Vorlesung

Methodische Grundlagen des Software-Engineering

im Sommersemester 2013

Prof. Dr. Jan Jürjens

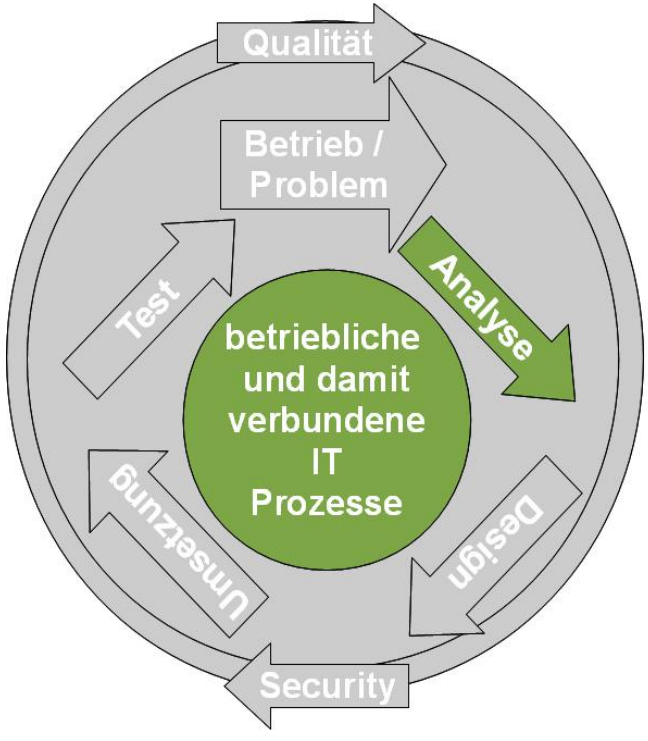
TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

3.4: Design Principles

v. 19.06.2013

3.4 Design Principles

- Geschäftsprozessmodellierung
- Process-Mining
- Modellbasierte Entwicklung sicherer Software
  - Model-Driven Architecture
  - Sicherheitsanforderungen
  - UMLsec
  - Design Principles
  - UML model analysis
  - Examples
    - TLS Variant
    - CEPS Purchase



Rules of Prudent Security Engineering

Saltzer, Schroeder (1975):

- Design principles for security-critical systems.
- Check how to enforce these with UMLsec.

Economy of Mechanism

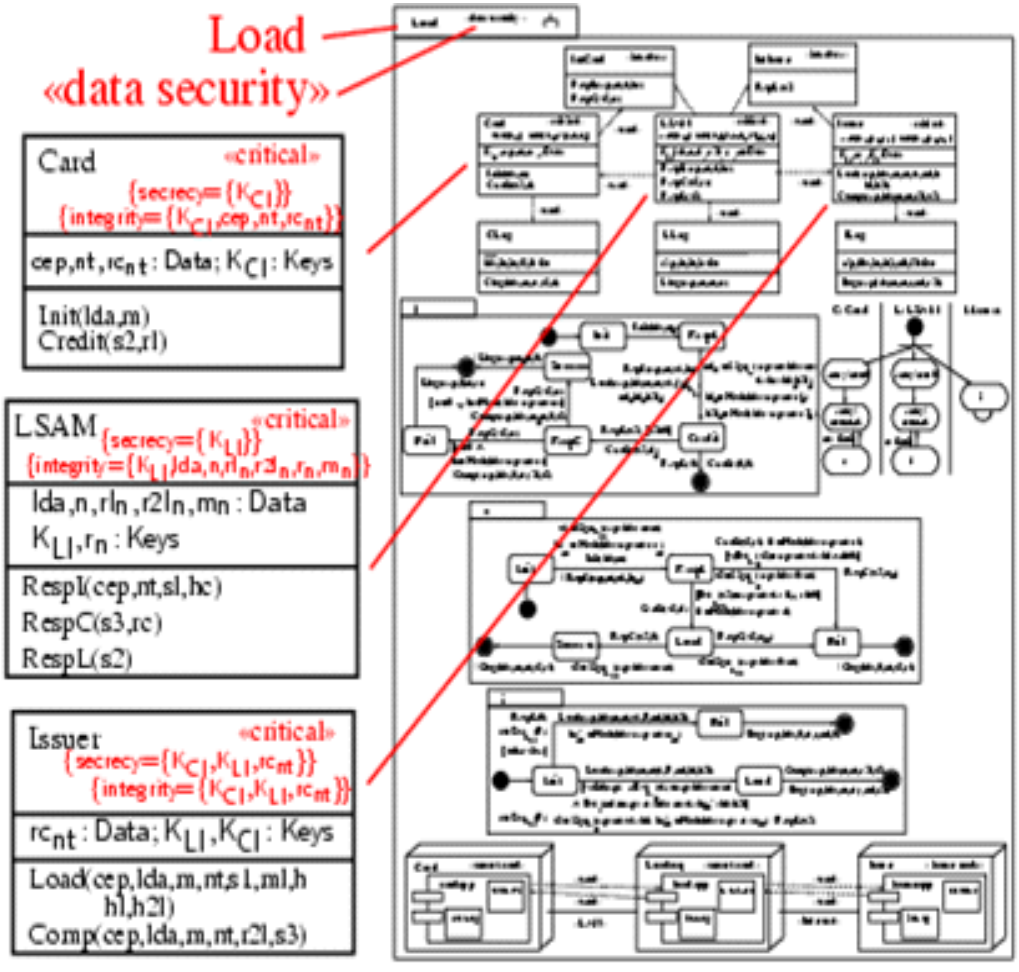
Keep design as simple and small as possible.

