Systematic Development of UMLsec Design Models Based On Security Requirements

Denis Hatebur[1,4] and Maritta Heisel[1] and Jan Jürjens[2,3] and Holger Schmidt[2]

[1] Software Engineering, Department of Computer Science and Applied Cognitive Science, Faculty of Engineering, University Duisburg-Essen, Germany
[2] Software Engineering, Department of Computer Science, TU Dortmund, Germany
[3] Fraunhofer Institut für Software- und Systemtechnik, Germany
[4] Institut für technische Systeme GmbH, Germany
{denis.hatebur,maritta.heisel}@uni-due.de
{jan.jurjens,holger.schmidt}@cs.tu-dortmund.de

Abstract. Developing security-critical systems in a way that makes sure that the developed systems actually enforce the desired security requirements is difficult, as can be seen by many security vulnerabilities arising in practice on a regular basis. Part of the difficulty is the transition from the security requirements analysis to the design, which is highly non-trivial and error-prone, leaving the risk of introducing vulnerabilities. Unfortunately, existing approaches bridging this gap largely only provide informal guidelines for the transition from security requirements to secure design.
We present a *method* to systematically develop structural and behavioral UMLsec design models based on security requirements. Each step of our method is supported by *model generation rules* expressed as pre- and postconditions using the formal specification language OCL. Moreover, we present a concept for a *CASE tool* based on the model generation rules. Thus, applying our method to generate UMLsec design models supported by this tool and based on previously captured and analyzed security requirements becomes systematic, less error-prone, and a more routine engineering activity.
We illustrate our method by the example of a patient monitoring system.

1 Introduction

When building *secure systems*, it is instrumental to take *security requirements* into account right from the beginning of the development process to reach the best possible match between the expressed requirements and the developed software product, and to eliminate any source of error as early as possible. Knowing that building secure systems is a highly sensitive process, it is important to accomplish the transition from security requirements to secure design *correctly*, i.e., without introducing vulnerabilities.

In fact, there already exist a number of approaches to security requirements analysis (see [3] for an overview) and secure design (e.g., [10, 9]). Although this can be considered a positive development, the different approaches are mostly not integrated with each other. In particular, existing approaches on bridging the gap between security requirements analysis and design only provide informal guidelines for the transition from security requirements to design. Carrying out the transition manually according to these guidelines is highly non-trivial and error-prone, which leaves the risk of inadvertently introducing vulnerabilities. Ultimately, this would lead to the security requirements not being enforced in the system design (and later its implementation).

We present a method to systematically develop structural and behavioral design models based on security requirements. We use a security requirement analysis method [6, 13] inspired by Jackson [8] that uses the UML (Unified Modeling Language)[5] profile *UML4PF* [5] to capture, structure, and analyze security requirements. We extend this approach by a detailed procedure for developing *UMLsec* [9] design models from previously captured and analyzed security requirements. Our method is supported by *model generation rules* expressed as pre- and postconditions using the formal specification language *OCL* (Object Constraint Language)[6]. We present a concept for a *CASE tool* based on the model generation rules. Since our rules are specified in a formal and analyzable way, the implementation of this tool can be checked automatically for correctness with respect to the model generation rules. Consequently, applying our method to generate UMLsec design models supported by our tool and based on previously captured and analyzed security requirements becomes systematic, less error-prone, and a more routine engineering activity. We illustrate our method by the example of a patient monitoring system.

The rest of the paper is organized as follows: Section 2 introduces our security requirements engineering approach. We give a brief introduction into UMLsec in Sect. 3, which we use in Sect. 4 to systematically develop UMLsec design models based on previously captured and analyzed security requirements. We consider related work in Sect. 5. In Sect. 6, we give a summary and directions for future research.

2 Environment Description and Security Requirements Analysis

We propose a requirements engineering approach inspired by Jackson [8]. We illustrate this approach using the example of a *patient monitoring system*, which displays the vital signs of patients to physicians and nurses, and controls an infusion flow according to previously configured rules. In this setting, the display data and the configuration rules are transmitted over an insecure wireless network. We use this case study as a running example throughout this paper.

Security requirements can only be guaranteed for a certain context. Therefore, it is important to describe the *environment*, since software (called *machine*) is built to improve something in its environment. A *context diagram* represents the environment in which the machine will operate. Figure 1 shows the context diagram of the PatientMonitoringSystem (PMS) case study in UML notation with stereotypes defined in the UML profile UML4PF [5]. This profile is available online via `http://swe.uni-due.de/en/research/tool/`. Stereotypes give a specific meaning to the elements of a UML diagram they are attached to, and they are represented by labels surrounded by double angle brackets.

