each. The *ExternalRequestGenerator* will remain in the *WaitResponse* state until the system responds to the request. The *success* attribute that shows the outcome of the request is reset to *zero* and the *waiting* flag is set to *one*. Those variables are used during the property verification. The reception of a *return(success)* event causes the state machine to enter the *EvaluateSuccess* state and set the local *success* attribute to the value submitted by the system. Subsequently, the *ExternalRequestGenerator* resets the *waiting* flag to *zero* and returns to the *genSender* state in order to create a new access request.

***InternalEntityStub* state machine**

The *InternalEntityStub* class that is shown in Figure 32 substitutes the internal entities of our system. Upon creation it notifies the _SYSTEMCLASS_ by sending a *ready()* event and enters the *Idle* state. The receipt of an *access()* event, dispatched by the *CheckPoint* class, initiates a transition. This transition sends a *return(1)* message to the *ExternalRequestGenerator*. The return value *one* informs the external entity that its current request was successfully processed.

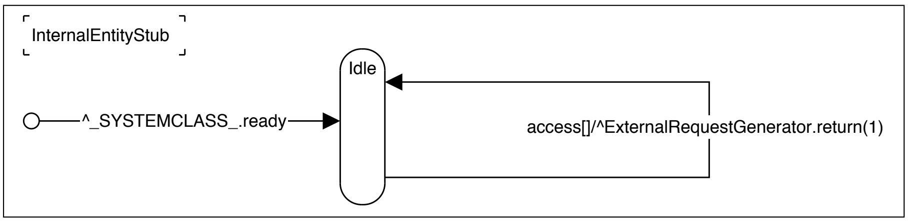


Figure 32: UML state-diagram: System 01: InternalEntityStub class

Log state machine

Similar to the other state machines (excluding _SYSTEMCLASS_), the *Log* (Figure 33) dispatches a notification of its creation to the _SYSTEMCLASS_. Then the *Idle* state is activated and the internal attribute *logged* is reset to *zero*. Upon receipt of a *logRequest()* event from the *SingleAccessPoint* class, the state machine enters the *Logging* state, while changing the *logged* attribute to *one*. When we reach this state, we assume that our system has successfully performed the logging task. Thereafter, the Log class returns to the *Idle* state and is ready to accept new logging requests. The *logged* attribute in this class is needed to determine whether the *Logging* state is reached during model checking.

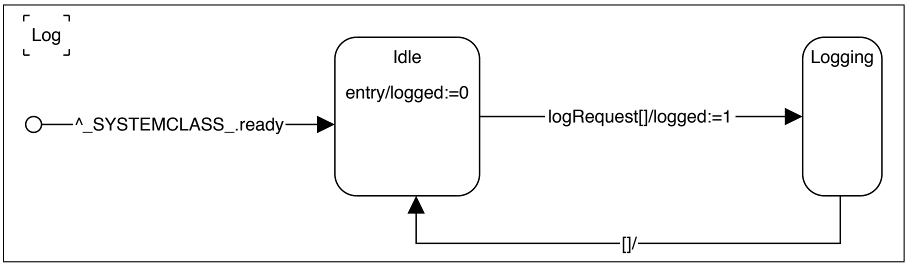


Figure 33: UML state-diagram: System 01: Log class

SingleAccessPoint state machine

The *SingleAccessPoint* state machine is depicted in Figure 34. After the execution of its starting-transition and notifying the _SYSTEMCLASS_, the *Idle* state is entered, thereby resetting the attributes *sender*, *recipient*, and *accessType* to their initial value: *zero*. Upon, receipt of an external request that we created with the *ExternalRequestGenerator* class, the request-parameters are transferred and stored in the corresponding local attribute. Due to the restrictions of our tools, the transfer of the request-information involves several messages (*requestAccess(sender)*, *setRecipient(recipient)*, and *setAccess(accessType)*) and states (*Idle*, *GetParam1*, and *GetParam2*). Once all information has arrived, a *logRequest()* event is dispatched to the *Log* class and the state *Forward* is entered. From there the state machine returns to its *Idle* state, while forwarding the request to the *CheckPoint* class using three separate messages.

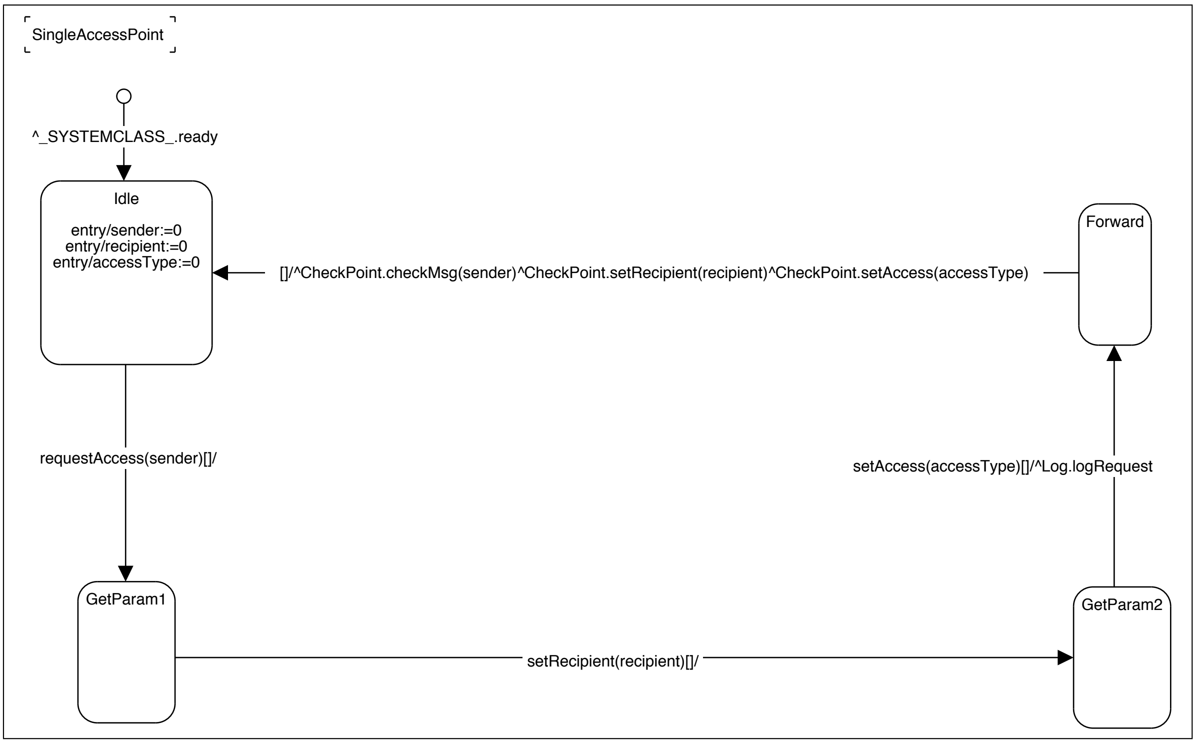

SingleAccessPoint
^_SYSTEMCLASS_.ready
Idle
entry/sender:=0
entry/recipient:=0
entry/accessType:=0
[]/^CheckPoint.checkMsg(sender)^CheckPoint.setRecipient(recipient)^CheckPoint.setAccess(accessType)
Forward
requestAccess(sender)[]/
setAccess(accessType)[]/^Log.logRequest
GetParam1
GetParam2
setRecipient(recipient)[]/

Figure 34: UML state-diagram: System 01: SingleAccessPoint class

5.2.2 Model Refinement

The final system that is shown in the previous section is the result of several revisions of the original models. This section attempts to give insight into the learning process inherent to the design of our example system. First, we depict modifications that were necessary in order to take into account the restrictions that were given by the system's complexity with regard to the model checking approach and the capability of the analysis tools. Second, we elucidate shortcomings and errors we came across while simulating and verifying our system's behavior.

Abstraction Techniques

In order to verify properties with Spin, the model checker generates the state space of the system and exhaustively searches for violations of the properties. A number of design factors can cause the state space to grow rapidly and easily reach the limits of computation, including the number of variables, nondeterminism, and concurrency.

When we generated our first system with several classes representing external and internal entities we recognized that the state space spanned by our model was too large to check. Various external and internal components in this design enabled several requests to be processed concurrently. Hence, significant effort was necessary to synchronize the messages. In order to cope with this problem, we used a number of abstraction techniques to reduce the state space. For example, similar tasks of all internal entities have been abstracted to be performed by only one class. External entities are represented in a similar fashion by one *ExternalRequestGenerator* class. An internal status variable *sender* that captures the origin of a message allows simulation requests from different external entities. Analogously, we distinguish among virtual internal entities by using a *receiver* attribute. This design allows us to simulate all possible combinations with multiple senders and receivers while eliminating concurrency. In this case it was possible to introduce abstractions that do not lose any individual behavior of the abstracted components. Regarding the given privilege-structure the issue of concurrency does not change whether a certain access is granted or not. The modification just minimizes the effort that is necessary to synchronize messages and thereby reduces the system's state space.

A second technique we used to reduce the computed state space is to limit the use of variables to a minimum and restrict their definition to include the smallest range needed. Thus, the number of possible states that is calculated by SPIN is reduced.

While creating the state space of a system, SPIN scans for occurrences of known state-variable-message-combinations in order to take loops into account. Once a known state-combination is found, SPIN stops the computation of the current transition-path. Hence, the earlier a known scenario occurs the smaller the created state space is. In order to make use of this relation, we try to reach known state-combinations as soon as possible. We do that by resetting variables to defaults as soon as the current values are no longer needed. We applied this technique to the *processRequest* state of the *CheckPoint* class, the *MainIdle* state of the *CheckPoint* class, and the *Idle* state in the *SingleAccessPoint* class. We explicitly did not reset the variables in the *ExternalRequestGenerator* class as we need their values during the property verification.

The current model checking approach is not intended to prove complex properties for large systems. Instead, its strength is showing the validity of an abstracted model or critical system parts.

Errors Detected

Besides modifications aimed for the optimization of the system, we needed to make several changes that were induced by errors, which we uncovered during the verification. Those errors can be organized in two classes: First, we identify problems concerning the simulation environment. Second, we focus on problems that relate to our actual system design. In terms of design improvement, the second class is very interesting as it points out errors that were revealed by model checking.

During the simulation of our original model that included concurrency we recognized that the execution of the system ran into a deadlock. We didn't anticipate the simulation environment to get stuck in a state of the *CheckPoint* class. The system was waiting for a message from the *Authorization* class. Unfortunately, a request from a different external entity was forwarded to the *CheckPoint* previous to the awaited *Authorization* message. Thus, the message queue was blocked by the new request, which could not be consumed in the current state. Usually this error could be solved by deferring all blocking events in the respective state, but this part of the UML-syntax is not yet supported by our tools. The awareness of this problem lead to the implementation of several synchronization constructs throughout the system. Basically, the modifications assured that all possible events in the queue can be consumed in any state and that they are preserved in local variables until needed. Once we applied our abstraction techniques many of these problems were solved.

Logical errors are another class of errors that we were able to detect in our diagrams. During the process of property verification, we identified errors in the access logic of the *Authorization* class that did not correctly represent the system's access structure. Basically, the verification assured that all possible parameter combinations are covered by the implemented evaluation and that all results correspond to the access matrix. Concrete examples of these errors are shown in the following section along with the properties that revealed the respective problems.

5.2.3 Property verification

The system model that is described in Section 5.2.1 can be used to prove constraints concerning the *Authorization*, *Check Point*, and *Single Access Point* patterns. In order to show that our design correctly represents and enforces those properties, we derived formal properties in LTL (linear time temporal logic) from the informal constraints. Table 4 gives an overview of LTL formulas that we successfully proved to hold for the execution of our system, which is specified by the PROMELA code in Appendix A.1. The ID column in the table shows to which *Security Pattern* each property relates. ID's beginning with *AUTH* denote a relation to the *Authorization* pattern. In a similar fashion, the ID *CHK* indicates a relation to the *Check Point* pattern whereas *SAP* denotes *Single Access Point*.

In the remainder of this section, we describe each property and present our verification results that were obtained with the SPIN model checker.

Properties related to the *Authorization* pattern

In order to verify the implementation of the *Authorization* pattern, we derived eight properties from the system's access matrix shown in Table 3. Four constraints address availability, two confidentiality, and two integrity.

The first four properties (*AUTH1* to *AUTH4*) that are expressed in LTL have the general form: $\Box(p \rightarrow q)$, which reads: it is globally true that expression p infers expression q. This constraint is violated if at any time during the execution of the system, expression p is fulfilled and q is not. Each column in the access matrix corresponds to a constraint.

The properties *AUTH1* and *AUTH2* account for confidentiality as they ensure that no unauthorized read-access is performed. They correspond to the read-access columns in Table 5.

In our properties, we define expression p to cover all situations of the respective column that should lead to a denial of the current request. Those situations are highlighted in Table 5. Defining expression q as *(EntityRequestGenerator.success==0)* ensures that those restricted requests cannot succeed without violating the current property, because p always infers q. Using information from Table 5, we can construct the two LTL-properties shown in Table 6.

Note, for space limitations and readability purposes, we use abbreviations for CheckPoint (CP), CounterMeasure (CM), EntityRequestGenerator (ERG), and SingelAccessPoint (SAP) in our LTL formulas.

The properties *AUTH3* and *AUTH4* improve integrity by checking the system for the existence of unauthorized write-accesses. Hence, our integrity-properties correspond to the columns that concern write-accesses. Those columns, including invalid constellations of request parameters are highlighted in Table 7.

ID	***LTL property***	***Expression p***	***Expression q***	***Expression r***	***Scope***
AUTH1	□(p→q)	(CP.recipient==1) && (CP.accessType==1) && (CP.sender!=1) && (CP.sender!=2)	(ERG.success==0)		Confidentiality
AUTH2	□(p→q)	(CP.recipient==2) && (CP.accessType==1) && (CP.sender!=1) && (CP.sender!=3)	(ERG.success==0)		Confidentiality
AUTH3	□(p→q)	(CP.recipient==1) && (CP.accessType==2) && (CP.sender!=1) && (CP.sender!=2)	(ERG.success==0)		Integrity
AUTH4	□(p→q)	(CP.recipient==2) && (CP.accessType==2) && (CP.sender!=1)	(ERG.success==0)		Integrity
AUTH5	□(p→((◊q) U r))	(CP.recipient==1) && (CP.accessType==1) && ((CP.sender==1) || (CP.sender==2))	(ERG.success==1)	(ERG.waiting==0)	Availability
AUTH6	□(p→((◊q) U r))	(CP.recipient==2) && (CP.accessType==1) && ((CP.sender==1) || (CP.sender==3))	(ERG.success==1)	(ERG.waiting==0)	Availability
AUTH7	□(p→((◊q) U r))	(CP.recipient==1) && (CP.accessType==2) && ((CP.sender==1) || (CP.sender==2))	(ERG.success==1)	(ERG.waiting==0)	Availability
AUTH8	□(p→((◊q) U r))	(CP.recipient==2) && (CP.accessType==2) && (CP.sender==1)	(ERG.success==1)	(ERG.waiting==0)	Availability
CHK1	□(p→(◊q))	(CP.grantAccess==2)	(CM.triggered==1)		General Security
CHK2	□(p→(q U r))	(CP.grantAccess==2)	(ERG.success==0)	(ERG.waiting==0)	Confidentiality, Integrity
CHK3	□(p→((◊q) U r))	(CP.grantAccess==1)	(ERG.success==1)	(ERG.waiting==0)	Availability
SAP1	□(p→(◊q))	(SAP.sender!=0)	(Log.logged==1)		Accountablility

Legend	accessType	∈ (1:read, 2:write)	CP : CheckPoint
	grantAccess	∈ (0:undetermined, 1:granted, 2:denied)	CM : CounterMeasure
	recipient	∈ (1,2)	ERG: ExternalRequestGenerator
	sender	∈ (1,2,3,4)	SAP: SingleAccessPoint

Table 4: System 01: LTL properties

		Internal Entities				Example
		Recipient 1		Recipient 2		
		read	*write*	*read*	*write*	
External Entities	Sender 1	yes	yes	yes	yes	Administrator
	Sender 2	yes	yes	no	no	Owner of Internal Entity 1
	Sender 3	no	no	yes	no	Reader of Internal Entity 2
	Sender 4	no	no	no	no	Unauthorized user
		Auth1→		Auth2→		

Table 5: System 01: Access matrix: Confidentiality

ID	***LTL property***	***Expression p***	***Expression q***
AUTH1	□(p→q)	(CP.recipient==1) && (CP.accessType==1) && (CP.sender!=1) && (CP.sender!=2)	(ERG.success==0)
AUTH2	□(p→q)	(CP.recipient==2) && (CP.accessType==1) && (CP.sender!=1) && (CP.sender!=3)	(ERG.success==0)

Table 6: System 01: LTL-properties for Authorization: Confidentiality

		Internal Entities				Example
		Recipient 1		Recipient 2		
		read	*write*	*read*	*write*	
External Entities	Sender 1	yes	yes	yes	yes	Administrator
	Sender 2	yes	yes	no	no	Owner of Internal Entity 1
	Sender 3	no	no	yes	no	Reader of Internal Entity 2
	Sender 4	no	no	no	no	Unauthorized user
			Auth3→		Auth4→	

Table 7: System 01: Access matrix: Integrity

The LTL-properties shown in Table 8 state that every unauthorized request of write-access remains unsuccessful. Each *no* tag in Table 7 denotes an unauthorized request. In the scope of integrity, we regard only write-access requests and formulate one property for each column. This systematic approach prevents errors during the property generation and facilitates understandability.