- Often systems made complicated to make them (look) secure.
- Method for reassurance may reduce this temptation.
- Payoffs from formal evaluation may increase incentive for following the rule.

Fail-safe Defaults

Base access decisions on permission rather than exclusion.

Example: secure log-keeping for audit control in Common Electronic Purse Specifications (CEPS).

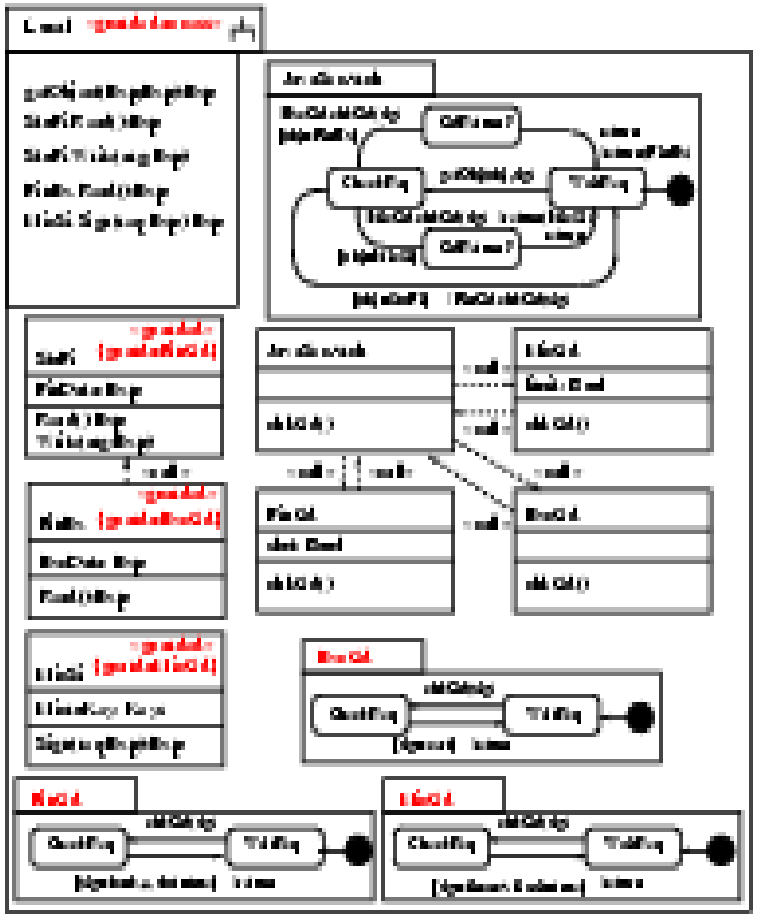


Complete Mediation

Every access to every object must be checked for authority.

E.g. in Java: use guarded objects. Use UMLsec to ensure proper use of guards.

More feasibly, mediation with respect to a set of sensitive objects.



Open Design

The design should not be secret.

Method of reassurance may help to develop systems whose security does not rely on the secrecy of its design.

Separation of Privilege

A protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.

Example: signature of two or more principals required for privilege. Formulate requirements with activity diagrams.

Verify behavioural specifications with respect to them.

Least Privilege

Every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Least privilege: every proper diminishing of privileges gives system not satisfying functionality requirements.

Can make precise and check this.

Least Common Mechanism

Minimize the amount of mechanism common to more than one user and depended on by all users.

Object-orientation:

- data encapsulation.
- data sharing well-defined (keep at necessary minimum).

Psychological Acceptability

Human interface must be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

With respect to development process: ease of use in development of secure systems.

User side: e.g. performance evaluation (acceptability of performance impact of security).

Discussion

No absolute rules, but warnings.

Violation of rules symptom of potential trouble; review design to be sure that trouble accounted for or unimportant.

Design principles reduce number and seriousness of flaws.

Patterns

Patterns* encapsulate the design knowledge of software engineers by presenting recurring design problems and standardized solutions.

One can use transformations of UMLsec models to introduce patterns within the design process.

- Goal: ensure that the patterns are introduced in a way that has previously been shown to be useful and correct.
- Also: having a sound way of introducing patterns using transformations can ease security analysis, since the analysis can be performed on the more abstract and simpler level.

(* E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.)

Application of a pattern (1/2)

The application of a pattern *p* corresponds to a function f_p

- which takes a UML specification *S*
- and returns a UML specification, namely the on obtained when applying *p* to *S*.

Technically, such a function can be presented by

- defining how it should act on certain subsystem instances.
- extending it to all possible UML specifications in a compositional way.