The machine is stereotyped ≪`machine`≫, and in our example in Fig. 1 it is represented by the class PatientMonitoringSystem. A context diagram structures

[5] `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/`
[6] `http://www.omg.org/docs/formal/06-05-01.pdf`

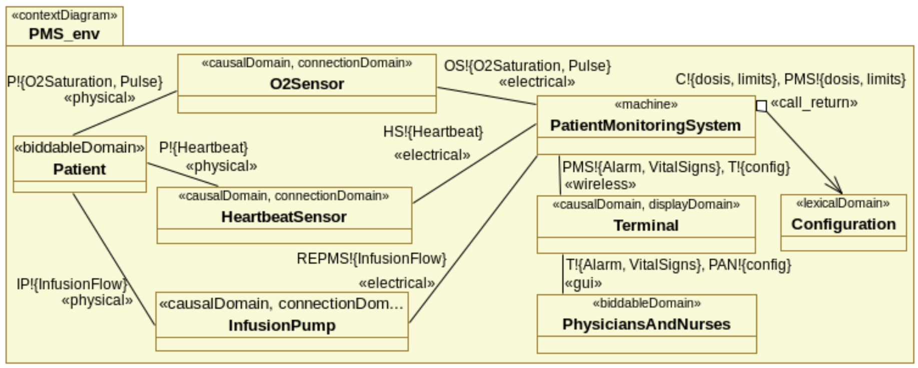

Fig. 1. Context Diagram of Patient Monitoring System

the environment using domains and interfaces. *Domains* describe entities in the environment. Jackson distinguishes the domain types *biddable domains* that are usually people, *causal domains* that comply with some physical laws, and *lexical domains* that are data representations. The domain types are modeled by the stereotypes ≪`BiddableDomain`≫ and ≪`CausalDomain`≫ being subclasses of the stereotype ≪`Domain`≫. A lexical domain (≪`LexicalDomain`≫) is modeled as a special case of a causal domain. To describe the problem context in more detail, *connection domains* may be necessary. Connection domains establish a connection between other domains by means of technical devices. They are modeled as classes with the stereotype ≪`ConnectionDomain`≫. Connection domains are, e.g., video cameras, sensors, or networks. A special type of connection domain is the *display* domain [2] for representing a display providing information. Display domains are modeled as classes with the stereotype ≪`DisplayDomain`≫. The context diagram in Fig. 1 shows the biddable domains Patient and PhysiciansAndNurses, and the causal domains O2Sensor, HeartbeatSensor, InfusionPump, and Terminal. These causal domains are also connection domains, and the Terminal is a display domain.

Interfaces connect domains, and they contain *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. These interfaces are represented as associations, and the name of the associations contain the phenomena and the domain controlling the phenomena. For example, in Fig. 1 the notation HS!{Heartbeat} means that the phenomenon Heartbeat is controlled by the domain HeartbeatSensor.

Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge*. The domain knowledge consists of *assumptions* and *facts*. Assumptions are conditions that are needed, so that the *requirements* are accomplishable. Usually, they describe required user behavior. For example, it must be assumed that a user ensures not to be observed by a malicious user when entering a password. Facts describe fixed properties of the problem environment, regardless of how the machine is built.

Domain knowledge and requirements are special statements. A statement is modeled as a class with a stereotype. In this stereotype, a unique identifier and the statement text are contained as stereotype attributes. When a requirement

No	Requirement	≪refersTo≫	≪constrains≫
R1	The vital signs should be displayed, and an alarm should be raised if the vital signs exceed the limits.	Patient, Configuration	Terminal
R2	Physicians and nurses can change the configuration.	PhysiciansAndNurses	Configuration
R3	The infusion flow is controlled according to the configured doses for the current vital signs.	Patient, Configuration	InfusionPump

Tab. 1. Functional Requirements of Patient Monitoring System

No	Security Statement	≪complements≫	≪refersTo≫	≪constrains≫/ Mechanism
1	Configuration should be protected from modification for Patient against Attacker or PhysiciansAndNurses should be informed.	R2	Configuration is asset, Terminal and WLAN know asset, Patient is stakeholder, against Attacker	TerminalDisplay/ MAC of SSL
2	Alarm and Vital Signs should be protected from modification for Patient against Attacker or PhysiciansAndNurses should be informed.	R1	Alarm and Vital Signs are assets, Terminal and WLAN know asset, Patient is stakeholder, against Attacker	TerminalDisplay/ MAC of SSL
3	Configuration, Alarm, and Vital Signs should be protected from disclosure for Patient against Attacker.	R1, R2	Configuration, Alarm, and Vital Signs are assets, Patient is stakeholder, against Attacker	WLAN/ encryption of SSL
4	The Shared Keys should be distributed to Terminal and PMS (for Patient) and Attacker should not be able to access Shared Keys.	R1, R2	Shared Keys are assets, Patient is stakeholder, against Attacker	WLAN/ key exchange of SSL (KE)

Tab. 2. Security Requirements of Patient Monitoring System

is stated, this means that something in the world should be changed by integrating the machine to be developed into it. Therefore, each requirement constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype ≪constrains≫. A requirement may refer to several domains in the environment of the machine. For example, security requirements have to refer to an attacker of a certain strength. These references are expressed by a dependency from the requirement to a domain with the stereotype ≪refersTo≫. The domains referred are also given in the requirements description. Table 1 lists the functional requirements of the PMS case study.