The following four availability-related properties follow a comparable approach. They state that each authorized request needs to successfully pass our security mechanisms. Authorized requests

ID	LTL property	Expression p	Expression q
AUTH3	$\Box(p \rightarrow q)$	(CP.recipient==1) && (CP.accessType==2) && (CP.sender!=1) && (CP.sender!=2)	(ERG.success==0)
AUTH4	$\Box(p \rightarrow q)$	(CP.recipient==2) && (CP.accessType==2) && (CP.sender!=1)	(ERG.success==0)

Table 8: System 01: LTL-properties for Authorization: Integrity

are represented by *yes* tags in the access matrix. Those are highlighted in Table 9. A successfully completed request results in the ExternalRequestGenerator switching its local attribute *success* to *one* at least once before it continues to generate a new request.

		Internal Entities				Example
		Recipient 1		Recipient 2		
		read	*write*	*read*	*write*	
External Entities	Sender 1	yes	yes	yes	yes	Administrator
	Sender 2	yes	yes	no	no	Owner of Internal Entity 1
	Sender 3	no	no	yes	no	Reader of Internal Entity 2
	Sender 4	no	no	no	no	Unauthorized user
		Auth5→	Auth7→	Auth6→	Auth8→	

Table 9: System 01: Access matrix: Availability

In order to check for this short change of the *ExternalRequestGenerator.success* attribute we use the following LTL-formula: $\Box(p \rightarrow ((\Diamond q) \cup r))$, which reads: its globally true that p infers eventually q until r. That means: if p becomes true at any time during the system's execution then q needs to become true at least once until r is true.

Our properties are shown in Table 10. Expression p defines circumstances in which a request is valid. By using *(ExternalRequestGenerator.success==1)* for expression q, the constraints require the *success*-attribute to become true at least once until r is met. Hence, expression r defines a boundary for the property evaluation and separates succeeding requests. Once the ExternalRequestGenerator is not anymore waiting for the system (that is: *(ExternalRequestGenerator.waiting==0)*), the previous request is completed. If the request has not been successful by that time then we can be sure that it was not approved by our system. We can summarize that the following constraints are violated if a valid request is not approved by our system.

ID	LTL property	Expression p	Expression q	Expression r
AUTH5	□(p→((◊q) U r))	(CP.recipient==1) && (CP.accessType==1) && ((CP.sender==1) || (CP.sender==2))	(ERG.success==1)	(ERG.waiting==0)
AUTH6	□(p→((◊q) U r))	(CP.recipient==2) && (CP.accessType==1) && ((CP.sender==1) || (CP.sender==3))	(ERG.success==1)	(ERG.waiting==0)
AUTH7	□(p→((◊q) U r))	(CP.recipient==1) && (CP.accessType==2) && ((CP.sender==1) || (CP.sender==2))	(ERG.success==1)	(ERG.waiting==0)
AUTH8	□(p→((◊q) U r))	(CP.recipient==2) && (CP.accessType==2) && (CP.sender==1)	(ERG.success==1)	(ERG.waiting==0)

Table 10: System 01: LTL-properties for Authorization: Availability

After correcting some errors in the authorization logic of our design, all eight authorization properties were successfully verified using the Spin model checker. Table 11 shows our verification results, including stored states, matched states, transitions, search depth and memory usage.

ID	States stored	States matched	Transitions	Depth	Memory usage	verified
AUTH1	66986	57725	124711	30189	36.440 MB	valid
AUTH2	66986	57725	124711	30189	36.440 MB	valid
AUTH3	66986	57725	124711	30189	36.440 MB	valid
AUTH4	66986	57725	124711	30189	36.440 MB	valid
AUTH5	70962	78097	153035	30189	38.603 MB	valid
AUTH6	70962	78097	153035	30189	38.603 MB	valid
AUTH7	70962	78097	153035	30189	38.603 MB	valid
AUTH8	68959	67785	138717	30189	37.514 MB	valid

Table 11: System 01: Authorization: Verification results

During the process of model checking we revealed errors in the state-diagram of the *Authorization* class. Figure 35 shows an excerpt from the old diagram, including the faulty transitions. Accidentally, we defined the wrong recipient in the highlighted transitions.

Instead of *recipient=2* in the left transition and *recipient!=2* in the right transition we specified *recipient=1* (left) and *recipient!=1* (right). This error did not cause the system to deadlock, because the two faulty transitions were still disjunct and complete, but the logic did not correctly represent the access structure of our system. This logical error caused the system to grant unauthorized access, if sender 3 requested read-access to recipient 1. The problem was recognized when property AUTH1 was violated. During property verification SPIN created a message trace that enabled us to locate the error in our design. Furthermore, sender 3 was not allowed to read from recipient

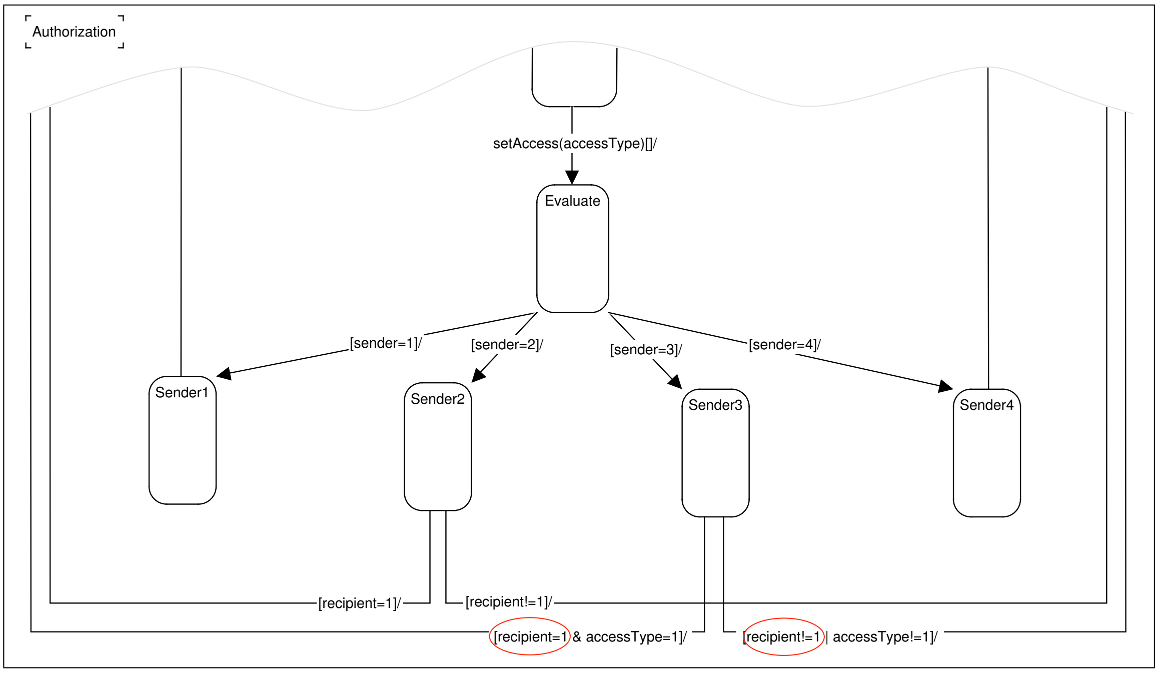


Figure 35: UML state-diagram: System 01: Errors in Authorization class

2 in our system although it has the appropriate rights in our access-matrix (Table 3). This error was revealed during the verification of the availability-constraint AUTH6, also leading to the faulty transitions mentioned before.

Properties related to the *Check Point* pattern

For the verification of the *Check Point* pattern we formulated three constraints, that we present herein.

The first property *CHK1* states that an unauthorized access leads to the activation of a countermeasure. In terms of LTL, this constraint can be expressed as $\Box(p \rightarrow (\Diamond q))$ (it is globally true that p infers eventually q). In his work on specification patterns [10] Dwyer coined the term of *Response* patterns explaining those LTL formulas. Once a starting condition p is met, the expression q must become true at least once during the remaining time of the system execution. Hence, q is a response to p. We use the *CHK1* property to ensure that a countermeasure is triggered (*CounterMeasure.triggered==1*) after the *CheckPoint* denied a request (*CheckPoint.grantAccess==2*).

The *CHK2* property checks if integrity or confidentiality is compromised after the denial of a request. In order to verify that a request remains unsuccessful after its refusal, we use the formula: $\Box(p \rightarrow (q \cup r))$ (it is globally true that p infers q until r). After precondition p (*CheckPoint.grantAccess==2* - stating that the *CheckPoint* denied the current request) is met q needs remain true until it is released by r being fulfilled. With *ExternalRequestGenerator.success==0* as q and *ExternalRequestGenerator.waiting==0* as r, the constraint states that the refused request cannot be successful.

Property *CHK3* embodies an availability-constraint. It verifies that the *CheckPoint's* approval of a request causes the internal entity to respond to the external entity's access request. The property's formula has the same structure as the availability-constraints related to the *Authorization* pattern (*AUTH5* to *AUTH8*): $\Box(p \rightarrow ((\Diamond q) \cup r))$ (it is globally true that p infers eventually q until r). Upon approval of a request (*CheckPoint.grantAccess==1*) that is stated in condition p, the attribute *ExternalRequestGenerator.success* needs to change its value to *one* at least once during the current request whose end is indicated by (ExternalRequestGenerator.waiting==0).

All three *CheckPoint*-related properties (*CHK1*, *CHK2*, and *CHK3*) that are shown in Table 12 were successfully verified using the Spin model-checker. Table 13 points out our verification results.

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

Table 12: System 01: LTL-properties for Check Point

ID	***States stored***	***States matched***	***Transitions***	***Depth***	***Memory usage***	***verified***
CHK1	67337	58490	126178	30189	36.631 MB	valid
CHK2	68075	60488	129652	30189	37.033 MB	valid
CHK3	69387	67049	138837	30189	37.747 MB	valid

Table 13: System 01: Check Point: Verification results

Properties related to the *Single Access Point* pattern

The last property that we verify against our first example system relates to the *Single Access Point* pattern and its logging function. In order to assure that each request is logged we use the response pattern. We check that each external message entering the *SingleAccessPoint* class causes a call of the *Log* class. Analogous to the specification of the *CHK1* property, the following LTL-formula is applied in *SAP1*: $\Box(p\rightarrow(\Diamond q))$ (it is globally true that p infers eventually q). If the precondition p is met then at least once during the runtime of the system q needs to become true. In *SAP1* the precondition is met if a request reaches the *SingleAccessPoint* class and the attribute *sender* is set to the actual request-parameter (*sender!=0*). Thereafter, the constraint requires the system to reach the *Logging* state in *Log* class at least once (◊*(Log.logged==1)*). Table 14 shows the *SAP1* LTL-property.

ID	LTL property	Expression p	Expression q
SAP1	□(p→(◊q))	(SAP.sender!=0)	(Log.logged==1)

Table 14: System 01: LTL-properties for Single Access Point

Similar to the other properties, *SAP1* was verified valid in our example system. Its verification results are outlined in Table 15.

ID	States stored	States matched	Transitions	Depth	Memory usage	verified
SAP1	131882	157281	353035	30189	71.744 MB	valid

Table 15: System 01: Single Access Point: Verification results

5.3 Exemplary verification 02

The second system, presented herein, is an online document management system implementing multilevel security. We use this system to demonstrate the verification of properties inherent to the patterns *Roles*, *Session*, and *Multilevel Security*.

The following sections are structured in a fashion similar to the first system. In Section 5.3.1 we begin with a complete description of our model, including class- and state-diagrams. Then the system's evolution and assumptions are shown in Section 5.3.2. Finally, Section 5.3.3 presents our LTL-properties and verification results.

5.3.1 System design

Similar to Section 5.2.1 this section elucidates the design of our second example system modelled in subset of UML. These models can be automatically transformed into the formal specification shown in Appendix A.2.

On a relatively high abstraction level our models depict an online service that offers access to several documents via the internet. The system stores documents (objects) from two different domains (compartments). Each user (subject) and each document has a security level in each compartment. According to the *Multilevel Security* pattern a subject needs equal or higher classification than an object in all compartments in order to gain access to it. In our example the two compartments: *University* and *Company XY* exist. Figure 36 depicts the system's different objects and their membership in a compartment.

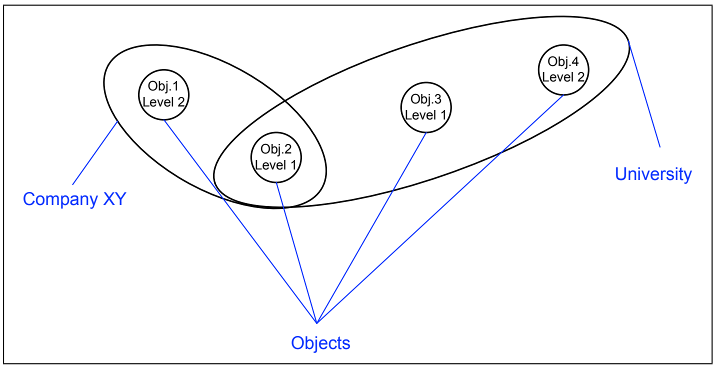


Figure 36: System 02: Objects and Compartments

Object 2 is an example of a research project including confidential data from Company XY and from the University. Thus, this object's classification has a security level higher than zero in each compartment. Only users with security clearance in each domain can access this object. For demonstration purposes, our system implements five users. Each user is assigned a different role that defines the subjects classification. Figure 37 exhibits both the system's subjects and according roles.

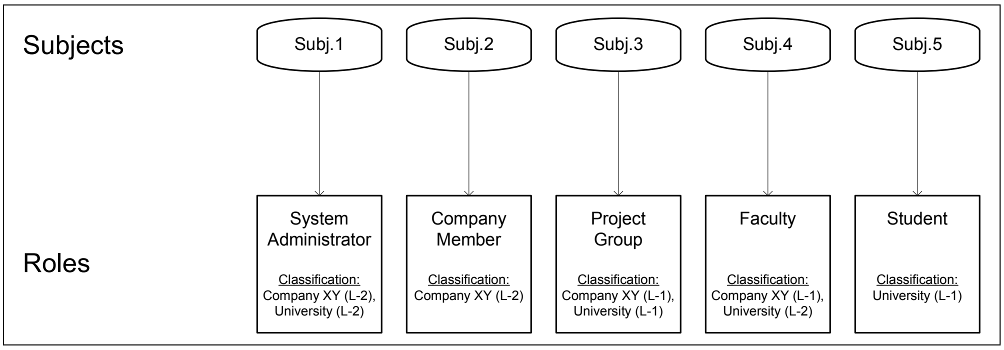

Subjects
Roles
Subj.1
Subj.2
Subj.3
Subj.4
Subj.5
System Administrator
Classification: Company XY (L-2), University (L-2)
Company Member
Classification: Company XY (L-2)
Project Group
Classification: Company XY (L-1), University (L-1)
Faculty
Classification: Company XY (L-1), University (L-2)
Student
Classification: University (L-1)

Figure 37: System 02: Subjects and Roles

In order to reduce the state-space of the system we store objects' and subjects' classifications in one single *Byte*. Figure 38 shows the internal representation of classification information in the system. The first four bits are not used whereas the lower four bits define the compartments' levels in binary notation with 2 bits each.

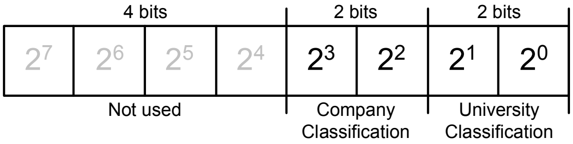
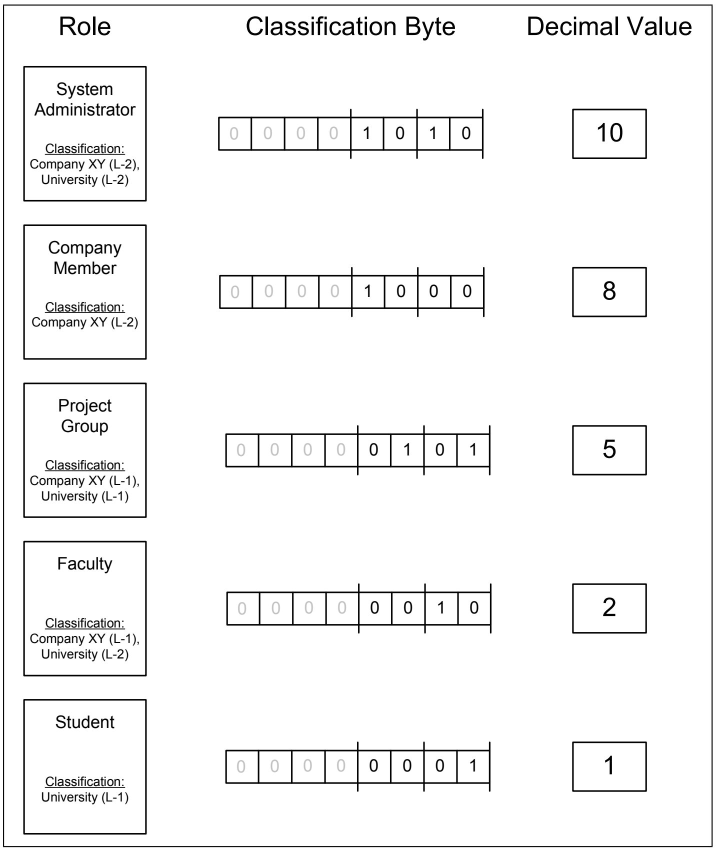


Figure 38: System 02: Internal Representation Object- and Subject-Classifications

The internal representation of the subjects' and objects' classifications are depicted in Figure 39 and Figure 40. Throughout the state-diagrams of our model the *Decimal Value* given in those figures is used to identify the respective object's or subject's classification.