Application of a pattern (2/2)

- We have a set S of subsystem instances such that none of the subsystem instances in S is contained in any other subsystem instance in S.
- For every subsystem instance $s \in S$ we are given a subsystem instance $f_p(s)$.
- Then for any UML specification U, we can define $f_p(U)$ by substituting each occurrence of a subsystem instance $s \in S$ in U by $f_p(s)$.
- The challenge: define such a function f_p that is applicable as widely as possible.
- How to do this on a technical level is beyond the scope of this presentation.

Secure Architecture Patterns

Architectural design patterns (Buschmann et al. 1996). Apply to security.

Example: Architectural primitive: Secure channel.

- Define a secure channel abstraction.
- Define concrete secure channel (protocol).
- Show simulates the abstraction.

Give conditions under which it is secure to substitute channel abstractions by concrete protocols.

Secure Channel Abstractions

So far, usually concentrated on specific properties of protocols in isolation.

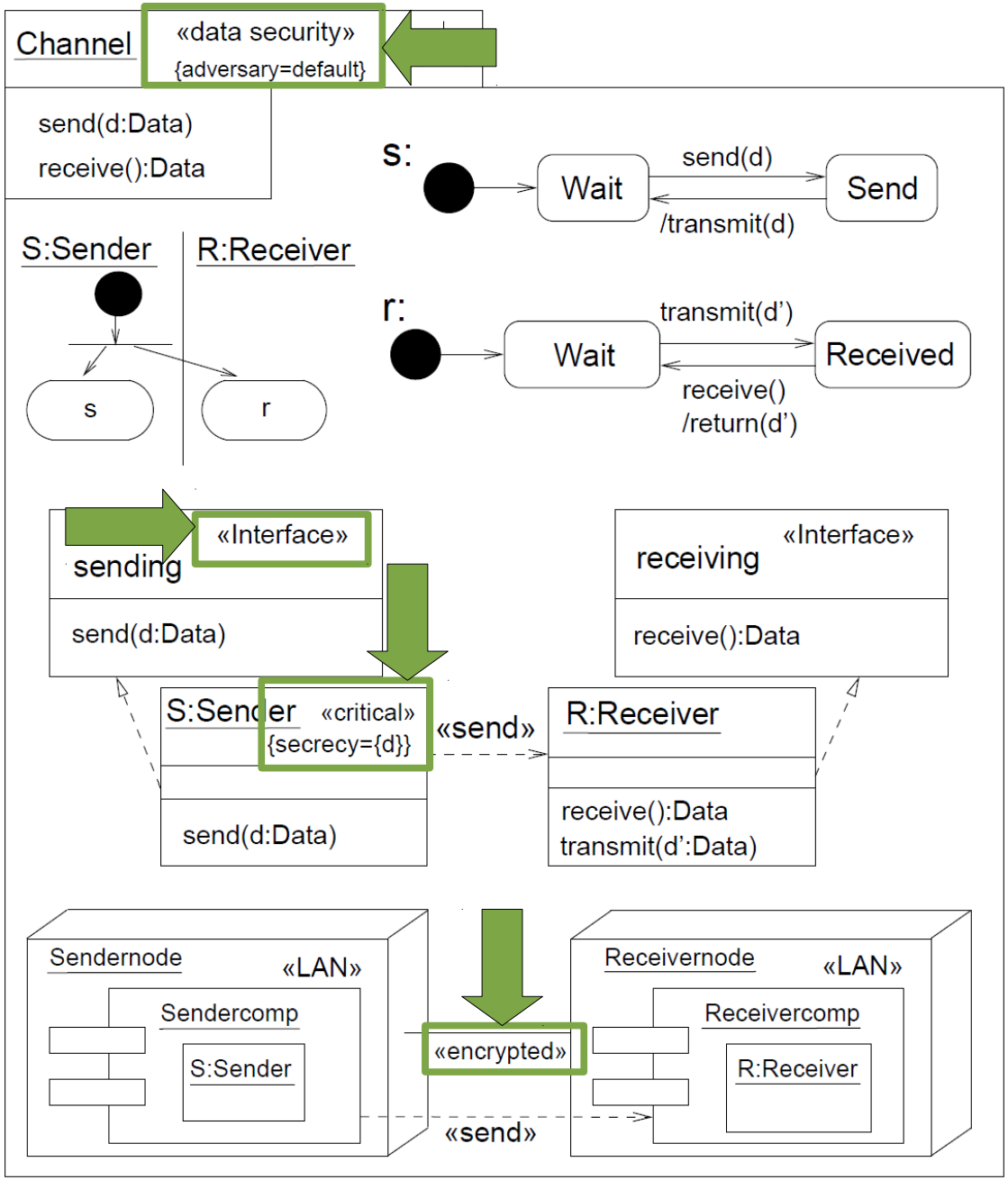
Need to refine security properties so protocol is still secure in system context. Surprisingly problematic.

Motivates research towards providing secure channel abstractions to use security protocols securely in the system context.

Secure Channel Pattern: Approach

- Define a secure channel abstraction.
- Define concrete secure channel (protocol).
- Show concrete secure channel simulates the abstraction.
  - Give conditions under which it is secure to substitute channel abstractions by concrete protocol.