Security requirements are associated with functional requirements, which we express using the stereotype ≪complements≫. For the functional requirements listed in Tab. 1, we initially identified some security requirements, as shown in Tab. 2 in rows 1-3, expressed as proposed in [5]. The required integrity (rows 1

No	Security Statement	≪complements≫	≪refersTo≫	≪constrains≫/ Mechanism
1	The KE keys should be distributed to Terminal and PMS for Patient, and Attacker should not be able to access Shared Keys.	R1, R2	KE keys are assets, Patient is stakeholder, against Attacker	WLAN/ manual import in physically protected area
2	Infusion Flow and PatientMonitoringSystem should be protected from modification for Patient against Attacker or Patient should know.	R1, R2, R3	Infusion Flow and Patient-Monitoring-System are assets, Patient is stakeholder, against Attacker	Infusion Pump, PatientMonitoring-System/ physical protection (e.g., EMF) and protection by Patient
3	Infusion Flow and PatientMonitoringSystem should be protected from disclosure for Patient against Attacker.	R1, R2, R3	Infusion Flow and Patient-Monitoring-System are assets, Patient is stakeholder, against Attacker	Infusion Pump, PatientMonitoring-System/ physical protection (e.g., EMF) and protection by Patient
4	Terminal should be protected from modification for Patient against Attacker or PhysiciansAndNurses should know.	R1, R2	Terminal is asset, Patient is stakeholder, against Attacker	Terminal/ physical protection (e.g., EMF) and protection by PhysiciansAndNurses
5	Terminal should be protected from disclosure for Patient against Attacker.	R1, R2	Terminal is asset, Patient is stakeholder, against Attacker	Terminal/ physical protection (e.g., EMF) and protection by PhysiciansAndNurses

Tab. 3. Security Domain Knowledge of Patient Monitoring System

and 2) supports the safety of the system and the required confidentiality (row 3) is necessary for privacy reasons. We decide on generic mechanisms that represent solutions of these requirements. To implement these mechanisms, additional domains have to be introduced, and additional requirements have to be fulfilled.

We choose the security mechanism MAC (Message Authentication Code) for integrity and symmetric encryption for confidentiality. For the mechanisms MAC and encryption, a Shared Key known by the Terminal and by the PMS is necessary. As required in Tab. 2 in row 4, this Shared Key must be distributed to the Terminal and to the PMS. The integrity and confidentiality of the Shared Key must be preserved. This will be implemented using a key exchange protocol. For the key exchange, additional secrets (KE keys) are necessary.

The KE keys should be distributed manually as described in Tab. 3 in row 1. Integrity and confidentiality of the Infusion Flow and the PatientMonitoringSystem should be ensured by physical protection (e.g., by reducing electromagnetic field (EMF) radiation and by protection against EMF radiation) and protection by Patient (e.g., Patient prevents physical access to the Infusion Flow) (Tab. 3 in rows 2 and 3). Integrity and confidentiality of the Terminal should be ensured by physical protection (e.g., by reducing electromagnetic field radiation and by protection against EMF radiation) and protection by PhysiciansAndNurses (Tab. 3 in rows 4 and 5).

For reasons of space, we do not depict the UML diagrams equipped with the mentioned stereotypes capturing these security requirements and the security domain knowledge. Instead, we present an overview of the security requirements and the security domain knowledge in Tabs. 2 and 3. These statements are the starting point for developing the design of the machine, which we achieve using UMLsec.

3 UMLsec

UMLsec constitutes a UML profile to develop and analyze security models. UMLsec offers new UML language elements, i.e., *stereotypes*, *tags*, and *constraints*, to specify typical security requirements such as secrecy, integrity, and authenticity, and attacker models. Examples for pre-defined UMLsec stereotypes are ≪`critical`≫ to label security-critical parts of UML diagrams, ≪`secure dependency`≫ to ensure that dependent parts of models preserve the security requirements relevant for the parts they depend on, ≪`secure links`≫ to introduce attacker models, and ≪`data security`≫ to analyze behavior models with respect to confidentiality and integrity requirements. The aforementioned stereotypes are used in the next section for creating UMLsec design models based on results from security requirements engineering. A detailed explanation and a formal foundation of the tags and stereotypes defined in UMLsec can be found in [9].

Based on UMLsec models and the semantics defined for the different UMLsec language elements, possible security vulnerabilities can be identified at a very early stage of software development. One can thus verify that the desired security requirements, if fulfilled, enforce a given security policy. This verification is supported by a tool suite, which is available online via `http://www.umlsec.de/`.

4 From Security Requirements to UMLsec Design Models

In this section, we connect the security requirements engineering approach presented in Sect. 2 with secure design based on UMLsec. We first present a procedure to generate UMLsec diagrams describing the environment in Sect. 4.1. Second, we introduce a procedure to generate UMLsec diagrams describing security mechanisms in Sect. 4.2. These procedures are supported by *model generation rules*, which we express using the formal specification language OCL. More precisely, the model generation rules consist of OCL *pre- and postconditions*. They can be considered as *patterns* that describe how existing security measures and cryptographic protocols can be developed based on results from security requirements engineering.

We finally present in Sect. 4.3 work in progress on the construction of a tool that realizes the aforementioned procedures to develop UMLsec design models based on security requirements.