Role	Classification Byte	Decimal Value
System Administrator Classification: Company XY (L-2), University (L-2)	0 0 0 0 1 0 1 0	10
Company Member Classification: Company XY (L-2)	0 0 0 0 1 0 0 0	8
Project Group Classification: Company XY (L-1), University (L-1)	0 0 0 0 0 1 0 1	5
Faculty Classification: Company XY (L-1), University (L-2)	0 0 0 0 0 0 1 0	2
Student Classification: University (L-1)	0 0 0 0 0 0 0 1	1

Figure 39: System 02: Internal representation of Role-Classifications

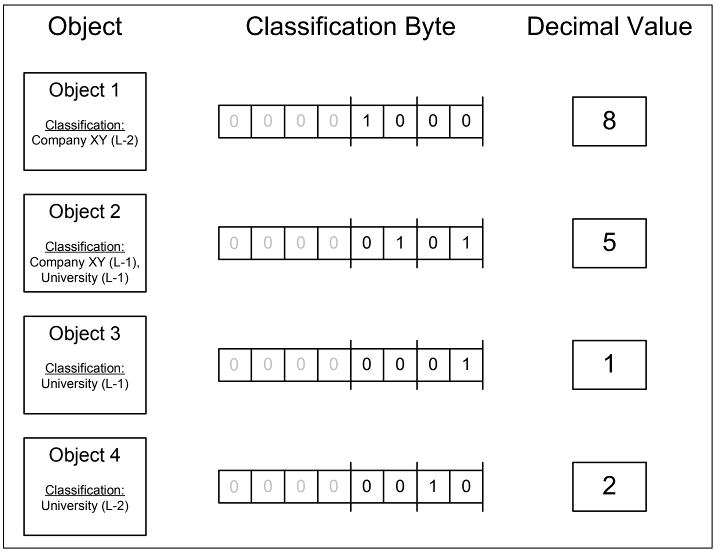


Figure 40: System 02: Internal representation of Object-Classifications

Class structure

The class structure of the system is depicted in Figure 41. Similar to our first system the *_SYSTEMCLASS_* controls the initialization and instantiation process of the system. The *ExternalRequestGenerator* randomly creates messages to simulate external entities that access the system. The *SingleAccessPoint* handles those messages and accounts for the login into the system. Thereby, a *Session* class is initialized with information about the user that is currently logged in. The *Roles* class provides the *Session* with the user's classification. This *SubjectClassification* is required by the *CheckPoint* in order to determine if a request is valid. Furthermore the *CheckPoint* needs the *ObjectClassification* of the requested object to evaluate if it may be accessed. If the *SubjectClassification* dominates the *ObjectClassification* the *CheckPoint* forwards the message to the *InternalObjects* class, which sends a success-message to the *ExternalRequestGenerator*. Invalid requests are answered by an error message to the *ExternalRequestGenerator*.

Attributes and ranges

For data transfer purposes and keeping status information, we use a limited number of attributes. Figure 42 depicts those variables and possible values arranged in three groups. The first group *Sender/Receiver Identity* displays the different objects' and subjects' representation throughout the system. *System Status* contains attributes that carry information on the current request and user. The last group *Classifications* shows how classification information is stored internally.

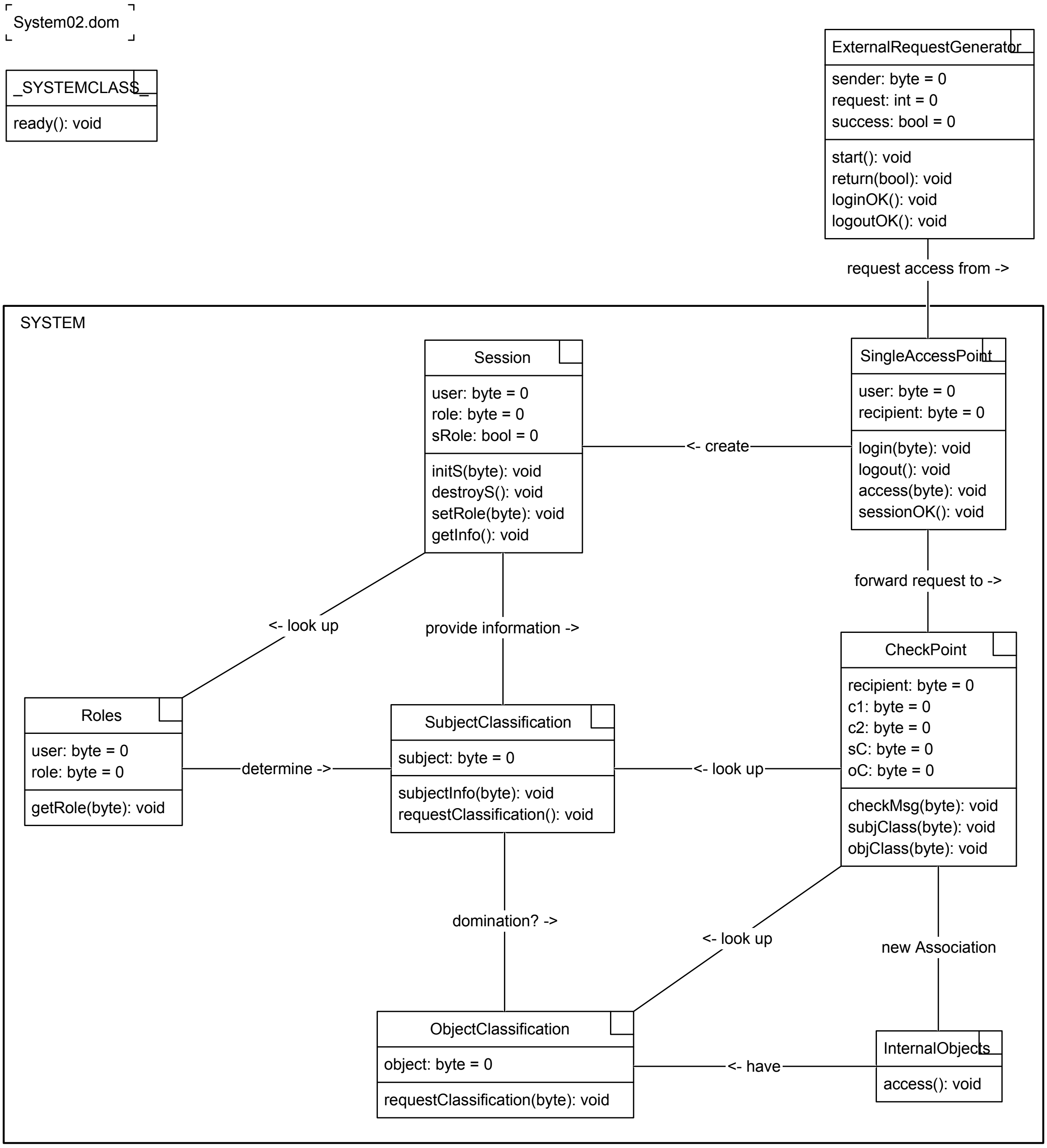


Figure 41: UML class-diagram: System 02

Sender/Receiver Identity

sender, subject, user =	
0	(Undetermined)
1	(User 1)
2	(User 2)
3	(User 3)
4	(User 4)
5	(User 5)

object, recipient, request =	
0	(Undetermined)
1	(Internal Object 1)
2	(Internal Object 2)
3	(Internal Object 3)
4	(Internal Object 4)

System Status

role=	
0	(Undetermined)
1	(System Administrator)
2	(Company Member)
3	(Project Group)
4	(Faculty)
5	(Student)

success=	
0	(Request unsuccessful)
1	(Request successful)

sRole=	
0	(Session will request role information)
1	(Role information has been set)

Classifications

oC, c2=	
0	(Undetermined)
8	(Classification Object 1)
5	(Classification Object 2)
1	(Classification Object 3)
2	(Classification Object 4)

sC, c1=	
0	(Undetermined)
10	(Classification System Administrator)
8	(Classification Company Member)
5	(Classification Project Group)
2	(Classification Faculty)
1	(Classification Student)

Figure 42: System 02: Attributes and ranges

In the remainder of this section we present UML state-diagrams of each class thereby elucidating the behavior of our system.

SYSTEMCLASS state machine

Proceeding in alphabetical order we depict the _SYSTEMCLASS_ in Figure 43. An instance of the _SYSTEMCLASS_ is created and run by the time our system is started. All remaining classes are instantiated by the _SYSTEMCLASS_ state machine. Hence, the _SYSTEMCLASS_ can be interpreted as a startup script of our system. Once all state machines are created, the *start()* event allows the *ExternalRequestGenerator* to begin its regular task: creating requests.

CheckPoint state machine

The *CheckPoint* UML state-diagram is depicted in Figure 44. Upon its creation the state machine enters the *Main Idle* state thereby resetting all internal variables to *zero*. Then the *CheckPoint* waits for a message from the *SingleAccessPoint* to handle. Once a request arrives, the *CheckPoint* stores the *recipient* and obtains classification data from the classes *SubjectClassification* and *ObjectClassification*. By the time the *CheckPoint* reaches the state *ExtractSubCLevel* the classification

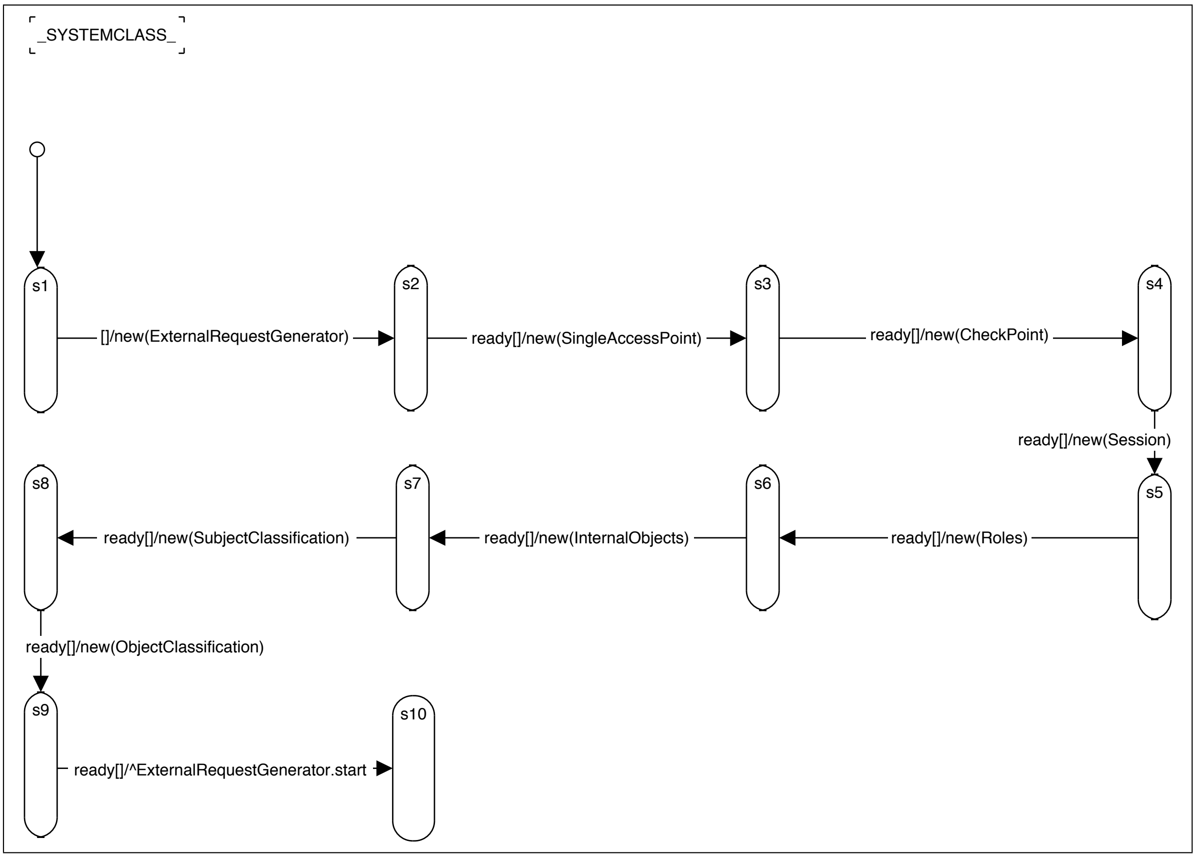

SYSTEMCLASS
s1
s2
s3
s4
s5
s6
s7
s8
s9
s10
[]/new(ExternalRequestGenerator)
ready[]/new(SingleAccessPoint)
ready[]/new(CheckPoint)
ready[]/new(Session)
ready[]/new(Roles)
ready[]/new(InternalObjects)
ready[]/new(SubjectClassification)
ready[]/new(ObjectClassification)
ready[]/^ExternalRequestGenerator.start

Figure 43: UML state-diagram: System 02: _SYSTEMCLASS_ (for initialization)

information is stored in the attributes *sC* (subject classification) and *oC* (object classification). On the following transitions the object's and subject's level for the compartment *Company XY* are extracted and saved in the variables *c1* and *c2*. If the object's level dominates the subject's level the state *SubjectsDominatesNot* is entered and the request is invalid. If the subject's level in compartment *Company XY* is adequate then the levels for the *University* compartment are extracted into *c1* and *c2*. In state *CheckUniversity* subject's and object's University-levels are compared finally classifying the request as valid (state *SubjectDominates*) or invalid (state *SubjectDominatesNot*). Valid messages are granted access to the *InternalObjects* class whereas invalid requests result in an error message to the *ExternalRequestGenerator*. Finally, the *CheckPoint* reaches the *Main Idle* state and is ready to handle the next message.

ExternalRequestGenerator state machine

Figure 45 shows the *ExternalRequestGenerator* class. Once a *start()* event from the _SYSTEMCLASS_

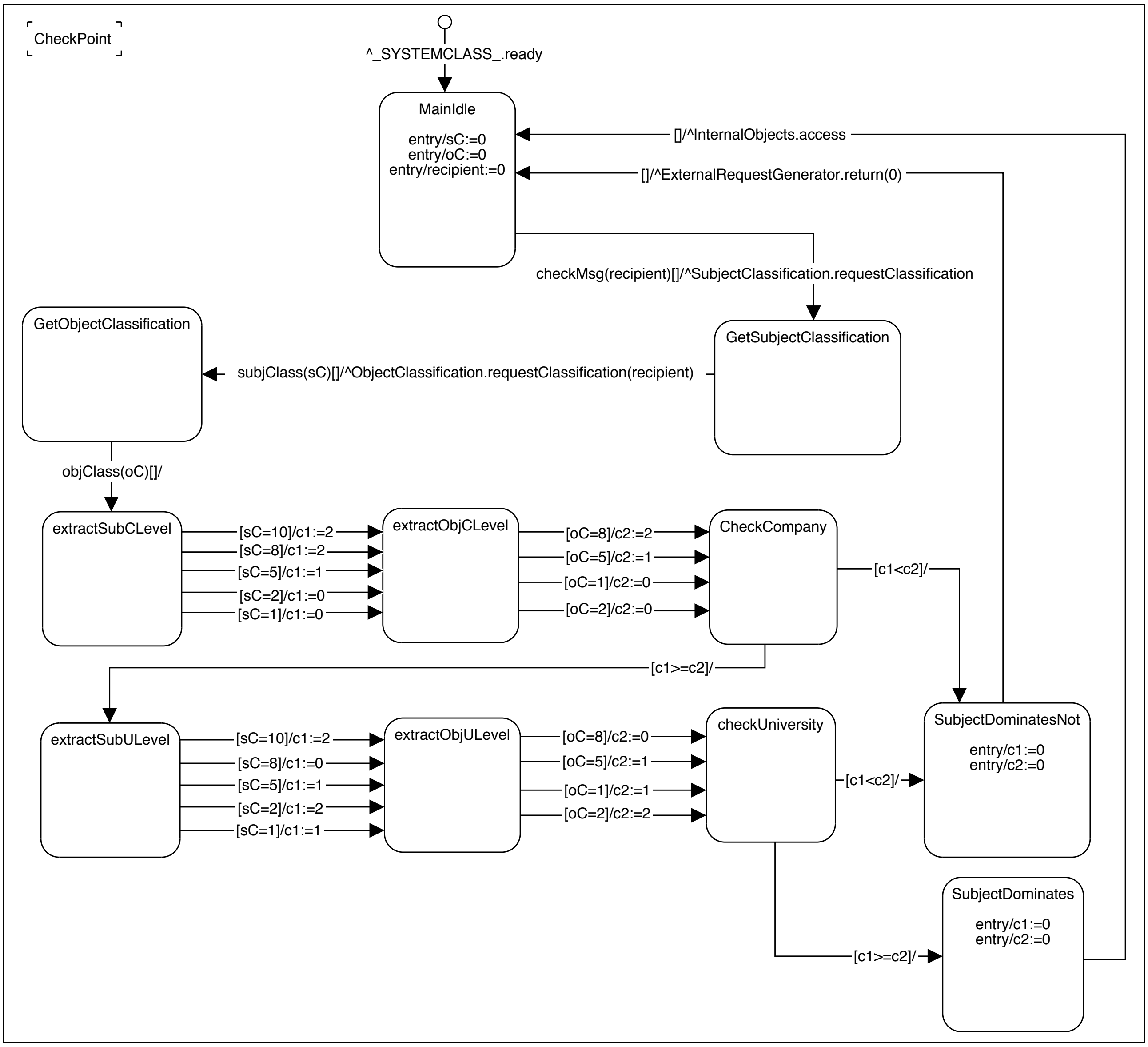


Figure 44: UML state-diagram: System 02: CheckPoint class

arrives the state machine enters the *LoggedOut* state. Then the *ExternalRequestGenerator* nondeterministically chooses a possible sender for the next message. After this a *login(sender)* request is transmitted to the *SingleAccessPoint*. Thereafter the state machine waits for the acknowledgement of the system before state *LoggedIn* can be entered. Upon confirmation the *ExternalRequestGenerator* has two options. The first possibility is to file a *Logout* message and to return to the *LoggedOut* state. The second transition creates an access request by choosing a recipient and sending an *access(recipient)* message to the *SingleAccessPoint*. Given the second case the *ExternalRequestGenerator* waits for the system to acknowledge the request and returns to the *LoggedIn* state. From there the state machine again has to choose whether to log out the current user or to send more *access(recipient)* requests.