Secure Channel Pattern: Abstract specification



To keep *d* secret, must be sent *encrypted*

Secure Channel Pattern: Abstract specification

The abstract specification on the previous slide ...

- is a high-level system specification.
- is in form of a UML subsystem *C*.
- is for communication from a sender object to a receiver object.
- includes a class diagram with appropriate interfaces.
- is a simplified example with fixed participants *S* and *R* ...
  - which should mainly demonstrate the idea of stepwise development.
  - where authentication is out of scope.

Intended specification behaviour:

- The *Sender* object is supposed to accept a value in the variable *d* as an argument of the operation *send* and send it over the *<<encrypted>>* Internet link to the *Receiver* object, which delivers the value as a return value of the operation *receive*.

Secure Channel Pattern: Abstract specification

Specification characteristics:

Proposition 5.1: The subsystem *C* preserves the secrecy of the variable *d* from adversaries of type *A = default* with specified previous knowledge K^p_A, given inputs from $Data \setminus K^p_A$.

- Note that, intuitively, this proposition is obvious, because the adversary cannot read the channels.
- Proof is on the next slide.

Since *d'* is intended to have the same value as *d*, secrecy of *d'* follows from secrecy of *d* and integrity of *d'* wrt. the value in *d*.

Integrity is not within the scope but holds for both *d* and *d'* since the adversary cannot interfere with the protocol.

Secure Channel Pattern: Abstract specification

Proof of *proposition 5.1*:

- We have to show that for every expression *E* which is a value of *d* at any point, *C* preserves the secrecy of *E*.
- Since the adversary of type *default* cannot access any of the components or links in *C*, we have
  - $K_A(C) = K^0_A$ (because there is no read access)
  - *d* takes values only in $\boldsymbol{Exp} / K^0_A$ (because there is no write access)
- Thus for every expression *E* which is a value of *d* at any point, *C* preserves the secrecy of *E*, by definition of preservation of secrecy.

Secure Channel Pattern: Approach

- Define a secure channel abstraction.
- Define concrete secure channel (protocol).
- Show concrete secure channel simulates the abstraction.
  - Give conditions under which it is secure to substitute channel abstractions by concrete protocol.

Secure Channel Pattern: (Toy) Solution

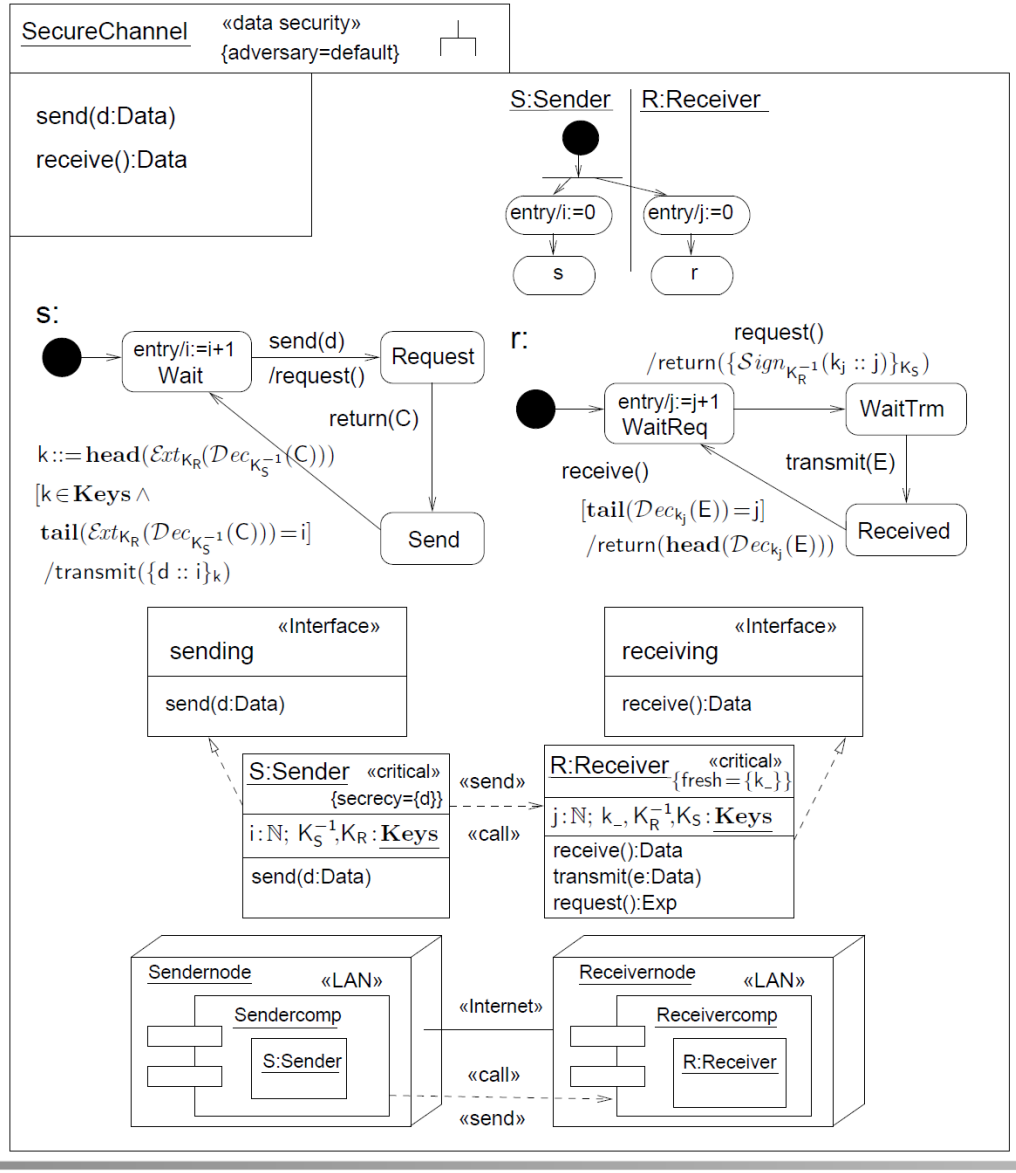
Well-known solution:

- Encrypt the traffic over the untrusted link using a key exchange protocol.

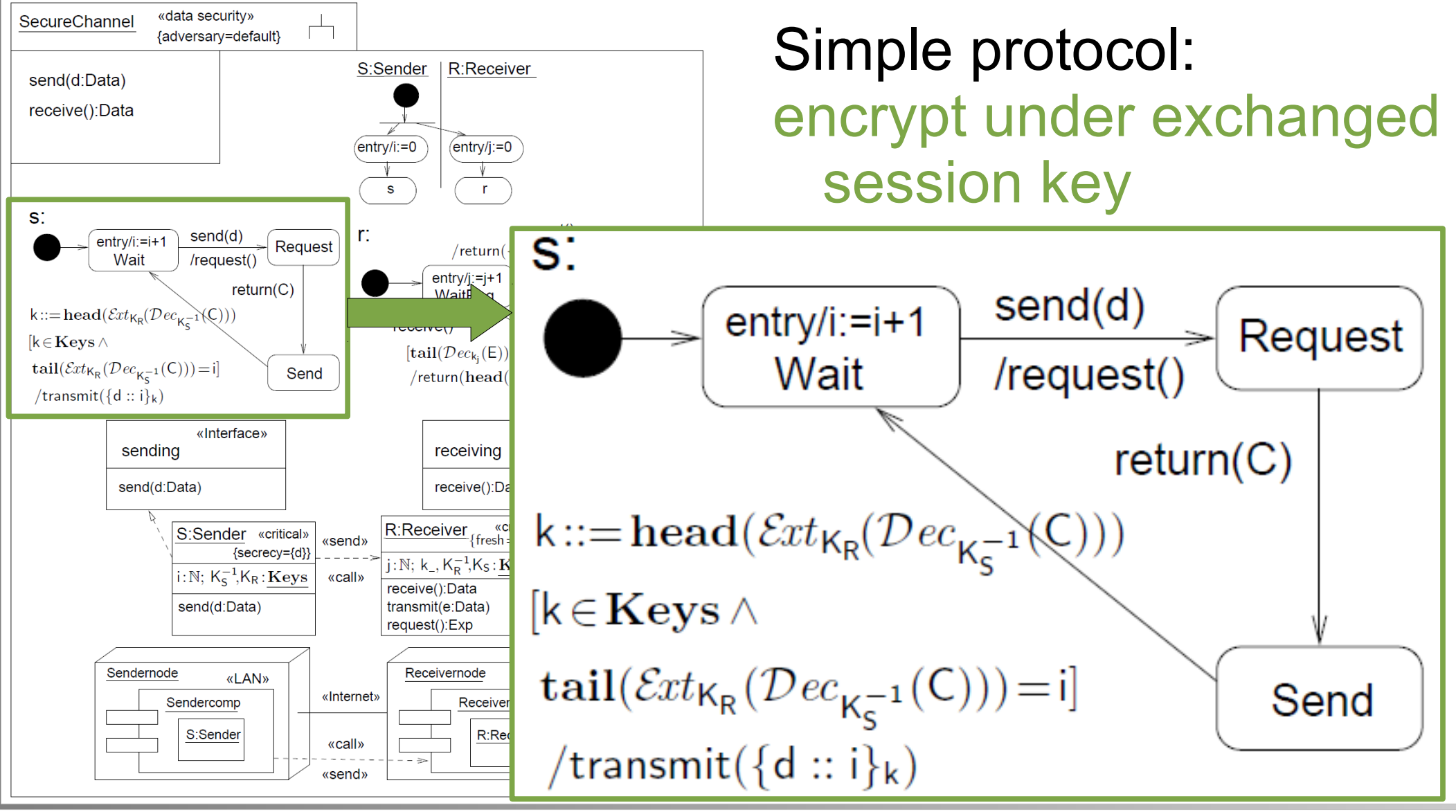
The Secure Channel Pattern could thus be formulated intuitively as follows:

- In a situation such as the one on the previous slides, one can implement the secure channel needed to enforce the security requirements using the following system.