4.1 UMLsec Deployment Diagrams for Environment Descriptions

According to our security requirements engineering approach as illustrated in Sect. 2, describing the operational environment of a secure software system is of great importance. In fact, the environment description is also necessary for secure design: security-critical design decisions should lead to the fulfillment of the security requirements in the given environment. However, in a different environment, the same design decisions might lead to an insecure system.

In the following, we present a procedure to develop deployment diagrams enriched with UMLsec elements from context diagrams and security requirements. For each step, an operation name with parameters is provided. These operations represent model generation rules.

1. Create a UML package named adequately that contains a deployment diagram *(it is required that such a diagram does not yet exist and that exactly one context diagram exists)*.
   `createDeploymentDiagram(diagramName: String)`
2. Add the ≪`secure links`≫ stereotype to the package and assign a certain type of attacker (e.g., *default* or *insider* as described in [9, Chapter 4.1]) to the {`adversary`} tag. Decide which attacker type is appropriate based on threats modeled in the context diagram and domain knowledge collected during security requirements engineering. For example, *default* attackers cannot execute attacks in a LAN environment, but *insider* attackers can. Hence, if the context diagram describes an attack in a LAN environment, the attacker is of type *insider*.
   `addSecureLinksStereotype(inDiagram: String, adv: String)`
3. Each domain contained in the context diagram *(it is required that exactly one context diagram exists and that the deployment diagram exists)* that is not a biddable domain is represented as a node in the deployment diagram.
   `createNodes(inDiagram: String)`
4. Moreover, each domain that is part of another domain in the context diagram is represented either as a nested node or class.
   `createNestedNodes(domainNames: String[])` or `createNestedClasses(domainNames: String[])`
5. Each connection between the aforementioned domains is represented as a communication path and a dependency:
   (a) We create a communication path stereotyped according to the communication type as described in Tab. 4. Note that only one of the UMLsec stereotypes is allowed for each communication path. Moreover, the defined mapping for context diagram stereotypes also applies to sub-stereotypes. For example, ≪`wireless`≫ is a sub-stereotype of ≪`network_connection`≫, and therefore, ≪`wireless`≫ can be mapped to ≪`Internet`≫, ≪`LAN`≫, and ≪`encrypted`≫, too.
   We create communication paths for all relevant associations, and we also associate a communication type where no decision is necessary (`createCommunicationPaths(inDiagram: String)`). For all network connections (retrievable with `getNetworkConnections(): String[]`), the developer has to choose between ≪`Internet`≫, ≪`LAN`≫, or ≪`Encrypted`≫ (`setCommunicationPathType(inDiagram: String, assName: String, type: String)`).
   (b) We create a dependency stereotyped according to the control direction of the interfaces in the security requirement diagram and according to the following rules:

Context Diagram	*UMLsec Deployment Diagram*
≪physical≫	≪wire≫ (physical protection against default adversary is assumed)
≪ui≫	not considered since biddable domains are not part of deployment diagrams
≪remote_call≫	see ≪network_connection≫
≪network_-connection≫	≪Internet≫, ≪LAN≫, ≪encrypted≫ depending on the domain knowledge collected during security requirements engineering

Tab. 4. From Context Diagrams to UMLsec Deployment Diagrams

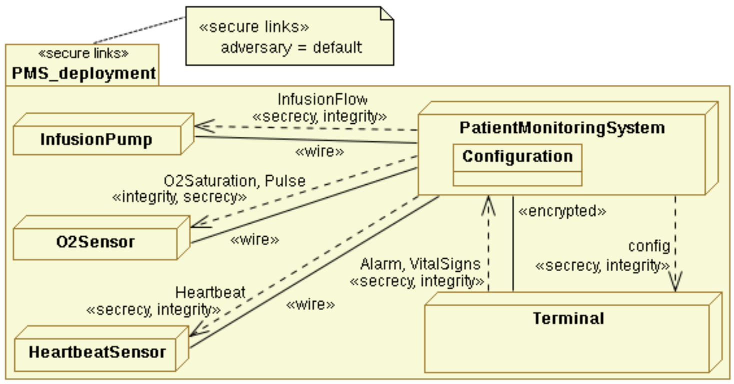

Fig. 2. UMLsec Deployment Diagram Representing the Target State of Patient Monitoring System

- The domain controlling the interface is translated into the target of the dependency.
- If more than one observing domains exist, the same number of dependencies must be introduced.
- If a confidentiality statement constraining the connection domain of the corresponding connection in the security requirement diagram exists, then the dependency is stereotyped ≪secrecy≫.
- If an integrity statement referring to the connection domain of the corresponding connection in the security requirement diagram exists, then the dependency is stereotyped ≪integrity≫.

`createDependencies(inDiagram: String)`

The result of applying this method to the context diagram of the patient monitoring system shown in Fig. 1 is presented in Fig. 2. This UMLsec deployment diagram can be created following the command sequence depicted in Listing 1.1.