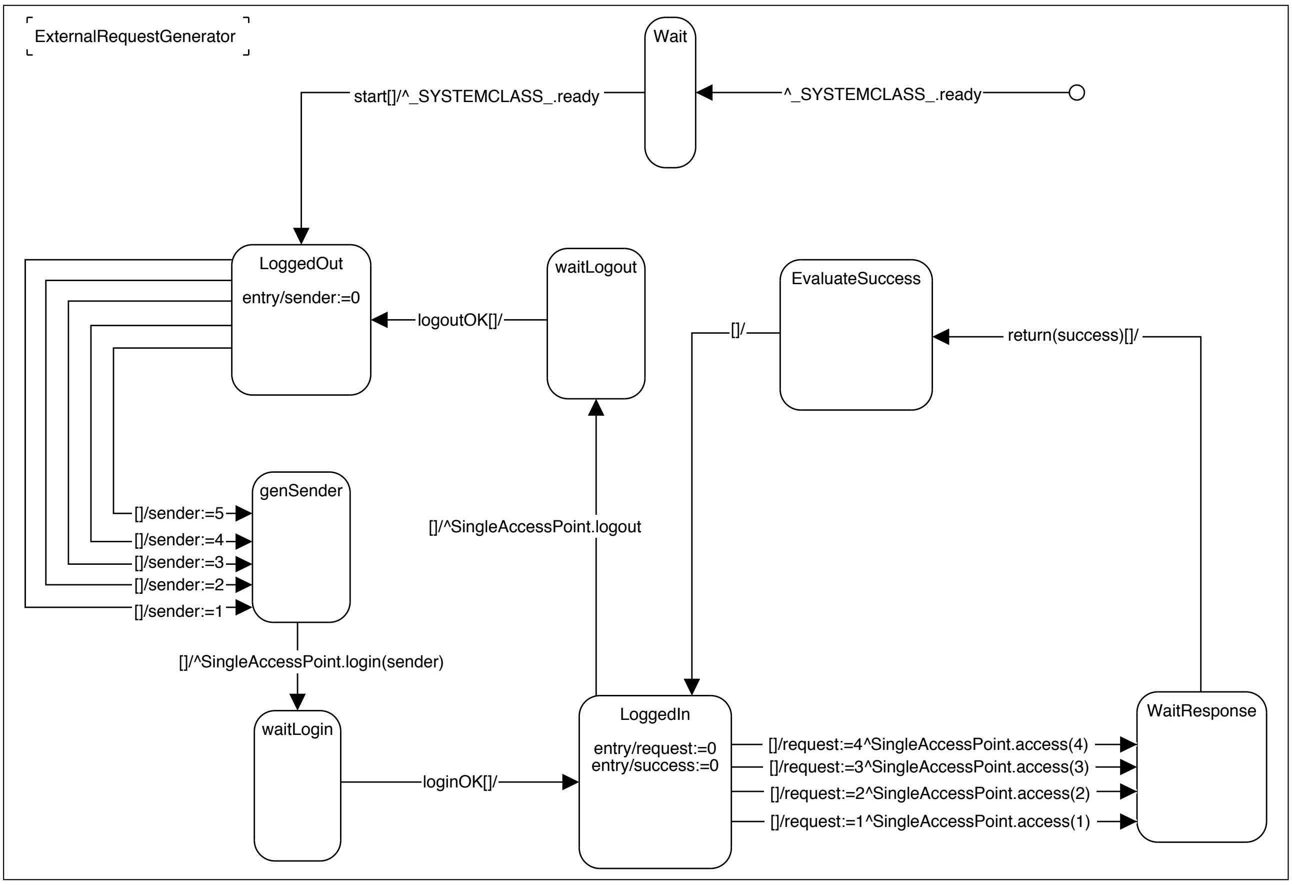


Figure 45: UML state-diagram: System 02: ExternalRequestGenerator class

InternalObjects state machine

The *InternalObjects* class abstracts all the internal objects inside our system that may be accessed.

Its state machine has only one state with the purpose to return a success message (*return(1)*) to the *ExternalRequestGenerator* if an *access()* message arrives. Figure 46 depicts the *InternalObjects* state diagram.

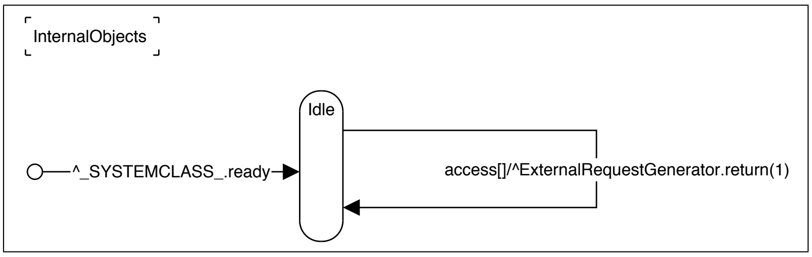


Figure 46: UML state-diagram: System 02: InternalObjects class

ObjectClassification **state machine**

The *ObjectClassification* answers requests from the *CheckPoint*. Thus, it remains in the *Idle* state until a *requestClassification(object)* message arrives. The parameter *object* specifies which classification the *CheckPoint* needs. Upon a request the state machine enters the *Evaluate* state. Then the *ObjectClassification* returns the respective classification and reenters the *Idle* state, thereby resetting the internal *object* attribute to *zero.* Figure 47 exhibits the *InternalObjects's* UML state-diagram.

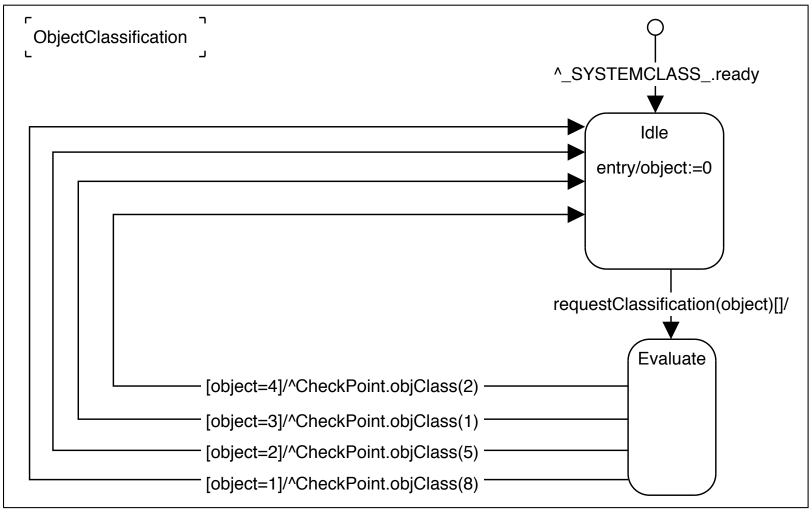

ObjectClassification
^_SYSTEMCLASS_.ready
Idle
entry/object:=0
requestClassification(object)[]/
Evaluate
[object=4]/^CheckPoint.objClass(2)
[object=3]/^CheckPoint.objClass(1)
[object=2]/^CheckPoint.objClass(5)
[object=1]/^CheckPoint.objClass(8)

Figure 47: UML state-diagram: System 02: ObjectClassification class

Roles state machine

The *Roles* class depicted in Figure 48 assigns a role to each user according to Figure 37 . This information is requested by the *Session* class, which stores the role for further use. Upon creation of the state machine, internal attributes are reset to *zero* and state *Idle* becomes active. Receiving a *getRole(user)* request, the *Roles* class enters the *getUser* state while storing the request parameter *user*. Dependent on the user either one of the states *SysAdmin*, *Company*, *Project*, *Faculty*, or *Student* is entered. The next transition sets the respective role and activates state *determineRole*. Finally, the role is forwarded to the *Session* class and the *Roles* state machine returns to the *Idle* state.

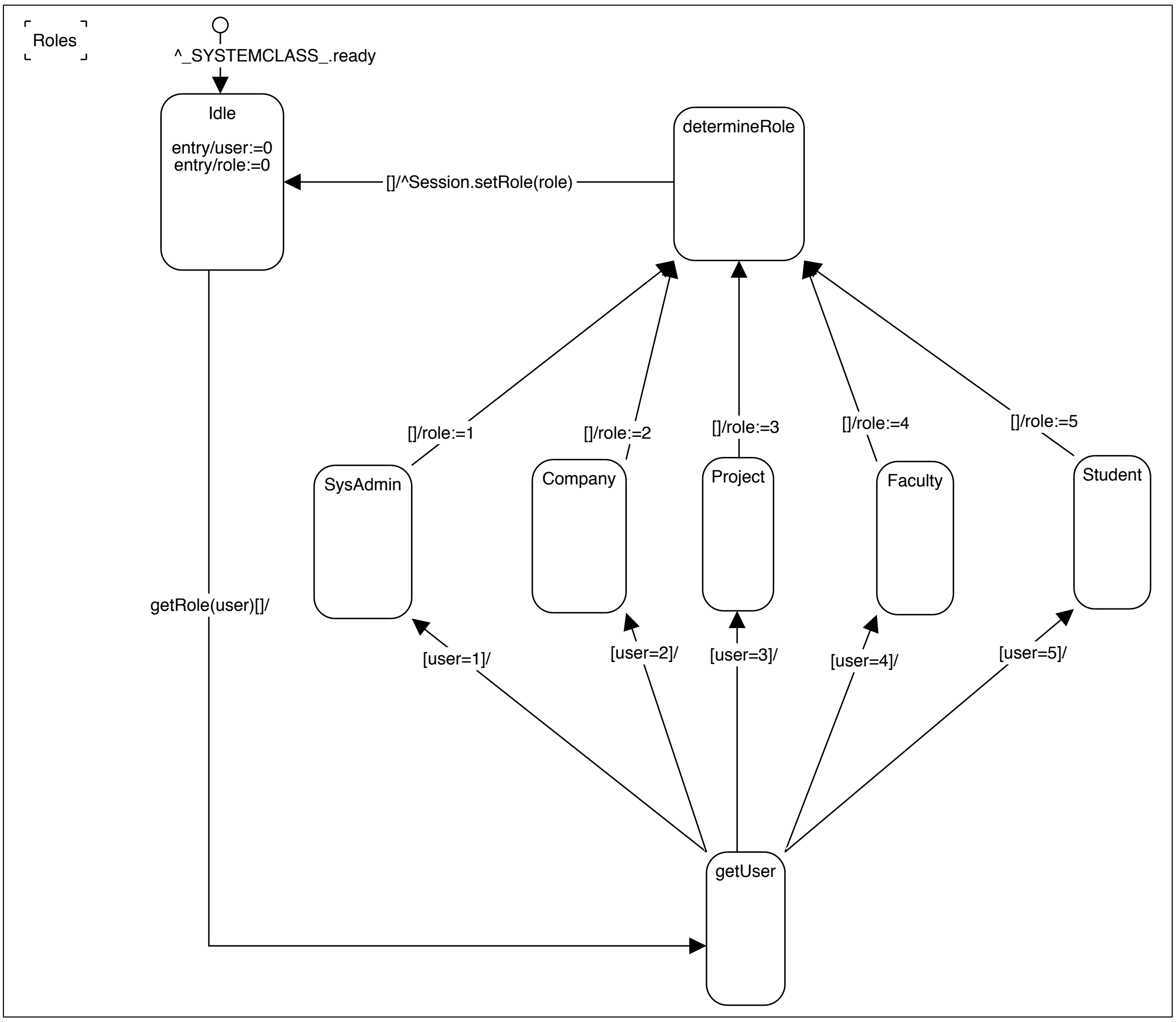


Figure 48: UML state-diagram: System 02: Roles class

Session **state machine**

The *Session* class is created upon the login of an user. It stores and provides information about the status of the system concerning the current user. Entering the *Init* state, *user* and *role* variables are cleared. An *InitS(user)* message from the *SingleAccessPoint* causes the *Session* to start the initialization process. The current user is stored in attribute *user* and the *WaitRole* state is activated while sending a request to the *Roles* class. For verification purposes we toggle the *sRole* flag to *one* indicating that the user's role will be set on the next transition. Once the *Roles* class returns the role information it is stored while entering the *Idle* state. Then, the *Session* accepts and handles *getInfo()* requests until the logout signal *destroyS()* arrives. *Session's* states and transitions are depicted in Figure 49.

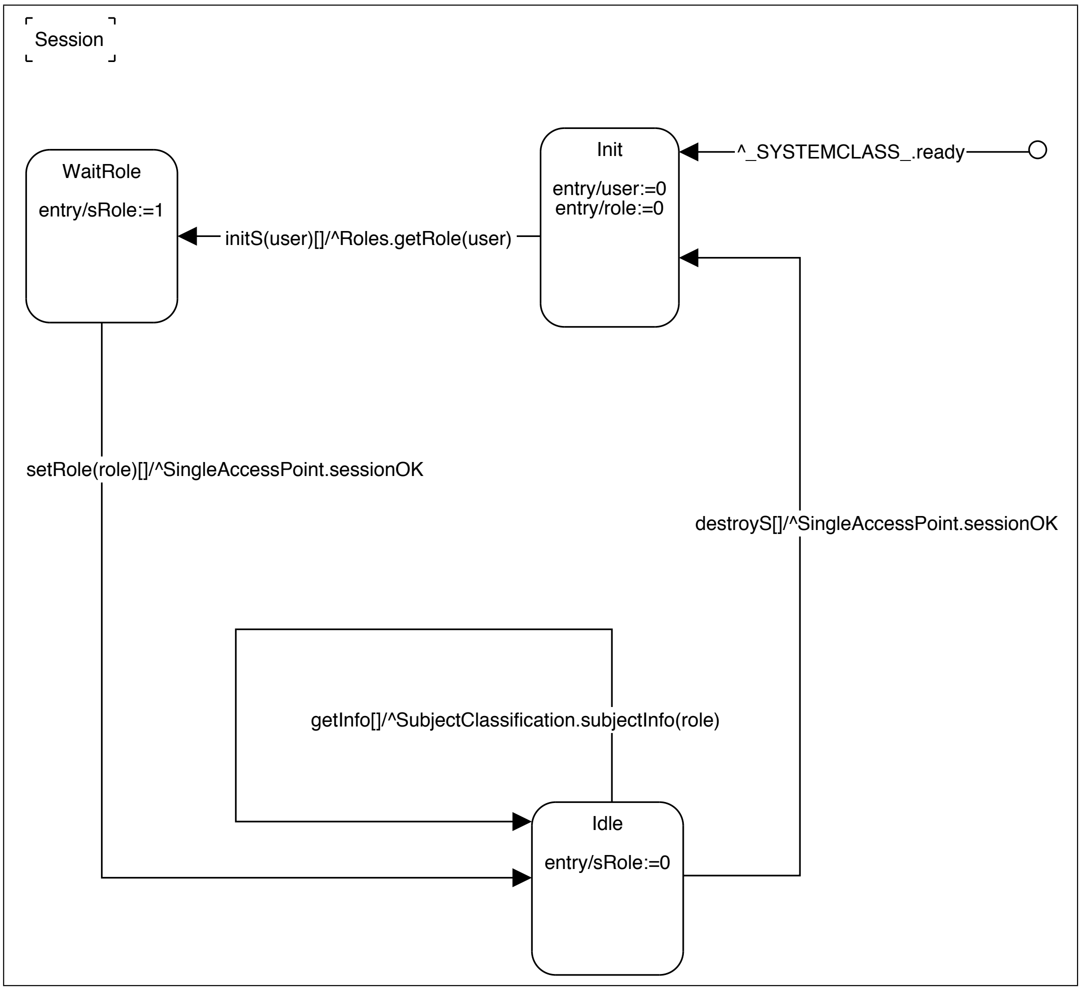

Session
WaitRole
entry/sRole:=1
Init
entry/user:=0
entry/role:=0
^_SYSTEMCLASS_.ready
initS(user)[]/^Roles.getRole(user)
setRole(role)[]/^SingleAccessPoint.sessionOK
destroyS[]/^SingleAccessPoint.sessionOK
getInfo[]/^SubjectClassification.subjectInfo(role)
Idle
entry/sRole:=0

Figure 49: UML state-diagram: System 02: Session class

SingleAccessPoint state machine

The *SingleAccessPoint* class handles all inbound communication from outside the system. Furthermore the *SingleAccessPoint* controls the login process. Figure 50 displays the *SingleAccessPoint* state machine. Starting in state *LoggedOut* the *SingleAccessPoint* class resets internal attributes to *zero* and waits for a *login(user)* message from the *ExternalRequestGenerator*. The arrival of a login request causes the state machine to send an *initS(user)* message to the *Session* class. Then the *WaitSession* state becomes active until the *Session* confirms the request. After this the *SingleAccessPoint* enters the *LoggedIn* state. Dependent on incoming messages from the *ExternalRequestGenerator* the state machine either forwards *access(user)* requests to the *CheckPoint* or logs the current user out. A *logout* message causes the *SingleAccessPoint* to terminate the current session and return to the *LoggedOut* state. Before the logout is completed the *SingleAccessPoint* waits for the confirmation of the *Session* class and then informs the *ExternalRequestGenerator* of its status.