Secure Channel Pattern: (Toy) Solution



Secure Channel Pattern: (Toy) Solution

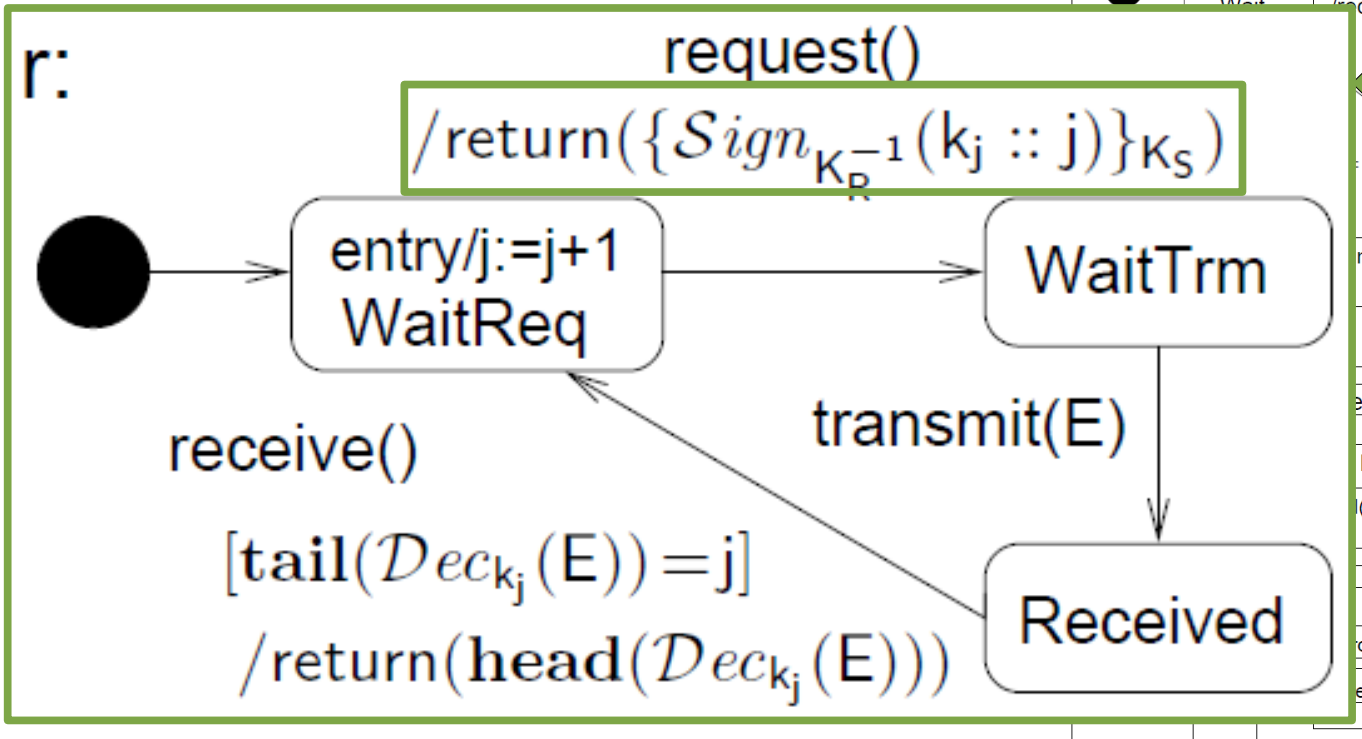


Simple protocol: encrypt under exchanged session key

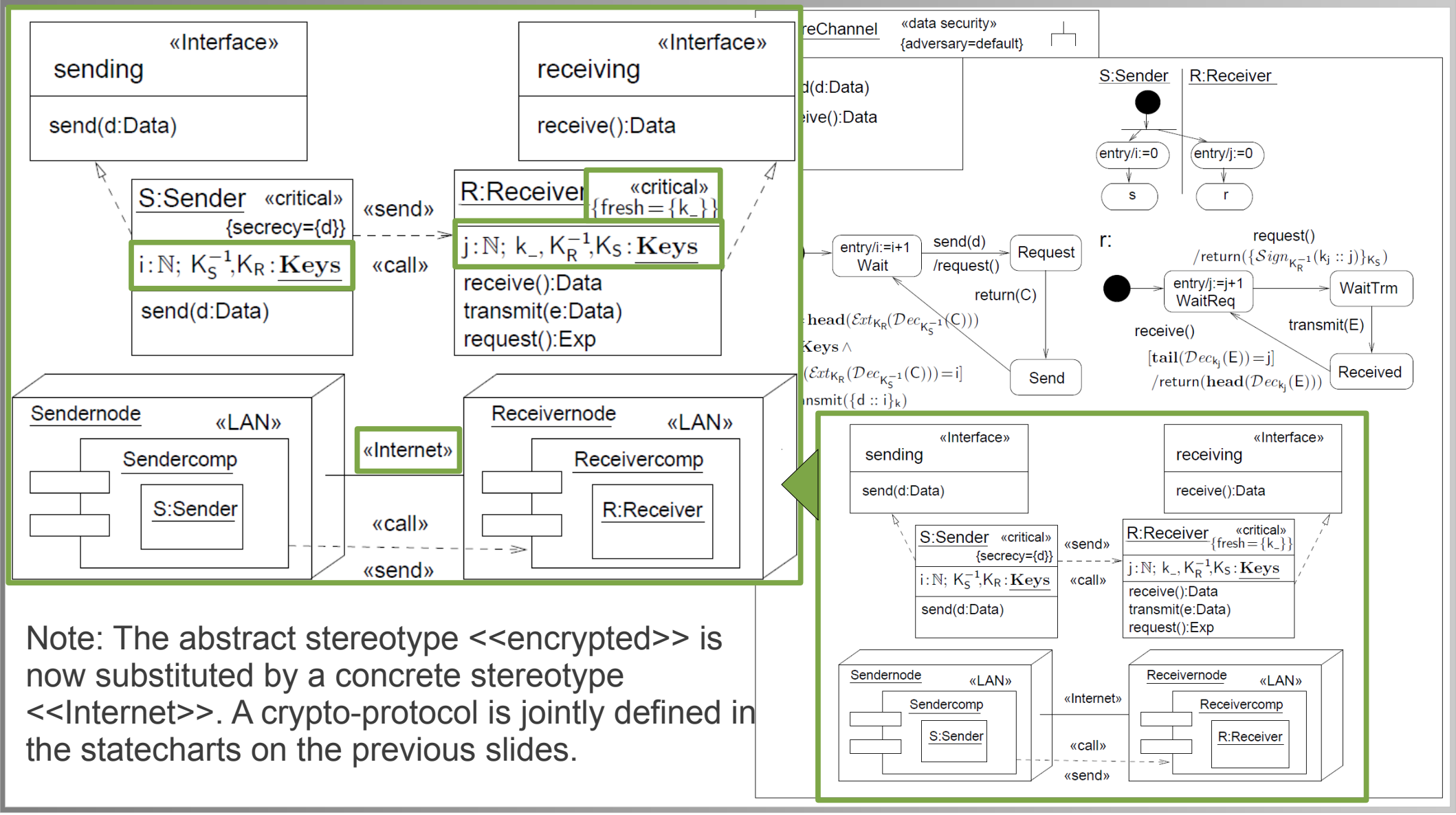
Secure Channel Pattern: (Toy) Solution

Simple protocol:

encrypt under exchanged session key



Secure Channel Pattern: (Toy) Solution



Note: The abstract stereotype <<encrypted>> is now substituted by a concrete stereotype <<Internet>>. A crypto-protocol is jointly defined in the statecharts on the previous slides.

Secure Channel Pattern: (Toy) Solution

- Since we only want to demonstrate the principle of developing a secure channel, we assume for simplicity that the sender and receiver already know each other´s public keys.
- The protocol then exchanges a symmetric session key using those public keys, since encryption under symmetric keys is more efficient.
- We assume that the secret keys belonging to the public ones are kept secure.
- The session keys are specified to be created freshly by the receiver before execution of the protocol, as stated by the tag *{fresh}*.

Secure Channel Pattern: (Toy) Solution

- The behaviour of the sender thus includes retrieving the signed and encrypted symmetric session key k_j from the receiver, checking the signature, and encrypting the data under the symmetric key.
- Encryption is done together with a sequence number *i*, to avoid replay.