We now present the OCL specification of the model generation rule for step 5. Listing 1.2 contains the specification for step 5, generating the communication paths and stereotypes for those associations that can be derived directly. The first two formulas of the precondition of the model generation rule `createCommunicationPaths(inDiagram: String)` state that there does not exist a package named equal to the parameter `diagramName` (lines 2-3 in

```
createDeploymentDiagram('PMS_Deployment');
addSecureLinksStereotype('PMS_Deployment','default');
createNodes('PMS_Deployment');
createNestedClasses({'Configuration'});
getNetworkConnections(); -- returns {'PMS!{Alarm,VitalSigns},T!{config}'}
createCommunicationPaths('PMS_Deployment');
setCommunicationPathType('PMS_Deployment','PMS!{Alarm,VitalSigns},
     T!{config}','encrypted');
createDependencies('PMS_Deployment');
```

Listing 1.1. Generating a UMLsec Deployment Diagram

```
1  createCommunicationPaths(inDiagram: String)
2  PRE   Package.allInstances() ->select(name=diagramName)
3            ->size()=1 and
4        Package.allInstances() ->select(getAppliedStereotypes()
5          .name ->includes('ContextDiagram')) ->size()=1 and
6        Package.allInstances() ->select(getAppliedStereotypes()
7         .name ->includes('ContextDiagram')) .clientDependency
8         .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
9          ->select(not endType.getAppliedStereotypes().name
10           ->includes('BiddableDomain')
11        ).getAppliedStereotypes() ->forAll(rel_ass_st|
12            not rel_ass_st.name ->includes('ui') and
13            not rel_ass_st.general.name ->includes('ui') and
14            -- similar for 'event', 'call_return', 'stream', 'shared_memory'
15        )
16 POST  Package.allInstances() ->select(name=inDiagram).ownedElement
17        ->select(oclIsTypeOf(CommunicationPath))
              .oclAsType(CommunicationPath
18       .endType.name =
19       Package.allInstances() ->select(getAppliedStereotypes()
20       .name ->includes('ContextDiagram')) .clientDependency
21       .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
22        ->select(not endType.getAppliedStereotypes().name
23        ->includes('BiddableDomain')).endType.name and
24       Package.allInstances() ->select(getAppliedStereotypes()
25       .name ->includes('ContextDiagram')) .clientDependency
26       .target ->select(oclIsTypeOf(Association)) .oclAsType(Association)
27        ->select(not endType.getAppliedStereotypes().name
28        ->includes('BiddableDomain')) ->forAll(rel_ass|
29         Package.allInstances() ->select(name=inDiagram).ownedElement
30          ->select(oclIsTypeOf(CommunicationPath))
                .oclAsType(CommunicationPath)
31          ->exists(cp |
32            cp.name = rel_ass.name and
33            cp.endType.name = rel_ass.endType.name and
34            ( cp.getAppliedStereotypes().name ->includes('physical') implies
35               rel_ass.getAppliedStereotypes().name ->includes('wire')) and
36            ( cp.getAppliedStereotypes().general.name ->includes('physical')
                 implies
37               rel_ass.getAppliedStereotypes().name ->includes('wire'))
38          )
39       )
```

Listing 1.2. createCommunicationPaths(inDiagram: String)

Listing 1.2), and that there exists a package that contains a diagram stereotyped ≪`ContextDiagram`≫ (lines 4-5). The third formula of the precondition expresses that associations between transformed domains do not contain any of the ≪`ui`≫, ≪`event`≫, ≪`call_return`≫, ≪`stream`≫, ≪`shared_memory`≫, stereotypes and subtypes (lines 6-15). If these conditions are fulfilled, then the postcondition can be guaranteed, i.e., names of nodes connected by each communication path are the same as the names of domains connected by an association in the context diagram (lines 16-29), and there exists for each relevant association contained in the context diagram a corresponding and equally named communication path in the deployment diagram that connects nodes with names equal to the names of the domains connected by the association. These communication paths are stereotyped ≪`wire`≫ if the corresponding associations are stereotyped ≪`physical`≫ or a subtype (lines 30-39).

4.2 UMLsec Class and Sequence Diagrams for Security Mechanism Descriptions

In the following, we show how to specify security mechanisms by developing UMLsec diagrams based on security requirements. For each communication path contained in the UMLsec deployment diagram developed as shown in Sect. 4.1 that is not stereotyped ≪`wire`≫, we select an appropriate security mechanism according to the results of the problem analysis, e.g., MAC for integrity, symmetric encryption for security, and a protocol for key exchange, see Tab. 2). A security mechanism specification commonly consists of a structural and a behavioral description, which we specify based on the UMLsec ≪`data security`≫ stereotype. To create security mechanism specifications, we developed a number of model generation rules, for example:

– Securing data transmissions using MAC: `createMACSecuredTransmission(senderNodeName: String, receiverNodeName: String, newPackage: String)`
– Symmetrically encrypted data transmissions: `createSymmetricallyEncryptedTransmission(senderNodeName: String, receiverNodeName: String, newPackage: String)`
– Key exchange protocol: `createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String, newPackage: String)`

Model generation rules can be regarded as *patterns* for security mechanism specifications. Each of the aforementioned model generation rules describes the construction of a package stereotyped ≪`data security`≫ containing structural and behavioral descriptions of the mechanism expressed as class and sequence diagrams. Moreover, the package contains a UMLsec deployment diagram developed as shown in Sect. 4.1.