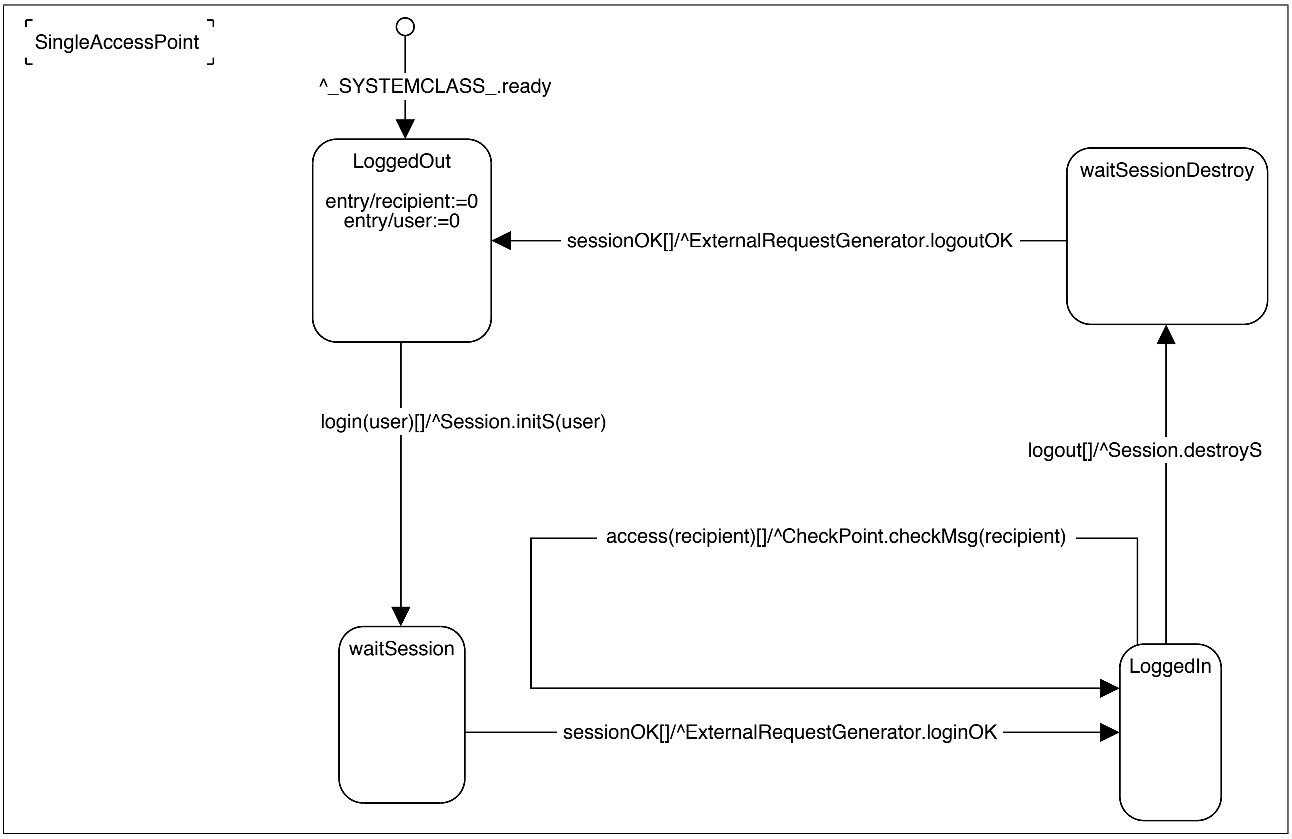

SingleAccessPoint
^_SYSTEMCLASS_.ready
LoggedOut
entry/recipient:=0
entry/user:=0
sessionOK[]/^ExternalRequestGenerator.logoutOK
waitSessionDestroy
login(user)[]/^Session.initS(user)
logout[]/^Session.destroyS
access(recipient)[]/^CheckPoint.checkMsg(recipient)
waitSession
LoggedIn
sessionOK[]/^ExternalRequestGenerator.loginOK

Figure 50: UML state-diagram: System 02: Roles class

***SubjectClassification* state machine**

The *SubjectClassification* handles *requestClassification()* requests from the *CheckPoint* thereby providing it with the classification of the currently logged in subject. In order to determine the correct classification the *SubjectClassification* queries user information from the *Session*. Reaching the state *Evaluation* all required information is available and the next transition returns the respective classification to the *CheckPoint*. Finally the state machine reenters the *Idle* state and is ready to process the next request.

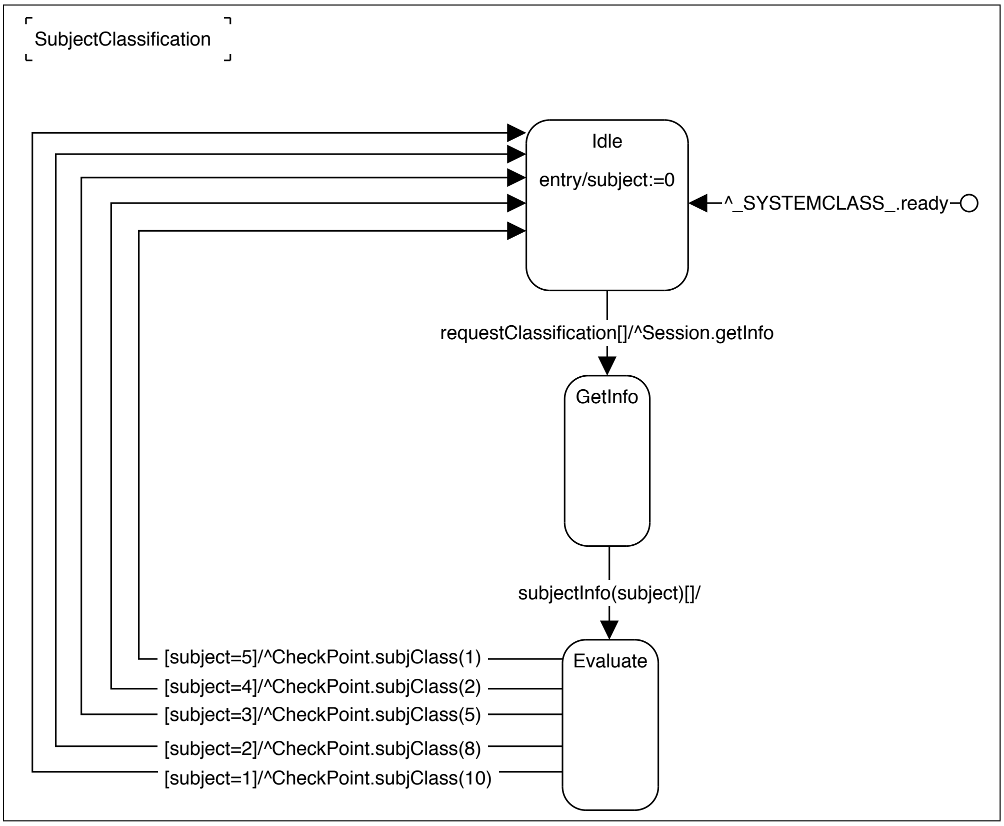

SubjectClassification
Idle
entry/subject:=0
^_SYSTEMCLASS_.ready
requestClassification[]/^Session.getInfo
GetInfo
subjectInfo(subject)[]/
Evaluate
[subject=5]/^CheckPoint.subjClass(1)
[subject=4]/^CheckPoint.subjClass(2)
[subject=3]/^CheckPoint.subjClass(5)
[subject=2]/^CheckPoint.subjClass(8)
[subject=1]/^CheckPoint.subjClass(10)

Figure 51: UML state-diagram: System 02: SubjectClassification class

5.3.2 Model Refinement

The system presented in the previous section was subject to several changes before it reached its final version. This section describes the most important revisions. Furthermore, we address assumptions that were made during the design of our system. Similar to the corresponding section regarding System 01, we begin with abstraction techniques and then elucidate errors that were resolved during the design process.

Abstraction Techniques

The following abstraction techniques have already been described in Section 5.2.2: We reused the idea of an external request generator to reduce the number of classes and also abstracted from internal objects. In order to obtain a smaller state space we choose the smallest possible ranges for all attributes. Furthermore, variables are reset to an initial value as soon as possible.

Besides those already used techniques from System 01, we abstracted any functionality that is not necessary to illustrate the concerned security patterns. Those abstractions include the following design decisions: We implement no invalid users that try to log into the system. Thus, no authentication of the different users is implemented in the system. Furthermore, we use roles for illustration purposes although the amount of five different users doesn't justify the additional effort itself.

Errors detected

During the simulation and verification of our system using the SPIN model checker two deadlocks were found in our models. First, a older version of the *CheckPoint* contained a faulty transition guard. Hence, an invalid state was reached causing the system to deadlock. A second error was induced by a synchronization problem between the *SingleAccessPoint* class and the *ExternalRequestGenerator* class. A previous version of the *ExternalRequestGenerator* had no *WaitLogout* state. Thus, a new *login(user)* request could reach the *SingleAccessPoint* and block its queue before the logout is confirmed by the *Session* class. Figure 52 depicts the message trace from SPIN that documents this error. As we can see in the sequence diagram, Message 108 needs to reach the *SingleAccessPoint* before message 105. Otherwise the *SingleAccessPoint* queue is blocked and our system deadlocks. To resolve the problem we include a wait-state into the *ExternalRequestGenerator* class and thereby force the system to complete the logout before filing a new request.

5.3.3 Property verification

In this section we check our second system against constraints concerning the *Multilevel Security*, *Roles*, and *Session* patterns. Table 16 gives an overview of LTL formulas that we successfully verified. The PROMELA code of the second system is given in Appendix A.2.

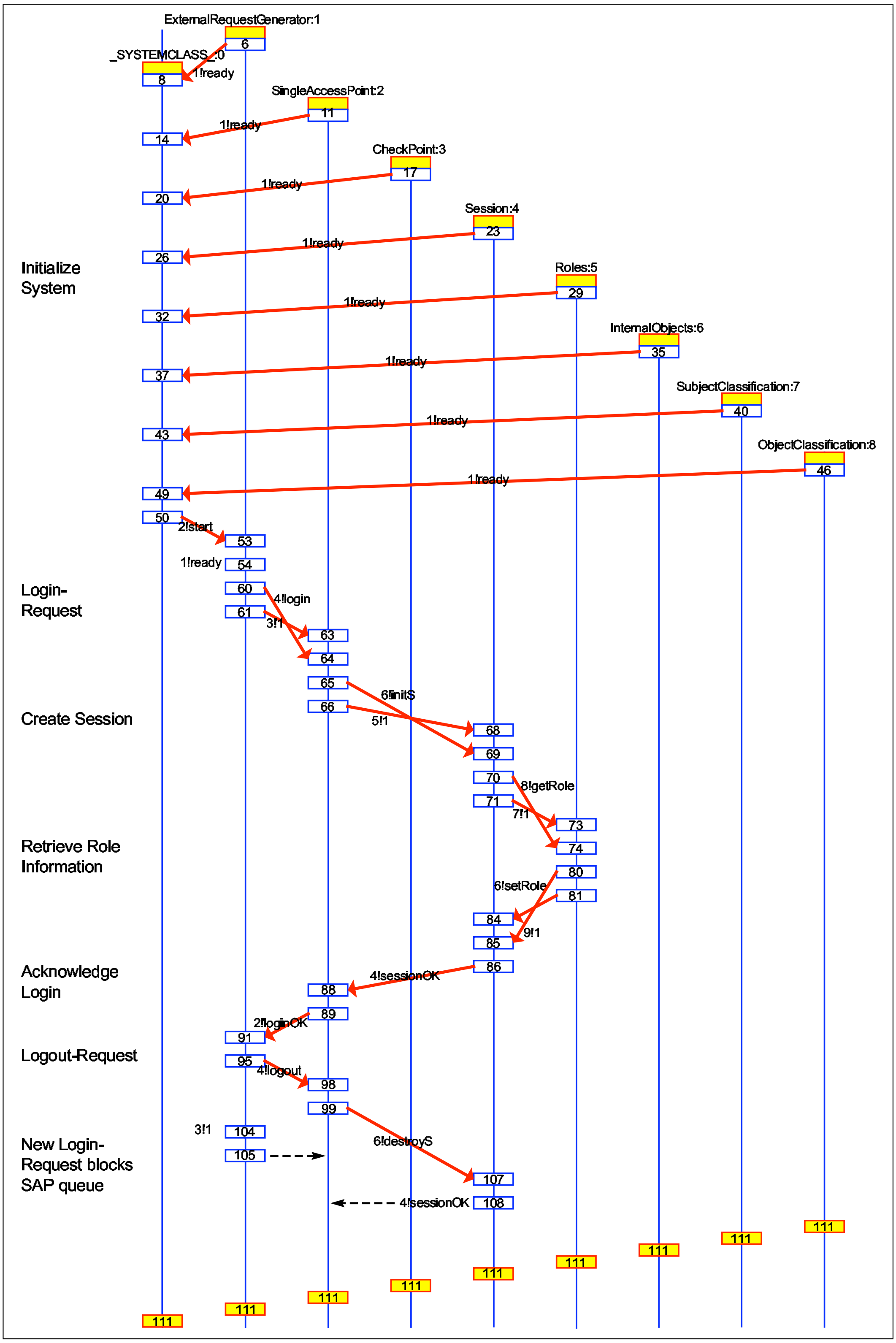


Figure 52: SPIN Output: Message Trace to Deadlock

The remainder of this section presents our properties in the order given in Table 16. We begin with the verification of properties related to the *Multilevel Security* pattern followed by the *Session* and *Roles* patterns.

Properties related to the ***Multilevel Security*** pattern

All properties that relate to the *Multilevel Security* pattern have an ID starting with *MLS* followed by a number in the range of 1 to 9. Their goal is to prove that our system meets the rules of multilevel access. This goal can be refined into two constraints the model has to fulfill:

1. If a subject dominates an object it has access to the object (Availability).
2. If a subject does not dominate an object it has no access to the object (Confidentiality, Integrity).

In order to prove these requirements for our given objects and subjects, we derive nine properties. Five properties relate to the first constraint, four properties relate to the second constraint. Table 17 depicts the Access-Matrix of our system. On the basis of this table we formulate two properties for each column (user). The first (availability) property checks if all *yes* cases of the respective user have access in our system. The second property (confidentiality, integrity) verifies that all *no* cases of that subject lead to a refusal of access. As our first user has super-user privileges we do not have a *no* case. Thus, there is a total number of nine properties instead of ten.

In terms of LTL our availability properties have the general structure: $\Box(p \rightarrow ((\Diamond q) \cup r))$: *It is globally true that p infers eventually q until r.* This formula consists of three expressions linked with temporal operators:

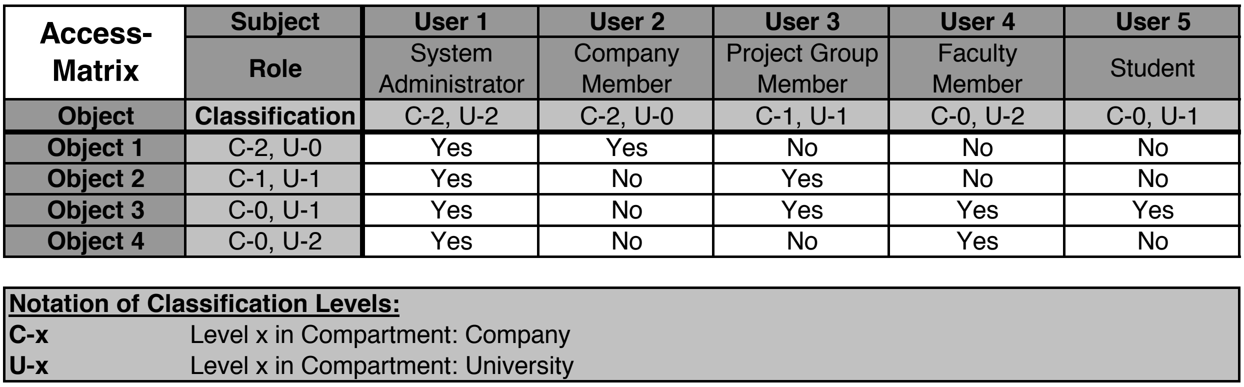
- p is a trigger that determines when this requirement becomes applicable.
- r is the fulfillment of the requirement, and
- q is a condition that must eventually become true in the interim.