- The receiver first gives out the key k_j with a signature and also with a sequence number *j*, and later decrypts the received data, checking the sequence number.

Secure Channel Pattern: Approach

- Define a secure channel abstraction.
- Define concrete secure channel (protocol).
- Show concrete secure channel simulates the abstraction.
  - Give conditions under which it is secure to substitute channel abstractions by concrete protocol.

Secure Channel Pattern: Concrete simulates abstraction

We show that the concrete secure channel *C'* is a refinement of *C* in the sense of the definition (repetition from slide deck 16):

> *T* is a *delayed black-box refinement* of *S* if every observable input/output behaviour of *T* differs from an input/output behaviour of *S* only in that delays may be introduced.

Proposition 5.2: The subsystem *C'* is a *delayed black-box refinement* of *C* in presence of adversaries of type *A = default* with

$$K^p_A \cap (\{K_S^{-1}, K_R^{-1}\} \cup \{k_n, \{x::n\}_{k_n} : x \in \mathbf{Exp} \wedge n \in \mathbb{N}\}) = \emptyset$$

and for which $Sign_{K_R^{-1}}(k' :: m) \in K^p_A$ implies $k' = k_m$ for all $m \in \mathbb{N}$ and $k \in \mathbf{Exp}$.

(SCP) Concrete simulates abstraction: Proof

Proof of *proposition 5.2*:

We have to show that for every adversary b of type A for the UMS $[[C']]$ there exists an adversary a of type A for the UMS $[[C]]$ such that the derived UML Machine $\mathbf{Exec}[[C']]_b$ is a delayed black-box refinement of the UML Machine $\mathbf{Exec}[[C]]_a$.