We explain in detail the model generation rule `createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String, newPackage: String)` shown in Listing 1.3. We use this protocol to realize the security requirement given in Table 2, row 4, of the patient monitoring system. We use the protocol that secures data transmissions using MACs for the security requirements in rows 1 and 2, and we use the protocol for symmetrically encrypted data transmissions for the security requirement in row 3.

The precondition of the model generation rule for key exchange protocols states that nodes named `initiatorNodeName` and `responderNodeName` exist (lines 2-3 in Listing 1.3). The communication path between these nodes (line 8) should have the stereotype ≪`encrypted`≫, ≪`Internet`≫, or ≪`LAN`≫ (lines 9-10). Additionally, a package named `newPackage` must not exist (line 11). If these conditions are fulfilled, then the postcondition can be guaranteed. The first part of the postcondition describes the construction of a class diagram, and the second part specifies the construction of a sequence diagram. The following class diagram elements are created as shown in the example in Fig. 3:

```
1  createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName:
       String, newPackage: String);
2  PRE   Node.allInstances() ->select(name=initiatorNodeName) ->size()=1 and
3        Node.allInstances() ->select(name=responderNodeName) ->size()=1 and
4        let cp_types: Bag(String) =
5          CommunicationPath.allInstances()->select( cp |
6            cp.endType->includes(Node.allInstances()
                 ->select(name=initiatorNodeName)->asSequence()->first() )
                 and
7            cp.endType->includes(Node.allInstances()
                 ->select(name=responderNodeName)->asSequence()->first() )
8          ).getAppliedStereotypes().name
9        in
10         cp_types->includes('encrypted') or cp_types->includes('Internet')
               or cp_types->includes('LAN') and
11       Package.allInstances() ->select(name=newPackage) ->size()=0
12
13 POST  Package.allInstances() ->select(name=newPackage) ->size()=1 and
14       -- ... Stereotype with attributes exists
15       Class.allInstances() ->select(name=initiatorNodeName)
             ->select(oclIsTypeOf(Class)) ->size()=1 and
16       Class.allInstances() ->select(name=responderNodeName)
             ->select(oclIsTypeOf(Class)) ->size()=1 and
17       -- ... dependencies with secrecy and integrity between initiator
             and responder (both direction) created ...
18       Class.allInstances() ->select(name=initiatorNodeName)
             ->select(oclIsTypeOf(Class)).ownedAttribute
19         ->select(name='inv(K_T)').type ->select(name = 'Keys') -> size()
               = 1 and
20       -- ... other attributes exist...
21       Class.allInstances() ->select(name=initiatorNodeName)
             ->select(oclIsTypeOf(Class)).ownedOperation
22       ->select(name='resp')
23       ->select( member->forAll(oclIsTypeOf(Parameter))) .member ->forAll(
             par |
24         par->select( name->includes('shrd')) ->one(
               oclAsType(Parameter).type.name->includes('Data')) xor
25         par->select( name->includes('cert')) ->one(
               oclAsType(Parameter).type.name->includes('Data'))
26       ) and
27       -- ... other operations exist
28       -- ... stereotype and tags for initiator and responder class exist
29       let intera : Bag(Interaction) =
30         Package.allInstances() ->select(name=newPackage) .ownedElement
               ->select(oclIsTypeOf(Collaboration))
31         .ownedElement ->select(oclIsTypeOf(Interaction))
               .oclAsType(Interaction)
32       in
33         intera.ownedElement ->select(oclIsTypeOf(Lifeline))
               .oclAsType(Lifeline).name ->includes(initiatorNodeName) and
34         intera.ownedElement ->select(oclIsTypeOf(Lifeline))
               .oclAsType(Lifeline).name ->includes(responderNodeName) and
35         intera.ownedElement ->select(oclIsTypeOf(Message))
               .oclAsType(Message).name
               ->includes('init(N_i,K_T,Sign(inv(K_T),T::K_T))') and
36         intera.ownedElement ->select(oclIsTypeOf(Message))
               .oclAsType(Message).name
               ->includes('resp({Sign(inv(K_P_i),k_j::N'::K'_T)}_K'_T,
               Sign(inv(K_CA),P_i::K_P_i))') and
37         intera.ownedElement ->select(oclIsTypeOf(Message))
               .oclAsType(Message).name ->includes('xchd({s_i}_k)') and
38       -- ... conditions in sequence diagram exist
```

Listing 1.3. createKeyExchangeProtocol(initiatorNodeName: String, responderNodeName: String, newPackage: String)

- exactly one package named `newPackage` (line 13)
- stereotype ≪`data security`≫ and tags (adversary) for this package