Expression q defines in which cases our property is checked. In MLS4, p is defined by the term (ExternalRequestGenerator.sender==4) ∧ ((ExternalRequestGenerator.request==3) ∨ (ExternalRequestGenerator.request==4)). Thus, the property is applicable if the sender of a request is

ID	LTL property	Expression p	Expression q	Expression r	Scope
MLS1	□(p→((◊q) U r))	(ERG.sender==1) && (ERG.request!=0)	(ERG.success==1)	(ERG.request==0)	Availability
MLS2	□(p→((◊q) U r))	(ERG.sender==2) && (ERG.request==1)	(ERG.success==1)	(ERG.request==0)	Availability
MLS3	□(p→((◊q) U r))	(ERG.sender==3) && ((ERG.request==2) || (ERG.request==3))	(ERG.success==1)	(ERG.request==0)	Availability
MLS4	□(p→((◊q) U r))	(ERG.sender==4) && ((ERG.request==3) || (ERG.request==4))	(ERG.success==1)	(ERG.request==0)	Availability
MLS5	□(p→((◊q) U r))	(ERG.sender==5)&&(ERG.request==3)	(ERG.success==1)	(ERG.request==0)	Availability
MLS6	□(p→(q U r))	(ERG.sender==2) && ((ERG.request==2) || (ERG.request==3) || (ERG.request==4))	(ERG.success==0)	(ERG.request==0)	Confidentiality/ Integrity
MLS7	□(p→(q U r))	(ERG.sender==3) && ((ERG.request==1) || (ERG.request==4))	(ERG.success==0)	(ERG.request==0)	Confidentiality/ Integrity
MLS8	□(p→(q U r))	(ERG.sender==4) && ((ERG.request==1) || (ERG.request==2))	(ERG.success==0)	(ERG.request==0)	Confidentiality/ Integrity
MLS9	□(p→(q U r))	(ERG.sender==5) && ((ERG.request==1) || (ERG.request==2) || (ERG.request==4))	(ERG.success==0)	(ERG.request==0)	Confidentiality/ Integrity
ROLE	□(p→((◊q) U r))	(ROL.user!=0)	(ROL.user==ROL.role)	(ROL.user==0)	Confidentiality/ Integrity
SESS	□(p→((◊q) U r))	(SES.sRole==1)	(SES.user==SES.role)	(SES.sRole==0)	Confidentiality/ Integrity

Legend ERG: ExternalRequestGenerator
ROL: Roles
SES: Session

Table 16: System 02: LTL properties



Access-Matrix	Subject	User 1	User 2	User 3	User 4	User 5
	Role	System Administrator	Company Member	Project Group Member	Faculty Member	Student
Object	Classification	C-2, U-2	C-2, U-0	C-1, U-1	C-0, U-2	C-0, U-1
Object 1	C-2, U-0	Yes	Yes	No	No	No
Object 2	C-1, U-1	Yes	No	Yes	No	No
Object 3	C-0, U-1	Yes	No	Yes	Yes	Yes
Object 4	C-0, U-2	Yes	No	No	Yes	No

Notation of Classification Levels:

C-x	Level x in Compartment: Company
U-x	Level x in Compartment: University

Table 17: System 02: Access-Matrix

User 4 and either one of Object 3 or Object 4 is the recipient. Expression q is defined by (ExternalRequestGenerator.success==1) and r by (ExternalRequestGenerator.request==0). Hence, the request must be granted access (success==1) before the next request begins (request==0).

The confidentiality and integrity properties have a slightly different structure than the availability properties. They follow the *Precedence* specification pattern by Dwyer [10] and have the LTL formula: $\Box(p \rightarrow (q \cup r))$: *It is globally true that p infers q until r.* The difference to the availability constraint is that expression q has to remain fulfilled all the time (not only once) until the releasing property r becomes true. This modification ensures that condition q stays true for the complete time of a request. In each property from MLS6 to MLS9 expression p defines the *no* cases of one column in the access matrix. Expression q:=(ExternalRequestGenerator.success==0) states that the request remains unsuccessful. r defines the end of the current request (ExternalRequestGenerator.request==0).

All nine properties shown in Table 16 were successfully verified against the Promela specification of System 02. Table 18 presents the verification results.

Properties related to the *Session* pattern

The *Session* pattern provides a place where information inherent to the currently logged in user can be kept. In order rely on the information we need to assure that the data is consistent at any time. In our system a subject has the same id as its associated role. Thus, it is easy to formulate the requirement (subject==role) that must be true once an user is logged into the system.

In terms of LTL we can use the same structure as the previously presented availability property:

ID	States stored	States matched	Transitions	Depth	Memory usage	verified
MLS1	863	1509	2604	556	37,369 MB	valid
MLS2	689	397	1144	556	37,267 MB	valid
MLS3	747	871	1734	556	37,267 MB	valid
MLS4	747	1083	1946	556	37,267 MB	valid
MLS5	689	605	1352	556	37,267 MB	valid
MLS6	688	654	1399	556	37,267 MB	valid
MLS7	666	422	1123	556	37,267 MB	valid
MLS8	663	340	1035	556	37,267 MB	valid
MLS9	682	490	1223	556	37,267 MB	valid

Table 18: System 02: Verification Results Multilevel Security-Properties

$\square(p\rightarrow((\Diamond q)\cup r))$: *It is globally true that p infers eventually q until r.* For the enabling and releasing condition we specifically implemented an attribute (*sRole*) that reflects the *Session*'s status. Once this variable becomes *one* (condition p) the property is triggered until the variable changes to *zero* (condition r). In the interim the invariant (Session.user==Session.role) needs to be fulfilled eventually.

Table 19 shows the verification results for this property.

ID	States stored	States matched	Transitions	Depth	Memory usage	verified
SESS	701	363	1134	556	37,267 MB	valid

Table 19: System 02: Verification Results Session-Properties

Properties related to the *Roles* pattern

The *Roles* pattern reduces administrative effort in multiuser systems by grouping frequently used privileges into roles. In our system the *Roles* class associates a subject with a certain role. This role determines the subject's classification and thereby its access rights. A correct mapping of roles and users is crucial in order to prevent misuse. In our ROLE property we use a similar approach as in the *Session* class. We check the invariant q:=(Roles.user==Roles.role) using the LTL formula $\square(p\rightarrow((\Diamond q)\cup r))$: *It is globally true that p infers eventually q until r.* The property is triggered by expression $p := (Roles.user! = 0)$ and released by expression $r := (Roles.user == 0)$. Thus, every time the user variable is set the property checks if the role attribute eventually takes the same value as the user before the mapping is completed. Using property ROLE, we showed that the mapping of users and roles is conducted correctly in the *Roles* class at any time. Table 20 displays the verification results produced by SPIN v4.02.

ID	***States stored***	***States matched***	***Transitions***	***Depth***	***Memory usage***	***verified***
ROLE	666	253	954	556	37,267 MB	valid

Table 20: System 02: Verification Results Roles-Properties

Conclusion

The preceding sections gave a brief understanding of how we can augment security patterns with formal specifications to enhance their usefulness. Although, most presented properties relate to our specific systems the capability and potential of this approach became apparent.

6 Discussion

This paper has presented how the pattern approach can be adopted to fit the domain of security. We proposed a security pattern template that addresses the needs inherent to the development of security-critical systems thereby contributing a number of new aspects to security patterns. Our template identifies security-specific consequences of applying a pattern and formally specifies security-oriented constraints. When applying a pattern related security-specific development principles have been identified to help developers new to addressing security issues. Using our patterns, insight into how security concerns can be modelled and analyzed is conveyed.

When used with previous UML formalization work [22] and tool suite [8] the formal analysis techniques presented in this paper can be a step towards more secure systems. Surely their application is no "silver bullet" that guarantees security. Furthermore, limitations inherent to the complexity of exhaustive model checking disable formal verification as a means of checking large systems. Upon implementing complex, concurrent systems the limitations of computation are quickly reached. Thus, formal analysis should be facilitated to rigorously scrutinize security-critical parts of a system.

This paper has shown that the pattern approach has many advantages that favor its use in the domain of software security. Using patterns, security aspects can be addressed during early stages of software development, expert knowledge can be conveyed and reuse facilitated. Thus, error-prone post-design security modifications can be reduced contributing to the quality of software.

Further research may explore how we can incorporate constraints that capture various beliefs into the security pattern template. Especially interesting would be a way of expressing and analyzing properties containing belief logic. Finally, investigating the use of timing information could increase the expressiveness of our security properties.

7 Acknowledgements

I wish to thank my supervisor at Michigan State University Dr. Betty Cheng for the excellent advice, her encouragement, and outstanding editing skills that significantly contributed to the quality of this paper. The author is also very grateful to Sascha Konrad and other members of the Software Engineering and Network Systems Laboratory at Michigan State University for helpful comments on this work. Furthermore, I would like to thank Dr. Reinhard Schwarz for agreeing to supervise this work in Germany despite the inconvenience caused by the geographic distance.

References

[1] Hamid Alavi, George Avrunin, James Corbett, Laura Dillon, Matt Dwyer, and Corina Pasareanu. A specification pattern system, 2002.

[2] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, builings, construction*. Oxford University Press, New York, 1977.

[3] Brad Appleton. Patterns and software: Essential concepts and terminology, February 2000.

[4] 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.

[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[6] A. Braga, C. Rubira, and R. Dahab. Tropyc: A pattern language for cryptographic software, 1998.

[7] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. 1960 Int. Congr. for Logic, Methodology, and Philosophy of Science*, pages 1–1. Stanford Univ. Press, 1962.

[8] Laura A. Campbell, Betty H. C. Cheng, William E. McUmber, and R. E. K. Stirewalt. Automatically detecting and visualizing errors in UML diagrams. *Requirements Engineering Journal*, 7(4):264–287, 2002.

[9] Eduardo Fernandez Dept. Metadata and authorization patterns, 2000.

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. Technical Report UM-CS-1998-035, , 1998.

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

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

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

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

[15] Honeywell. http://www.htc.honeywell.com/dome.

[16] Darell M. Kienzle, Matthew C. Elder, David S. Tyree, and James Edwards-Hewitt. Security patterns template and tutorial, June 2002.

[17] Sascha Konrad, Laura A. Campbell, and Betty H. C. Cheng. Adding formal specifications to requirements patterns. In *Proceedings of the Requirements for High Assurance Systems Workshop (RHAS02) as part of the IEEE Joint International Conference on Requirements Engineering (RE02)*, Essen, Germany, September 2002.

[18] Sascha Konrad, Laura A. Campbell, and Betty H.C. Cheng. Requirements patterns for embedded systems. In *International Workshop on Requirements for High Assurance Systems*, September 2002.

[19] Sascha Konrad, Laura A. Campbell, Betty H.C. Cheng, and Min Deng. A requirements patterns-driven approach to check systems and specify properties. In Thomas Ball and Sirom K. Rajamani, editors, *Model Checking Software*, number 2648 in Lecture Notes in Computer Science, pages 18–33. Springer Verlag, May 2003.

[20] Sascha Konrad and Betty H. C. Cheng. Requirements patterns for embedded systems. In *Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE02)*, Essen, Germany, September 2002.

[21] Sascha Konrad, Betty H. C. Cheng, Laura A. Campbell, and Ronald Wassermann. Using security patterns to model and analyze security requirements. In *Proceedings of the Requirements for High Assurance Systems Workshop (RHAS03) as part of the IEEE Joint International Conference on Requirements Engineering (RE03)*, Monterey Bay, CA, USA, September 2003.

[22] William E. McUmber and Betty 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.

[23] William Eugene McUmber. *A generic framework for formalizing object-oriented modeling notations for embedded systems development.* PhD thesis, Michigan State University, August 2000.

[24] Object Management Group (OMG). Unified modeling language specification. Technical report, OMG, March 2003.

[25] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design.* Prentice-Hall, Inc., 1991.

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

[27] Markus Schumacher and Utz Roedig. Security engineering with patterns. In *8th Conference on Pattern Languages of Programs*, July 2001.

[28] Rita C. Summers. *Secure Computing: Threads and Safeguards.* McGraw-Hill, 1997.

[29] AG Communication Systems. Pattern template, March 1996.

[30] National Security Telecommunications and Information Systems Security Committee. National information systems security (infosec) glossary. Technical report, National Security Telecommunications and Information Systems Security Committee, September 2000.

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

[32] www.searchSecurity.com. Social engineering, April 2001.

[33] www.security patterns.org. Security pattern forum.

[34] J. Yoder and J. Barcalow. Architectural patterns for enabling application security, 1997.