- Note that $K_A(C')$ is contained in the algebra generated by

  $K^0_A \cup \{\{Sign_{K_R^{-1}}(k_i::j)\}_{K_S}\}$ and the expressions $\{d::n\}_K$ for inputs d.
- The adversary can obtain no certificate $\{\{Sign_{K_R^{-1}}(k::j)\}_{K_S}\}$ for $k \neq k_j$, because the Receiver object only outputs the certificates $\{Sign_{K_R^{-1}}(k_j::j)\}_{K_S}$ (for $j \in \mathbb{N}$) to the Internet
- The sender outputs only messages of the form $\{d::n\}_k$ to the Internet, for inputs d and any $k \in \mathbf{Keys}$ for which a certificate $\{Sign_{K_R^{-1}}(k::n)\}_{K_S}$ has been received.

(SCP) Concrete simulates abstraction: Proof

- K must be K_n since no other certificate can be produced, since the key K_R^{-1} is never transmitted.
- Note also that $K^p_A = K^0_A$ since there are no components accessed by the adversary.
- The values that an adversary for C' may insert into the Internet link may only delay the behaviour of the two objects regarding $outQu_{C'}$ since the adversary has no other certificate signed with K_R^{-1} and does not have access to the key K_R^{-1} and because of the transaction numbers used.
- Any other value inserted is ignored by the two objects.
- For any adversary b for C' we can derive an adversary a for C by omitting insert and read commands such that the UML Machine $\mathbf{Exec}[[C']]_b$ is a delayed black-box refinement of the UML Machine $\mathbf{Exec}[[C]]_a$ since the outputs to $outQu_C$ (resp. $outQu_{C'}$) are stutter-equivalent.

(SCP) Concrete simulates abstraction: Proof

- The condition in the statement on the previous slides means that the previous adversary knowledge K^p_A may not contain
  - the secret keys K_S^{-1}, K_R^{-1} of the sender and the receiver,
  - the secret session keys k_n,
  - any encryptions of the form $\{x :: n\}_{k_n}$,
  - any signatures $Sign_{K_R^{-1}}(k' :: m)$ except for $k' = k_n$.
- Remember: K^p_A denotes the knowledge of the adversary before the start of the execution of the system, that is in this case, before the first iteration of the protocol.
- Thus the condition does not prevent the adversary from remembering information gained from early iterations of the protocol and use it in later iterations.

(SCP) Concrete simulates abstraction: Proof

- If the adversary knows the expression $\{x :: n\}_{k_n}$ before the execution,
  - which is different from the expression $\{y :: n\}_{k_n}$ which is sent out by S in the n th round of the protocol
  - the adversary could substitute $\{y :: n\}_{k_n}$ with $\{x :: n\}_{k_n}$ without being noticed which would destroy the integrity of the communication channel.
  - This means: C' would not be a refinement of C.
- Note that the sequence number n is necessary to enable the receiver to check that the right session key is used for decryption in the condition $\mathbf{tail}(Dec_{k_j}(E)) = j$, to prevent replay.

Proposition 5.3: The subsystem *C'* preserves the secrecy of the variable *d* from adversaries of type *A = default* with

$$K^p_A \cap (\{K_S^{-1}, K_R^{-1}\} \cup \{k_n, \{x::n\}_{k_n} : x \in \mathbf{Exp} \wedge n \in \mathbb{N}\}) = \emptyset$$

and for which $Sign_{K_R^{-1}}(k' :: m) \in K^p_A$ implies $k' = k_m$ for all $m \in \mathbb{N}$ and $k' \in \mathbf{Exp}$. (Proof on the next slide)

The specification fulfils the constraints of the stereotype *<<data security>>* with respect to the adversary type.

Proof of *proposition 5.3*:

- *C* preserves the secrecy of the variable *d* from *default* adversaries given inputs from $\mathbf{Data} \setminus K^p_A$. (see proposition 5.1)
- *C'* is a delayed black-box refinement of *C* given *default* adversaries. (see proposition 5.2)
- We can conclude that *C'* preserves the secrecy of the variable *d* from *default* adversaries with

  $$K^p_A \cap (\{K_{CA}^{-1}, K^{-1}\} \cup \{\{x::n\}_K : x \in \mathbf{Exp} \wedge n \in \mathbb{N}\}) = \emptyset$$

  and for which

  $Sign_{K_{CA}^{-1}}(R::k') \in K^p_A$ implies $K = k'$,

  given inputs from $\mathbf{Data} \setminus K^p_A$.

Summary

We presented the UML extension UMLsec for secure systems development.

- It is a UML profile which uses the standard UML extension mechanisms.
- Recurring security requirements are written as stereotypes.
- The associated constraints ensure the security requirements on the level of formal semantics, by referring to the threat scenario also given as a stereotype.
- Now we can evaluate UML specifications to indicate possible vulnerabilities.
- After that we can verify that the stated security requirements, if fulfilled, enforce a given security policy.

Summary

We indicated how one could use UMLsec to

- model security requirements.
- threat scenarios.
- security concepts.
- security mechanisms.
- security primitives.
- underlying physical security.
- security management.

These are the aspects which were argued to be required for a secure systems extension of UML.

Summary

We also saw how UMLsec could be used to encapsulate established rules on prudent security engineering

- by applying security patterns.
- to make them available to developers who are not security experts.

While UML was developed to model object-oriented systems, we can also use UML and UMLsec to analyse systems that are component-oriented by not making use of OO-specific features and make sure that the underlying assumptions, such as controlled access to data, are ensured.