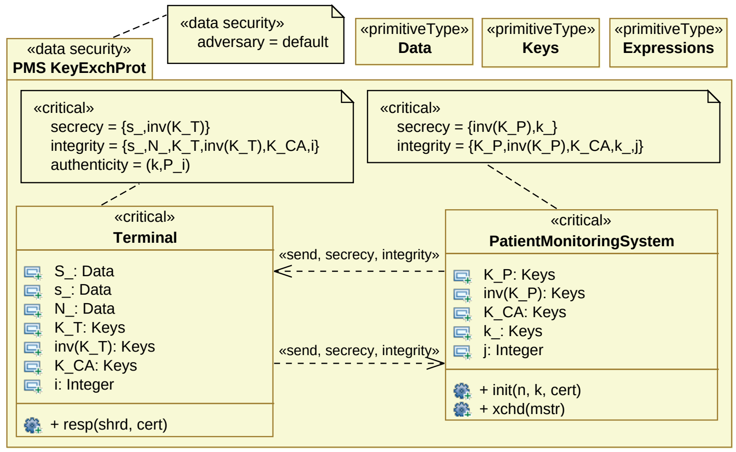

Fig. 3. Class Diagram of Key Exchange Protocol for Patient Monitoring System

- classes for initiator and responder named `initiatorNodeName` and `responderNodeName` (lines 15-16)
- dependencies with ≪`secrecy`≫ and ≪`integrity`≫ between initiator and responder (both directions)
- attributes for initiator and responder classes (lines 18-20)
- methods with parameters for initiator and responder class (lines 21-27)
- stereotype ≪`critical`≫ and corresponding tags (e.g., secrecy) for initiator and responder classes

The following sequence diagram elements are created as shown in the example in Fig. 4:

- lifelines for initiator and for responder in an interaction being part of a collaboration that is part of the created package (lines 29-34)
- messages in sequence diagram (lines 35-37)
- conditions in sequence diagram

A detailed description of this protocol pattern is given in [9, Chapter 5.2].

Figure 3 shows the class diagram and Fig. 4 the sequence diagram developed for the patient monitoring system according to this model generation rule. They are created with `createKeyExchangeProtocol('Terminal', 'PatientMonitoringSystem', 'KeyExchProt')`. In the created model, the tag {`secrecy`} of the ≪`critical`≫ class Terminal contains the secret s_, which represents an array of secrets to be exchanged in different rounds of this protocol. It also contains the private key inv(K_T) of the Terminal. Next to these assets, the {`integrity`} tag additionally contains the nonces N_ used for the protocol, the public key K_T of the Terminal, the public key K_CA of the certification authority, and the round iterator i. These tag values are reasonable because the security domain knowledge in Tab. 3, rows 2 and 3 states that the PatientMonitoringSystem with its contained data is kept confidential and its integrity is preserved. The tag {`authenticity`} expresses that the PatientMonitoringSystem P_i is authenticated with respect to the Terminal. This is ensured by the domain knowledge in Tab. 3, row 1. The tag {`secrecy`} of the ≪`critical`≫ class PatientMonitoringSystem contains the

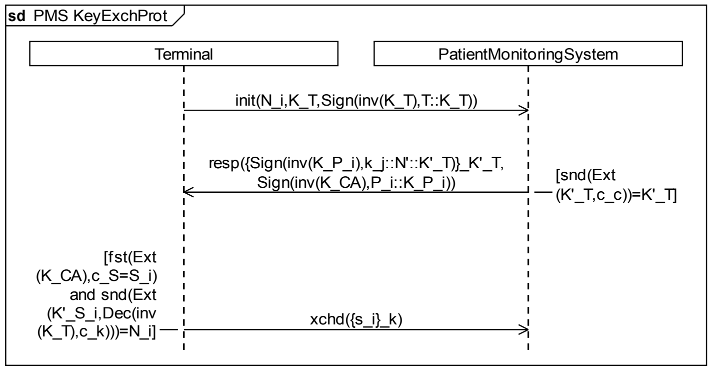

Fig. 4. Sequence Diagram of Key Exchange Protocol for Patient Monitoring System

session keys k_ and the private key inv(K_P) of the PatientMonitoringSystem. The {integrity} tag consists of assets similar to the ones of the same tag of the Terminal. The tag {authenticity} is not used, since two-sided authentication is not necessary. Integrity and confidentiality of the data stored in the PatientMonitoringSystem (private key inv(K_P), the public key K_P, the public key K_CA of the certification authority, and the round iterator j) is covered by the domain knowledge in Tab. 3, rows 4 and 5.

The sequence diagram in Fig. 4 specifies three messages and two guards, and it considers the ith protocol run of the Terminal, and the jth protocol run of the PatientMonitoringSystem. The sequence counters i and j are part of the Terminal and the PatientMonitoringSystem, respectively. The init(...) message sent from the Terminal to the PatientMonitoringSystem initiates the protocol. If the guard at the lifeline of the PatientMonitoringSystem is true, i.e., the key K_T contained in the signature matches the one transmitted in the clear, then the PatientMonitoringSystem sends the message resp(...) to the Terminal. If the guard at the lifeline of the Terminal is true, i.e., the certificate is actually for S and the correct nonce is returned, then the Terminal sends xchd(...) to the PatientMonitoringSystem. If the protocol is executed successfully, i.e., the two guards are evaluated to true, then both parties share the secret s_i.

The key exchange protocol only fulfills the corresponding security requirements if integrity, confidentiality, and authenticity of the keys are ensured. According to our pattern system for security requirements engineering [5], applying the key exchange mechanism leads to dependent statements about integrity, confidentiality, and authenticity of the keys as stated in Tab. 3.