A Promela specification

A.1 System 01

```
#define min(x,y) (x<y->x:y)
#define max(x,y) (x>y->x:y)
chan evq=[10] of {mtype,int};
chan evt=[10] of {mtype,int};
chan wait=[10] of {int,mtype};
mtype={
        start, requestAccess,
        setRecipient, setAccess,
        ready, checkMsg,
        logRequest, return,
        isAuthorized, authOK,
        access, trigger
};
typedef SingleAccessPoint_T {
        byte sender;
        byte recipient;
        byte accessType;
        }
SingleAccessPoint_T SingleAccessPoint_V;
chan SingleAccessPoint_q=[5] of {mtype};
chan SingleAccessPoint_C=[0] of {bit};
chan SingleAccessPoint_requestAccess_p1=[5] of {byte};
chan SingleAccessPoint_setRecipient_p1=[5] of {byte};
chan SingleAccessPoint_setAccess_p1=[5] of {byte};
chan _SYSTEMCLASS__q=[5] of {mtype};
chan _SYSTEMCLASS__C=[0] of {bit};
typedef ExternalRequestGenerator_T {
        bool success;
        bool waiting;
        byte sender;
        byte recipient;
        byte accessType;
        }
ExternalRequestGenerator_T ExternalRequestGenerator_V;
chan ExternalRequestGenerator_q=[5] of {mtype};
chan ExternalRequestGenerator_C=[0] of {bit};
chan ExternalRequestGenerator_return_p1=[5] of {bool};
typedef Authorization_T {
        byte sender;
        byte recipient;
        byte accessType;
        }
Authorization_T Authorization_V;
chan Authorization_q=[5] of {mtype};
chan Authorization_C=[0] of {bit};
chan Authorization_isAuthorized_p1=[5] of {byte};
chan Authorization_setRecipient_p1=[5] of {byte};
chan Authorization_setAccess_p1=[5] of {byte};
chan InternalEntityStub_q=[5] of {mtype};
chan InternalEntityStub_C=[0] of {bit};
```

```
typedef CheckPoint_T {
        byte sender;
        byte recipient;
        byte grantAccess;
        byte accessType;
        }
CheckPoint_T CheckPoint_V;
chan CheckPoint_q=[5] of {mtype};
chan CheckPoint_C=[0] of {bit};
chan CheckPoint_checkMsg_p1=[5] of {byte};
chan CheckPoint_setRecipient_p1=[5] of {byte};
chan CheckPoint_authOK_p1=[5] of {byte};
chan CheckPoint_setAccess_p1=[5] of {byte};
typedef CounterMeasure_T {
        bool triggered;
        }
CounterMeasure_T CounterMeasure_V;
chan CounterMeasure_q=[5] of {mtype};
chan CounterMeasure_C=[0] of {bit};
typedef Log_T {
        bool logged;
        }
Log_T Log_V;
chan Log_q=[5] of {mtype};
chan Log_C=[0] of {bit};


proctype SingleAccessPoint()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Idle; skip;};
/* State Idle */
atomic{skip;
Idle:     printf("in state SingleAccessPoint.Idle\n");
/* entry actions */
        SingleAccessPoint_V.sender=0;
        SingleAccessPoint_V.recipient=0;
        SingleAccessPoint_V.accessType=0;
        }
        if
        :: atomic {SingleAccessPoint_q?requestAccess ->
SingleAccessPoint_requestAccess_p1?SingleAccessPoint_V.sender
                   ->
           goto GetParam1; skip;}
        fi;
/* State GetParam1 */
atomic{skip;
GetParam1:     printf("in state SingleAccessPoint.GetParam1\n");
        }
        if
        :: atomic {SingleAccessPoint_q?setRecipient ->
SingleAccessPoint_setRecipient_p1?SingleAccessPoint_V.recipient
```

```
                   ->
           goto GetParam2; skip;}
        fi;
/* State Forward */
atomic{skip;
Forward:     printf("in state SingleAccessPoint.Forward\n");
        }
        if
        :: atomic{1 ->
CheckPoint_checkMsg_p1!SingleAccessPoint_V.sender;
CheckPoint_q!checkMsg;
CheckPoint_setRecipient_p1!SingleAccessPoint_V.recipient;
CheckPoint_q!setRecipient;
CheckPoint_setAccess_p1!SingleAccessPoint_V.accessType;
CheckPoint_q!setAccess;
           goto Idle; skip;}
        fi;
/* State GetParam2 */
atomic{skip;
GetParam2:     printf("in state SingleAccessPoint.GetParam2\n");
        }
        if
        :: atomic {SingleAccessPoint_q?setAccess ->
SingleAccessPoint_setAccess_p1?SingleAccessPoint_V.accessType
                   ->
           Log_q!logRequest;
           goto Forward; skip;}
        fi;
exit:   skip
}


active proctype _SYSTEMCLASS_()
{atomic{
mtype m;
bit dummy;
/*Init state*/
        goto s1; skip;};
/* State s1 */
atomic{skip;
s1:     printf("in state _SYSTEMCLASS_.s1\n");
        }
        if
        :: atomic{1 ->
           run ExternalRequestGenerator();
           goto s2; skip;}
        fi;
/* State s2 */
atomic{skip;
s2:     printf("in state _SYSTEMCLASS_.s2\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run SingleAccessPoint();
           goto s3; skip;}
        fi;
```

```
/* State s3 */
atomic{skip;
s3:     printf("in state _SYSTEMCLASS_.s3\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run CheckPoint();
           goto s4; skip;}
        fi;
/* State s4 */
atomic{skip;
s4:     printf("in state _SYSTEMCLASS_.s4\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run Authorization();
           goto s5; skip;}
        fi;
/* State s5 */
atomic{skip;
s5:     printf("in state _SYSTEMCLASS_.s5\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run InternalEntityStub();
           goto s6; skip;}
        fi;
/* State s6 */
atomic{skip;
s6:     printf("in state _SYSTEMCLASS_.s6\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run CounterMeasure();
           goto s7; skip;}
        fi;
/* State s7 */
atomic{skip;
s7:     printf("in state _SYSTEMCLASS_.s7\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run Log();
           goto s8; skip;}
        fi;
/* State s8 */
atomic{skip;
s8:     printf("in state _SYSTEMCLASS_.s8\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           ExternalRequestGenerator_q!start;
           goto s9; skip;}
        fi;
/* State s9 */
atomic{skip;
```

```
s9:     printf("in state _SYSTEMCLASS_.s9\n");
        }
        if
        :: skip -> false
        fi;
exit:   skip
}


proctype ExternalRequestGenerator()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Wait; skip;};
/* State Wait */
atomic{skip;
Wait:     printf("in state ExternalRequestGenerator.Wait\n");
        }
        if
        :: atomic{ExternalRequestGenerator_q?start ->
           _SYSTEMCLASS__q!ready;
           goto genSender; skip;}
        fi;
/* State genSender */
atomic{skip;
genSender:     printf("in state ExternalRequestGenerator.genSender\n");
        }
        if
        :: atomic{1 ->
           ExternalRequestGenerator_V.sender=1;
           goto genRecipient; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.sender=2;
           goto genRecipient; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.sender=3;
           goto genRecipient; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.sender=4;
           goto genRecipient; skip;}
        fi;
/* State WaitResponse */
atomic{skip;
WaitResponse:
printf("in state ExternalRequestGenerator.WaitResponse\n");
/* entry actions */
        ExternalRequestGenerator_V.success=0;
        ExternalRequestGenerator_V.waiting=1;
        }
        if
        :: atomic {ExternalRequestGenerator_q?return ->
ExternalRequestGenerator_return_p1?ExternalRequestGenerator_V.success
                   ->
           goto EvaluateSuccess; skip;}
        fi;
/* State EvaluateSuccess */
atomic{skip;
EvaluateSuccess:
printf("in state ExternalRequestGenerator.EvaluateSuccess\n");
        }
        if
        :: atomic{1 ->
           ExternalRequestGenerator_V.waiting=0;
           goto genSender; skip;}
        fi;
/* State sendReq */
atomic{skip;
sendReq:     printf("in state ExternalRequestGenerator.sendReq\n");
        }
        if
        :: atomic{1 ->
SingleAccessPoint_requestAccess_p1!ExternalRequestGenerator_V.sender;
SingleAccessPoint_q!requestAccess;
SingleAccessPoint_setRecipient_p1!ExternalRequestGenerator_V.recipient;
SingleAccessPoint_q!setRecipient;
SingleAccessPoint_setAccess_p1!ExternalRequestGenerator_V.accessType;
SingleAccessPoint_q!setAccess;
           goto WaitResponse; skip;}
        fi;
/* State genRecipient */
atomic{skip;
genRecipient:
printf("in state ExternalRequestGenerator.genRecipient\n");
        }
        if
        :: atomic{1 ->
           ExternalRequestGenerator_V.recipient=2;
           goto genAccess; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.recipient=1;
           goto genAccess; skip;}
        fi;
/* State genAccess */
atomic{skip;
genAccess:     printf("in state ExternalRequestGenerator.genAccess\n");
        }
        if
        :: atomic{1 ->
           ExternalRequestGenerator_V.accessType=2;
           goto sendReq; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.accessType=1;
           goto sendReq; skip;}
        fi;
exit:   skip
}


proctype Authorization()
```

```
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Idle; skip;};
/* State Idle */
atomic{skip;
Idle:     printf("in state Authorization.Idle\n");
        }
        if
        :: atomic {Authorization_q?isAuthorized ->
                   Authorization_isAuthorized_p1?Authorization_V.sender
                   ->
           goto getParam1; skip;}
        fi;
/* State Evaluate */
atomic{skip;
Evaluate:     printf("in state Authorization.Evaluate\n");
        }
Evaluate_G:
        if
        :: atomic{1 ->
           if
           :: atomic{Authorization_V.sender==1 ->
              goto Sender1; skip;}
           :: atomic{Authorization_V.sender==2 ->
              goto Sender2; skip;}
           :: atomic{Authorization_V.sender==3 ->
              goto Sender3; skip;}
           :: atomic{Authorization_V.sender==4 ->
              goto Sender4; skip;}
           :: else -> goto Evaluate_G; skip;
           fi}
        fi;
/* State getParam1 */
atomic{skip;
getParam1:     printf("in state Authorization.getParam1\n");
        }
        if
        :: atomic {Authorization_q?setRecipient ->
Authorization_setRecipient_p1?Authorization_V.recipient
                   ->
           goto getParam2; skip;}
        fi;
/* State getParam2 */
atomic{skip;
getParam2:     printf("in state Authorization.getParam2\n");
        }
        if
        :: atomic {Authorization_q?setAccess ->
                   Authorization_setAccess_p1?Authorization_V.accessType
                   ->
           goto Evaluate; skip;}
        fi;
```

```
/* State AccessDenied */
atomic{skip;
AccessDenied:     printf("in state Authorization.AccessDenied\n");
        }
        if
        :: atomic{1 ->
           CheckPoint_authOK_p1!2; CheckPoint_q!authOK;
           goto Idle; skip;}
        fi;
/* State AccessGranted */
atomic{skip;
AccessGranted:     printf("in state Authorization.AccessGranted\n");
        }
        if
        :: atomic{1 ->
           CheckPoint_authOK_p1!1; CheckPoint_q!authOK;
           goto Idle; skip;}
        fi;
/* State Sender1 */
atomic{skip;
Sender1:     printf("in state Authorization.Sender1\n");
        }
        if
        :: atomic{1 ->
           goto AccessGranted; skip;}
        fi;
/* State Sender2 */
atomic{skip;
Sender2:     printf("in state Authorization.Sender2\n");
        }
Sender2_G:
        if
        :: atomic{1 ->
           if
           :: atomic{Authorization_V.recipient!=1 ->
              goto AccessDenied; skip;}
           :: atomic{Authorization_V.recipient==1 ->
              goto AccessGranted; skip;}
           :: else -> goto Sender2_G; skip;
           fi}
        fi;
/* State Sender3 */
atomic{skip;
Sender3:     printf("in state Authorization.Sender3\n");
        }
Sender3_G:
        if
        :: atomic{1 ->
           if
:: atomic{Authorization_V.recipient==2 && Authorization_V.accessType==1
->
              goto AccessGranted; skip;}
:: atomic{Authorization_V.recipient!=2 || Authorization_V.accessType!=1
->
              goto AccessDenied; skip;}
           :: else -> goto Sender3_G; skip;
           fi}
        fi;
/* State Sender4 */
atomic{skip;
Sender4:     printf("in state Authorization.Sender4\n");
        }
        if
        :: atomic{1 ->
           goto AccessDenied; skip;}
        fi;
exit:   skip
}


proctype InternalEntityStub()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Idle; skip;};
/* State Idle */
atomic{skip;
Idle:     printf("in state InternalEntityStub.Idle\n");
        }
        if
        :: atomic{InternalEntityStub_q?access ->
ExternalRequestGenerator_return_p1!1; ExternalRequestGenerator_q!return;
           goto Idle; skip;}
        fi;
exit:   skip
}


proctype CheckPoint()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto MainIdle; skip;};
/* State MainIdle */
atomic{skip;
MainIdle:     printf("in state CheckPoint.MainIdle\n");
/* entry actions */
        CheckPoint_V.grantAccess=0;
        }
        if
        :: atomic {CheckPoint_q?checkMsg ->
                   CheckPoint_checkMsg_p1?CheckPoint_V.sender
                   ->
           goto getReq1; skip;}
        fi;
/* State AccessDenied */
```

```
atomic{skip;
AccessDenied:     printf("in state CheckPoint.AccessDenied\n");
        }
        if
        :: atomic{1 ->
           CounterMeasure_q!trigger;
           goto MainIdle; skip;}
        fi;
/* State waitAuth */
atomic{skip;
waitAuth:     printf("in state CheckPoint.waitAuth\n");
        }
        if
        :: atomic {CheckPoint_q?authOK ->
                   CheckPoint_authOK_p1?CheckPoint_V.grantAccess
                   ->
           goto processRequest; skip;}
        fi;
/* State AccessGranted */
atomic{skip;
AccessGranted:     printf("in state CheckPoint.AccessGranted\n");
        }
        if
        :: atomic{1 ->
           InternalEntityStub_q!access;
           goto MainIdle; skip;}
        fi;
/* State processRequest */
atomic{skip;
processRequest:     printf("in state CheckPoint.processRequest\n");
/* entry actions */
        CheckPoint_V.sender=0;
        CheckPoint_V.recipient=0;
        CheckPoint_V.accessType=0;
        }
processRequest_G:
        if
        :: atomic{1 ->
           if
           :: atomic{CheckPoint_V.grantAccess==2 ->
              goto AccessDenied; skip;}
           :: atomic{CheckPoint_V.grantAccess==1 ->
              goto AccessGranted; skip;}
           :: else -> goto processRequest_G; skip;
           fi}
        fi;
/* State getReq1 */
atomic{skip;
getReq1:     printf("in state CheckPoint.getReq1\n");
        }
        if
        :: atomic {CheckPoint_q?setRecipient ->
                   CheckPoint_setRecipient_p1?CheckPoint_V.recipient
                   ->
           goto getReq2; skip;}
        fi;
```

```
/* State getReq2 */
atomic{skip;
getReq2:     printf("in state CheckPoint.getReq2\n");
        }
        if
        :: atomic {CheckPoint_q?setAccess ->
                   CheckPoint_setAccess_p1?CheckPoint_V.accessType
                   ->
Authorization_isAuthorized_p1!CheckPoint_V.sender;
Authorization_q!isAuthorized;
Authorization_setRecipient_p1!CheckPoint_V.recipient;
Authorization_q!setRecipient;
Authorization_setAccess_p1!CheckPoint_V.accessType;
Authorization_q!setAccess;
           goto waitAuth; skip;}
        fi;
exit:   skip
}


proctype CounterMeasure()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Idle; skip;};
/* State Idle */
atomic{skip;
Idle:     printf("in state CounterMeasure.Idle\n");
        }
        if
        :: atomic{CounterMeasure_q?trigger ->
           CounterMeasure_V.triggered=1;
ExternalRequestGenerator_return_p1!0; ExternalRequestGenerator_q!return;
           goto CMTriggered; skip;}
        fi;
/* State CMTriggered */
atomic{skip;
CMTriggered:     printf("in state CounterMeasure.CMTriggered\n");
        }
        if
        :: atomic{1 ->
           CounterMeasure_V.triggered=0;
           goto Idle; skip;}
        fi;
exit:   skip
}


proctype Log()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Idle; skip;};
/* State Idle */
atomic{skip;
Idle:     printf("in state Log.Idle\n");
/* entry actions */
        Log_V.logged=0;
        }
        if
        :: atomic{Log_q?logRequest ->
           Log_V.logged=1;
           goto Logging; skip;}
        fi;
/* State Logging */
atomic{skip;
Logging:     printf("in state Log.Logging\n");
        }
        if
        :: atomic{1 ->
           goto Idle; skip;}
        fi;
exit:   skip
}


/* This is the universal event dispatcher routine */
proctype event(mtype msg)
{
   mtype type;
   int process_id;

   atomic {
   do
   :: evq??[eval(msg),process_id] ->
      evq??eval(msg),process_id;
      evt!msg,process_id;
      do
      :: if
         :: evq??[type,eval(process_id)] -> evq??type,eval(process_id)
         :: else break;
         fi
      od
   :: else -> break
   od}
exit: skip
}
```