4.3 Tool Design

We are currently constructing a graphical wizard-based tool that supports a software engineer in interactively generating UMLsec design models. The tool will implement the model generation rules presented in the previous subsections to generate UMLsec deployment, class, and sequence diagrams. A graphical user interface allows users to choose the parameters, and it ensures that these parameters fulfill the preconditions. For example, users can choose the value of the second parameter of the model generation rule `setCommunicationPathType(inDiagram: String, assName: String, type: String)` based on the return values of the rule `getNetworkConnections()`. Our tool will automatically construct the corresponding parts of the UMLsec model as described in the postcondition. Since our model generation rules are specified with OCL in a formal and analyzable way, our tool implementation can be checked automatically for correctness with respect to our specification based on an appropriate API such as the Eclipse implementation for EMF-based models [7]. In addition to realizing the OCL specification, the tool will support workflows adequate to generate the desired UMLsec models, e.g., as depicted in Listing 1.1.

In summary, we presented in this section a novel integrated and formal approach connecting security requirements analysis and secure design.

5 Related Work

The approach presented in this paper can be compared on the one hand-side to other work bridging the gap between security requirements engineering secure design, and on the other hand-side to work on transforming UML models based on rules expressed in OCL.

Relatively little work has been done on the first category of related work, i.e., bridging the gap between security requirements analysis and design. Recently, an approach [12] to connect the security requirements analysis method *Secure Tropos* by Mouratidis et al. [4] and UMLsec [9] is published. A further approach [7] connects UMLsec with security requirements analysis based on heuristics. In contrast to our work, these approaches only provide informal guidelines for the transition from security requirements to design. Consequently, they do not allow to verify the correctness of this transition step.

The second category of related work considers the transformation of UML models based on *OCL transformation contracts* [1, 11]. We basically use parts of this work, e.g., the specification of transformation operations using OCL pre- and postconditions. Additionally, our model generation rules can be seen as patterns, since they describe the generation of completely new model elements according to generic security mechanisms, e.g., cryptographic keys.

6 Conclusions and Future Work

We presented in this paper a *novel method* to bridge the gap between security requirements analysis and secure design. We complemented our method by *formal model generation rules* expressed in OCL. Thus, the construction of UMLsec design models based on results from security requirements engineering becomes *more feasible*, *systematic*, *less error-prone*, and a *more routine* engineering activity. We illustrated our approach using the sample development of a patient monitoring system.

In the future, we would like to elaborate more on the connection between the presented security requirements engineering approach and UMLsec. For example, we intend to develop a notion of correctness for the step from security requirements engineering to secure design based on the approach presented in this paper.

[7] *Eclipse Modeling Framework (EMF)*:`http://www.eclipse.org/modeling/emf/`

References

[1] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the specification of model transformation contracts. In *Proceedings of the Workshop on OCL and Model Driven Engineering at the International UML Conference LNCS 3273*. Springer, 2004.

[2] I. Côté, D. Hatebur, M. Heisel, H. Schmidt, and I. Wentzlaff. A systematic account of problem frames. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 749–767. Universitätsverlag Konstanz, 2008.

[3] B. Fabian, S. Gürses, M. Heisel, T. Santen, and H. Schmidt. A comparison of security requirements engineering methods. *Requirements Engineering – Special Issue on Security Requirements Engineering*, 15(1):7–40, 2010.

[4] P. Giorgini and H. Mouratidis. Secure tropos: A security-oriented extension of the tropos methodology. *International Journal of Software Engineering and Knowledge Engineering*, 17(2):285–309, 2007.

[5] D. Hatebur and M. Heisel. A UML profile for requirements analysis of dependable software. In E. Schoitsch, editor, *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP) (LNCS 6351)*, pages 317–331. Springer, 2010.

[6] D. Hatebur, M. Heisel, and H. Schmidt. Analysis and component-based realization of security requirements. In *Proceedings of the International Conference on Availability, Reliability and Security (AReS)*, pages 195–203. IEEE Computer Society, 2008.

[7] S. H. Houmb, S. Islam, E. Knauss, J. Jürjens, and K. Schneider. Eliciting security requirements and tracing them to design: An integration of common criteria, heuristics, and UMLsec. *Requirements Engineering – Special Issue on Security Requirements Engineering*, 15(1):63–93, 2010.

[8] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[9] J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[10] T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *Proceedings of the International Conference on the Unified Modeling Language (UML)*, pages 426–441, London, UK, 2002. Springer.

[11] T. Millan, L. Sabatier, T.-T. Le Thi, P. Bazex, and C. Percebois. An OCL extension for checking and transforming uml models. In *Proceedings of the WSEAS International Conference on Software Engineering, Parallel and distributed Systems (SEPADS)*, pages 144–149, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).

[12] H. Mouratidis and J. Jürjens. From goal-driven security requirements engineering to secure design. *International Journal of Intelligent Systems – Special issue on Goal-Driven Requirements Engineering*, 25(8):813 – 840, June 2010.

[13] H. Schmidt. *A Pattern- and Component-Based Method to Develop Secure Software*. Deutscher Wissenschafts-Verlag (DWV) Baden-Baden, April 2010.