A.2 System 02

```
#define min(x,y) (x<y->x:y)
#define max(x,y) (x>y->x:y)
chan evq=[10] of {mtype,int};
chan evt=[10] of {mtype,int};
chan wait=[10] of {int,mtype};
mtype={
        login, logout,
        access, sessionOK,
        initS, ready,
        loginOK, destroyS,
        checkMsg, logoutOK,
        start, return,
        subjClass, objClass,
        requestClassification, setRole,
        getInfo, getRole,
        subjectInfo
};
typedef SingleAccessPoint_T {
        byte user;
        byte recipient;
        }
SingleAccessPoint_T SingleAccessPoint_V;
chan SingleAccessPoint_q=[5] of {mtype};
chan SingleAccessPoint_C=[0] of {bit};
chan SingleAccessPoint_login_p1=[5] of {byte};
chan SingleAccessPoint_access_p1=[5] of {byte};
chan _SYSTEMCLASS__q=[5] of {mtype};
chan _SYSTEMCLASS__C=[0] of {bit};
typedef ExternalRequestGenerator_T {
        byte sender;
        int request;
        bool success;
        }
ExternalRequestGenerator_T ExternalRequestGenerator_V;
chan ExternalRequestGenerator_q=[5] of {mtype};
chan ExternalRequestGenerator_C=[0] of {bit};
chan ExternalRequestGenerator_return_p1=[5] of {bool};
typedef CheckPoint_T {
        byte recipient;
        byte c1;
        byte c2;
        byte sC;
        byte oC;
        }
CheckPoint_T CheckPoint_V;
chan CheckPoint_q=[5] of {mtype};
chan CheckPoint_C=[0] of {bit};
chan CheckPoint_checkMsg_p1=[5] of {byte};
chan CheckPoint_subjClass_p1=[5] of {byte};
chan CheckPoint_objClass_p1=[5] of {byte};
chan InternalObjects_q=[5] of {mtype};
chan InternalObjects_C=[0] of {bit};
typedef Session_T {
        byte user;
        byte role;
        bool sRole;
        }
Session_T Session_V;
chan Session_q=[5] of {mtype};
chan Session_C=[0] of {bit};
chan Session_initS_p1=[5] of {byte};
chan Session_setRole_p1=[5] of {byte};
typedef Roles_T {
        byte user;
        byte role;
        }
Roles_T Roles_V;
chan Roles_q=[5] of {mtype};
chan Roles_C=[0] of {bit};
chan Roles_getRole_p1=[5] of {byte};
typedef SubjectClassification_T {
        byte subject;
        }
SubjectClassification_T SubjectClassification_V;
chan SubjectClassification_q=[5] of {mtype};
chan SubjectClassification_C=[0] of {bit};
chan SubjectClassification_subjectInfo_p1=[5] of {byte};
typedef ObjectClassification_T {
        byte object;
        }
ObjectClassification_T ObjectClassification_V;
chan ObjectClassification_q=[5] of {mtype};
chan ObjectClassification_C=[0] of {bit};
chan ObjectClassification_requestClassification_p1=[5] of {byte};


proctype SingleAccessPoint()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto LoggedOut; skip;};
/* State LoggedOut */
atomic{skip;
LoggedOut:     printf("in state SingleAccessPoint.LoggedOut\n");
/* entry actions */
        SingleAccessPoint_V.recipient=0;
        SingleAccessPoint_V.user=0;
        }
        if
        :: atomic {SingleAccessPoint_q?login ->
                   SingleAccessPoint_login_p1?SingleAccessPoint_V.user
                   ->
           Session_initS_p1!SingleAccessPoint_V.user; Session_q!initS;
           goto waitSession; skip;}
        fi;
/* State waitSession */
atomic{skip;
waitSession:     printf("in state SingleAccessPoint.waitSession\n");
        }
        if
        :: atomic{SingleAccessPoint_q?sessionOK ->
           ExternalRequestGenerator_q!loginOK;
           goto LoggedIn; skip;}
        fi;
/* State LoggedIn */
atomic{skip;
LoggedIn:     printf("in state SingleAccessPoint.LoggedIn\n");
        }
        if
        :: atomic{SingleAccessPoint_q?logout ->
           Session_q!destroyS;
           goto waitSessionDestroy; skip;}
        :: atomic {SingleAccessPoint_q?access ->
SingleAccessPoint_access_p1?SingleAccessPoint_V.recipient
                   ->
CheckPoint_checkMsg_p1!SingleAccessPoint_V.recipient;
CheckPoint_q!checkMsg;
           goto LoggedIn; skip;}
        fi;
/* State waitSessionDestroy */
atomic{skip;
waitSessionDestroy:
printf("in state SingleAccessPoint.waitSessionDestroy\n");
        }
        if
        :: atomic{SingleAccessPoint_q?sessionOK ->
           ExternalRequestGenerator_q!logoutOK;
           goto LoggedOut; skip;}
        fi;
exit:   skip
}


active proctype _SYSTEMCLASS_()
{atomic{
mtype m;
bit dummy;
/*Init state*/
        goto s1; skip;};
/* State s1 */
atomic{skip;
s1:     printf("in state _SYSTEMCLASS_.s1\n");
        }
        if
        :: atomic{1 ->
           run ExternalRequestGenerator();
           goto s2; skip;}
        fi;
/* State s2 */
atomic{skip;
s2:     printf("in state _SYSTEMCLASS_.s2\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run SingleAccessPoint();
           goto s3; skip;}
        fi;
/* State s3 */
atomic{skip;
s3:     printf("in state _SYSTEMCLASS_.s3\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run CheckPoint();
           goto s4; skip;}
        fi;
/* State s4 */
atomic{skip;
s4:     printf("in state _SYSTEMCLASS_.s4\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run Session();
           goto s5; skip;}
        fi;
/* State s5 */
atomic{skip;
s5:     printf("in state _SYSTEMCLASS_.s5\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run Roles();
           goto s6; skip;}
        fi;
/* State s6 */
atomic{skip;
s6:     printf("in state _SYSTEMCLASS_.s6\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run InternalObjects();
           goto s7; skip;}
        fi;
/* State s7 */
atomic{skip;
s7:     printf("in state _SYSTEMCLASS_.s7\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run SubjectClassification();
           goto s8; skip;}
        fi;
/* State s8 */
atomic{skip;
s8:     printf("in state _SYSTEMCLASS_.s8\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           run ObjectClassification();
           goto s9; skip;}
        fi;
/* State s9 */
atomic{skip;
s9:     printf("in state _SYSTEMCLASS_.s9\n");
        }
        if
        :: atomic{_SYSTEMCLASS__q?ready ->
           ExternalRequestGenerator_q!start;
           goto s10; skip;}
        fi;
/* State s10 */
atomic{skip;
s10:     printf("in state _SYSTEMCLASS_.s10\n");
        }
        if
        :: skip -> false
        fi;
exit:   skip
}


proctype ExternalRequestGenerator()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Wait; skip;};
/* State Wait */
atomic{skip;
Wait:     printf("in state ExternalRequestGenerator.Wait\n");
        }
        if
        :: atomic{ExternalRequestGenerator_q?start ->
           _SYSTEMCLASS__q!ready;
           goto LoggedOut; skip;}
        fi;
/* State LoggedOut */
atomic{skip;
LoggedOut:     printf("in state ExternalRequestGenerator.LoggedOut\n");
/* entry actions */
        ExternalRequestGenerator_V.sender=0;
        }
        if
        :: atomic{1 ->
           ExternalRequestGenerator_V.sender=1;
           goto genSender; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.sender=2;
           goto genSender; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.sender=3;
           goto genSender; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.sender=4;
           goto genSender; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.sender=5;
           goto genSender; skip;}
        fi;
```

```
/* State WaitResponse */
atomic{skip;
WaitResponse:
printf("in state ExternalRequestGenerator.WaitResponse\n");
        }
        if
        :: atomic {ExternalRequestGenerator_q?return ->
ExternalRequestGenerator_return_p1?ExternalRequestGenerator_V.success
                   ->
           goto EvaluateSuccess; skip;}
        fi;
/* State EvaluateSuccess */
atomic{skip;
EvaluateSuccess:
printf("in state ExternalRequestGenerator.EvaluateSuccess\n");
        }
        if
        :: atomic{1 ->
           goto LoggedIn; skip;}
        fi;
/* State waitLogin */
atomic{skip;
waitLogin:     printf("in state ExternalRequestGenerator.waitLogin\n");
        }
        if
        :: atomic{ExternalRequestGenerator_q?loginOK ->
           goto LoggedIn; skip;}
        fi;
/* State genSender */
atomic{skip;
genSender:     printf("in state ExternalRequestGenerator.genSender\n");
        }
        if
        :: atomic{1 ->
SingleAccessPoint_login_p1!ExternalRequestGenerator_V.sender;
SingleAccessPoint_q!login;
           goto waitLogin; skip;}
        fi;
/* State LoggedIn */
atomic{skip;
LoggedIn:     printf("in state ExternalRequestGenerator.LoggedIn\n");
/* entry actions */
        ExternalRequestGenerator_V.request=0;
        ExternalRequestGenerator_V.success=0;
        }
        if
        :: atomic{1 ->
           SingleAccessPoint_q!logout;
           goto waitLogout; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.request=1;
           SingleAccessPoint_access_p1!1; SingleAccessPoint_q!access;
           goto WaitResponse; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.request=2;
           SingleAccessPoint_access_p1!2; SingleAccessPoint_q!access;
           goto WaitResponse; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.request=3;
           SingleAccessPoint_access_p1!3; SingleAccessPoint_q!access;
           goto WaitResponse; skip;}
        :: atomic{1 ->
           ExternalRequestGenerator_V.request=4;
           SingleAccessPoint_access_p1!4; SingleAccessPoint_q!access;
           goto WaitResponse; skip;}
        fi;
/* State waitLogout */
atomic{skip;
waitLogout:
printf("in state ExternalRequestGenerator.waitLogout\n");
        }
        if
        :: atomic{ExternalRequestGenerator_q?logoutOK ->
           goto LoggedOut; skip;}
        fi;
exit:   skip
}


proctype CheckPoint()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto MainIdle; skip;};
/* State MainIdle */
atomic{skip;
MainIdle:     printf("in state CheckPoint.MainIdle\n");
/* entry actions */
        CheckPoint_V.sC=0;
        CheckPoint_V.oC=0;
        CheckPoint_V.recipient=0;
        }
        if
        :: atomic {CheckPoint_q?checkMsg ->
                   CheckPoint_checkMsg_p1?CheckPoint_V.recipient
                   ->
           SubjectClassification_q!requestClassification;
           goto GetSubjectClassification; skip;}
        fi;
/* State GetSubjectClassification */
atomic{skip;
GetSubjectClassification:
printf("in state CheckPoint.GetSubjectClassification\n");
        }
        if
        :: atomic {CheckPoint_q?subjClass ->
                   CheckPoint_subjClass_p1?CheckPoint_V.sC
                   ->
ObjectClassification_requestClassification_p1!CheckPoint_V.recipient;
```

```
ObjectClassification_q!requestClassification;
           goto GetObjectClassification; skip;}
        fi;
/* State GetObjectClassification */
atomic{skip;
GetObjectClassification:
printf("in state CheckPoint.GetObjectClassification\n");
        }
        if
        :: atomic {CheckPoint_q?objClass ->
                   CheckPoint_objClass_p1?CheckPoint_V.oC
                   ->
           goto extractSubCLevel; skip;}
        fi;
/* State SubjectDominates */
atomic{skip;
SubjectDominates:     printf("in state CheckPoint.SubjectDominates\n");
/* entry actions */
        CheckPoint_V.c1=0;
        CheckPoint_V.c2=0;
        }
        if
        :: atomic{1 ->
           InternalObjects_q!access;
           goto MainIdle; skip;}
        fi;
/* State SubjectDominatesNot */
atomic{skip;
SubjectDominatesNot:
printf("in state CheckPoint.SubjectDominatesNot\n");
/* entry actions */
        CheckPoint_V.c1=0;
        CheckPoint_V.c2=0;
        }
        if
        :: atomic{1 ->
ExternalRequestGenerator_return_p1!0; ExternalRequestGenerator_q!return;
           goto MainIdle; skip;}
        fi;
/* State extractSubCLevel */
atomic{skip;
extractSubCLevel:     printf("in state CheckPoint.extractSubCLevel\n");
        }
extractSubCLevel_G:
        if
        :: atomic{1 ->
           if
           :: atomic{CheckPoint_V.sC==10 ->
              CheckPoint_V.c1=2;
              goto extractObjCLevel; skip;}
           :: atomic{CheckPoint_V.sC==8 ->
              CheckPoint_V.c1=2;
              goto extractObjCLevel; skip;}
           :: atomic{CheckPoint_V.sC==5 ->
              CheckPoint_V.c1=1;
              goto extractObjCLevel; skip;}
           :: atomic{CheckPoint_V.sC==2 ->
              CheckPoint_V.c1=0;
              goto extractObjCLevel; skip;}
           :: atomic{CheckPoint_V.sC==1 ->
              CheckPoint_V.c1=0;
              goto extractObjCLevel; skip;}
           :: else -> goto extractSubCLevel_G; skip;
           fi}
        fi;
/* State extractObjCLevel */
atomic{skip;
extractObjCLevel:     printf("in state CheckPoint.extractObjCLevel\n");
        }
extractObjCLevel_G:
        if
        :: atomic{1 ->
           if
           :: atomic{CheckPoint_V.oC==8 ->
              CheckPoint_V.c2=2;
              goto CheckCompany; skip;}
           :: atomic{CheckPoint_V.oC==5 ->
              CheckPoint_V.c2=1;
              goto CheckCompany; skip;}
           :: atomic{CheckPoint_V.oC==1 ->
              CheckPoint_V.c2=0;
              goto CheckCompany; skip;}
           :: atomic{CheckPoint_V.oC==2 ->
              CheckPoint_V.c2=0;
              goto CheckCompany; skip;}
           :: else -> goto extractObjCLevel_G; skip;
           fi}
        fi;
/* State CheckCompany */
atomic{skip;
CheckCompany:     printf("in state CheckPoint.CheckCompany\n");
        }
CheckCompany_G:
        if
        :: atomic{1 ->
           if
           :: atomic{CheckPoint_V.c1<CheckPoint_V.c2 ->
              goto SubjectDominatesNot; skip;}
           :: atomic{CheckPoint_V.c1>=CheckPoint_V.c2 ->
              goto extractSubULevel; skip;}
           :: else -> goto CheckCompany_G; skip;
           fi}
        fi;
/* State extractSubULevel */
atomic{skip;
extractSubULevel:     printf("in state CheckPoint.extractSubULevel\n");
        }
extractSubULevel_G:
        if
        :: atomic{1 ->
           if
           :: atomic{CheckPoint_V.sC==10 ->
```

```
              CheckPoint_V.c1=2;
              goto extractObjULevel; skip;}
           :: atomic{CheckPoint_V.sC==8 ->
              CheckPoint_V.c1=0;
              goto extractObjULevel; skip;}
           :: atomic{CheckPoint_V.sC==5 ->
              CheckPoint_V.c1=1;
              goto extractObjULevel; skip;}
           :: atomic{CheckPoint_V.sC==2 ->
              CheckPoint_V.c1=2;
              goto extractObjULevel; skip;}
           :: atomic{CheckPoint_V.sC==1 ->
              CheckPoint_V.c1=1;
              goto extractObjULevel; skip;}
           :: else -> goto extractSubULevel_G; skip;
           fi}
        fi;
/* State extractObjULevel */
atomic{skip;
extractObjULevel:     printf("in state CheckPoint.extractObjULevel\n");
        }
extractObjULevel_G:
        if
        :: atomic{1 ->
           if
           :: atomic{CheckPoint_V.oC==5 ->
              CheckPoint_V.c2=1;
              goto checkUniversity; skip;}
           :: atomic{CheckPoint_V.oC==8 ->
              CheckPoint_V.c2=0;
              goto checkUniversity; skip;}
           :: atomic{CheckPoint_V.oC==2 ->
              CheckPoint_V.c2=2;
              goto checkUniversity; skip;}
           :: atomic{CheckPoint_V.oC==1 ->
              CheckPoint_V.c2=1;
              goto checkUniversity; skip;}
           :: else -> goto extractObjULevel_G; skip;
           fi}
        fi;
/* State checkUniversity */
atomic{skip;
checkUniversity:     printf("in state CheckPoint.checkUniversity\n");
        }
checkUniversity_G:
        if
        :: atomic{1 ->
           if
           :: atomic{CheckPoint_V.c1<CheckPoint_V.c2 ->
              goto SubjectDominatesNot; skip;}
           :: atomic{CheckPoint_V.c1>=CheckPoint_V.c2 ->
              goto SubjectDominates; skip;}
           :: else -> goto checkUniversity_G; skip;
           fi}
        fi;
exit:   skip
```

```
}


proctype InternalObjects()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Idle; skip;};
/* State Idle */
atomic{skip;
Idle:     printf("in state InternalObjects.Idle\n");
        }
        if
        :: atomic{InternalObjects_q?access ->
ExternalRequestGenerator_return_p1!1; ExternalRequestGenerator_q!return;
           goto Idle; skip;}
        fi;
exit:   skip
}


proctype Session()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Init; skip;};
/* State Init */
atomic{skip;
Init:     printf("in state Session.Init\n");
/* entry actions */
        Session_V.user=0;
        Session_V.role=0;
        }
        if
        :: atomic {Session_q?initS ->
                   Session_initS_p1?Session_V.user
                   ->
           Roles_getRole_p1!Session_V.user; Roles_q!getRole;
           goto WaitRole; skip;}
        fi;
/* State Idle */
atomic{skip;
Idle:     printf("in state Session.Idle\n");
/* entry actions */
        Session_V.sRole=0;
        }
        if
        :: atomic{Session_q?destroyS ->
           SingleAccessPoint_q!sessionOK;
           goto Init; skip;}
        :: atomic{Session_q?getInfo ->
SubjectClassification_subjectInfo_p1!Session_V.role;
SubjectClassification_q!subjectInfo;
           goto Idle; skip;}
        fi;
/* State WaitRole */
atomic{skip;
WaitRole:     printf("in state Session.WaitRole\n");
/* entry actions */
        Session_V.sRole=1;
        }
        if
        :: atomic {Session_q?setRole ->
                   Session_setRole_p1?Session_V.role
                   ->
           SingleAccessPoint_q!sessionOK;
           goto Idle; skip;}
        fi;
exit:   skip
}


proctype Roles()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Idle; skip;};
/* State Idle */
atomic{skip;
Idle:     printf("in state Roles.Idle\n");
/* entry actions */
        Roles_V.user=0;
        Roles_V.role=0;
        }
        if
        :: atomic {Roles_q?getRole ->
                   Roles_getRole_p1?Roles_V.user
                   ->
           goto getUser; skip;}
        fi;
/* State getUser */
atomic{skip;
getUser:     printf("in state Roles.getUser\n");
        }
getUser_G:
        if
        :: atomic{1 ->
           if
           :: atomic{Roles_V.user==1 ->
              goto SysAdmin; skip;}
           :: atomic{Roles_V.user==3 ->
              goto Project; skip;}
           :: atomic{Roles_V.user==4 ->
```

```
              goto Faculty; skip;}
           :: atomic{Roles_V.user==2 ->
              goto Company; skip;}
           :: atomic{Roles_V.user==5 ->
              goto Student; skip;}
           :: else -> goto getUser_G; skip;
           fi}
        fi;
/* State SysAdmin */
atomic{skip;
SysAdmin:     printf("in state Roles.SysAdmin\n");
        }
        if
        :: atomic{1 ->
           Roles_V.role=1;
           goto determineRole; skip;}
        fi;
/* State Project */
atomic{skip;
Project:     printf("in state Roles.Project\n");
        }
        if
        :: atomic{1 ->
           Roles_V.role=3;
           goto determineRole; skip;}
        fi;
/* State Faculty */
atomic{skip;
Faculty:     printf("in state Roles.Faculty\n");
        }
        if
        :: atomic{1 ->
           Roles_V.role=4;
           goto determineRole; skip;}
        fi;
/* State Company */
atomic{skip;
Company:     printf("in state Roles.Company\n");
        }
        if
        :: atomic{1 ->
           Roles_V.role=2;
           goto determineRole; skip;}
        fi;
/* State Student */
atomic{skip;
Student:     printf("in state Roles.Student\n");
        }
        if
        :: atomic{1 ->
           Roles_V.role=5;
           goto determineRole; skip;}
        fi;
/* State determineRole */
atomic{skip;
determineRole:     printf("in state Roles.determineRole\n");
```

```
        }
        if
        :: atomic{1 ->
           Session_setRole_p1!Roles_V.role; Session_q!setRole;
           goto Idle; skip;}
        fi;
exit:   skip
}


proctype SubjectClassification()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Idle; skip;};
/* State Idle */
atomic{skip;
Idle:    printf("in state SubjectClassification.Idle\n");
/* entry actions */
        SubjectClassification_V.subject=0;
        }
        if
        :: atomic{SubjectClassification_q?requestClassification ->
           Session_q!getInfo;
           goto GetInfo; skip;}
        fi;
/* State GetInfo */
atomic{skip;
GetInfo:    printf("in state SubjectClassification.GetInfo\n");
        }
        if
        :: atomic {SubjectClassification_q?subjectInfo ->
SubjectClassification_subjectInfo_p1?SubjectClassification_V.subject
                   ->
           goto Evaluate; skip;}
        fi;
/* State Evaluate */
atomic{skip;
Evaluate:    printf("in state SubjectClassification.Evaluate\n");
        }
Evaluate_G:
        if
        :: atomic{1 ->
           if
           :: atomic{SubjectClassification_V.subject==3 ->
              CheckPoint_subjClass_p1!5; CheckPoint_q!subjClass;
              goto Idle; skip;}
           :: atomic{SubjectClassification_V.subject==4 ->
              CheckPoint_subjClass_p1!2; CheckPoint_q!subjClass;
              goto Idle; skip;}
           :: atomic{SubjectClassification_V.subject==1 ->
              CheckPoint_subjClass_p1!10; CheckPoint_q!subjClass;
              goto Idle; skip;}
           :: atomic{SubjectClassification_V.subject==2 ->
              CheckPoint_subjClass_p1!8; CheckPoint_q!subjClass;
              goto Idle; skip;}
           :: atomic{SubjectClassification_V.subject==5 ->
              CheckPoint_subjClass_p1!1; CheckPoint_q!subjClass;
              goto Idle; skip;}
           :: else -> goto Evaluate_G; skip;
           fi}
        fi;
exit:   skip
}


proctype ObjectClassification()
{atomic{
mtype m;
bit dummy;
/*Init state*/
/*Initial actions / messages */
        _SYSTEMCLASS__q!ready;
        goto Idle; skip;};
/* State Idle */
atomic{skip;
Idle:     printf("in state ObjectClassification.Idle\n");
/* entry actions */
        ObjectClassification_V.object=0;
        }
        if
        :: atomic {ObjectClassification_q?requestClassification ->
ObjectClassification_requestClassification_p1?ObjectClassification_V.object
                   ->
           goto Evaluate; skip;}
        fi;
/* State Evaluate */
atomic{skip;
Evaluate:     printf("in state ObjectClassification.Evaluate\n");
        }
Evaluate_G:
        if
        :: atomic{1 ->
           if
           :: atomic{ObjectClassification_V.object==2 ->
              CheckPoint_objClass_p1!5; CheckPoint_q!objClass;
              goto Idle; skip;}
           :: atomic{ObjectClassification_V.object==1 ->
              CheckPoint_objClass_p1!8; CheckPoint_q!objClass;
              goto Idle; skip;}
           :: atomic{ObjectClassification_V.object==3 ->
              CheckPoint_objClass_p1!1; CheckPoint_q!objClass;
              goto Idle; skip;}
           :: atomic{ObjectClassification_V.object==4 ->
              CheckPoint_objClass_p1!2; CheckPoint_q!objClass;
              goto Idle; skip;}
           :: else -> goto Evaluate_G; skip;
           fi}
        fi;
```

```
exit:   skip
}


/* This is the universal event dispatcher routine */
proctype event(mtype msg)
{
   mtype type;
   int process_id;

   atomic {
   do
   :: evq??[eval(msg),process_id] ->
      evq??eval(msg),process_id;
      evt!msg,process_id;
      do
      :: if
         :: evq??[type,eval(process_id)] -> evq??type,eval(process_id)
         :: else break;
         fi
      od
   :: else -> break
   od}
exit: skip
}
```