Model-Based Security Engineering with UML

Jan Jürjens*

Dep. of Informatics, TU Munich, Germany
http://www4.in.tum.de/~juerjens

Abstract. Developing security-critical systems is difficult and there are many well-known examples of security weaknesses exploited in practice. Thus a sound methodology supporting secure systems development is urgently needed.

Our aim is to aid the difficult task of developing security-critical systems in a formally based approach using the notation of the Unified Modeling Language. We present the extension UMLsec of UML that allows one to express security-relevant information within the diagrams in a system specification. UMLsec is defined in form of a UML profile using the standard UML extension mechanisms. In particular, the associated constraints give criteria to evaluate the security aspects of a system design, by referring to a formal semantics of a simplified fragment of UML. We explain how these constraints can be formally verified against the dynamic behavior of the specification using automated theorem provers for first-order logic. This formal security verification can also be extended to C code generated from the specifications.

1 Introduction

Modern society and modern economies rely on infrastructures for communication, finance, energy distribution, and transportation. These infrastructures depend increasingly on networked information systems. Attacks against these systems can threaten the economical or even physical well-being of people and organizations. Due to the widespread interconnection of information systems, attacks can be waged anonymously and from a safe distance. Many security incidents have been reported, sometimes with potentially quite severe consequences.

Many problems with security-critical systems arise from the fact that their developers do not always have a strong background in computer security. This is problematic since in practice, security is compromised most often not by breaking mechanisms such as encryption or security protocols, but by exploiting weaknesses in the way they are being used. Security mechanisms cannot be "blindly" inserted into a security-critical system, but the overall system development must take security aspects into account.

* This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the author(s).

A. Aldini, R. Gorrieri, and F. Martinelli (Eds.): FOSAD 2004/2005, LNCS 3655, pp. 42–77, 2005.

Furthermore, sometimes security mechanisms (such as security protocols) cannot be used off-the-shelf, but have to be designed specifically to satisfy given requirements (see [GHJW03] for an example). Such mechanisms are notoriously hard to design correctly, even for experts, as many examples of protocols designed by experts that were later found to contain flaws show.

Enforcing security requirements is intrinsically subtle, because one has to take into account the interaction of the system with motivated adversaries that act independently. Risks are very hard to calculate because of the possibility to quickly distribute new information to exploit vulnerabilities, for example across the Internet.

Any support to aid secure systems development is thus dearly needed. In particular, it would be desirable to consider security aspects already in the design phase, before a system is actually implemented, since removing security flaws in the design phase, as opposed to patching fielded systems, saves cost and time, reduces security risks and increases customer confidence.

This has motivated a significant amount of successful research into using formal methods for secure systems development. However, part of the difficulty of security-critical systems development is that correctness is often in conflict with cost. Where thorough methods of system design pose high cost through personnel training and use, they are all too often avoided.

The Unified Modeling Language (UML, [UML01], the de facto industry-standard in object-oriented modeling) offers an interesting opportunity for high-quality secure systems development that is feasible in an industrial context.

- As the de facto standard in industrial modeling, a large number of developers is trained in UML.
- Compared to previous notations with a user community of comparable size, UML is relatively precisely defined.

Also, because of its expressitivity and the formal foundation of the UML fragment under consideration, UML gives an interesting theoretical basis for research into open problems in the foundations of security, such as the composability and consistency of the various formalized security requirements. Because our underlying formal system model is largely independent from UML specifics, it provide a suitable platform for such investigations also independently from UML.

To exploit this opportunity, however, some challenges remain which include the following:

- Extending the UML to be able to conveniently express security requirements within a UML specification.
- Defining a formal execution semantics for a sufficient simplified fragment of UML as a basis for the formalization of behavioral security requirements.
- Providing automated tool-support for a formal security verification of UML specifications against the security requirements.

The present work reports on research towards overcoming these challenges.

To support using UML for secure systems development, we define the extension UMLsec of the UML. Various recurring security requirements (such as secrecy, integrity, authenticity and others), as well as assumptions on the security of the system environment, are offered as stereotypes and tags by the UMLsec definition. These can be included in UML diagrams firstly to keep track of the information. Using the associated constraints that refer to a formal semantics of the used simplified fragment of UML, the properties can be used to evaluate diagrams of various kinds and to indicate possible vulnerabilities. One can thus verify that the stated security requirements, if fulfilled, enforce a given security policy. One can also ensure that the requirements are actually met by the given UML specification of the system. This way we can encapsulate knowledge on prudent security engineering and thereby make it available to developers which may not be specialized in security. One can also go further by checking whether the constraints associated with the UMLsec stereotypes are fulfilled in a given specification, if desired by performing an automated formal security verification using automated theorem provers for first order logic or model-checkers.

Due to space restrictions, we can only present a partial overview on the work. A complete acccount, with many more examples and industrial applications, and the necessary background on distributed system security, can be found in [Jür04].

We explain how to formally evaluate UML specifications for security requirements in Sect. 2. We introduce a fragment of the UMLsec notation in Sect. 3 and explain the various stereotypes with examples. We describe how to use automated theorem provers for first-order logic to verify UML specifications against seurity requirements in Sect. 4. We point to further work applying these analyses to the source-code level in Sect. 5. We report on an industrial application to biometric authentication systems in Sect. 6.

2 Security Evaluation of UML Diagrams

A UMLsec diagram is essentially a UML diagram where security properties and requirements are inserted as stereotypes with tags and constraints, although certain restrictions apply to enable automated formal verification. UML[1] offers three main "light-weight" language extension mechanisms: stereotypes, tagged values, and constraints [UML01]. Stereotypes define new types of modeling elements extending the semantics of existing types or classes in the UML metamodel. Their notation consists of the name of the stereotype written in double angle brackets « », attached to the extended model element. This model element is then interpreted according to the meaning ascribed to the stereotype. One way of explicitly defining a property is by attaching a tagged value to a model element. A tagged value is a name-value pair, where the name is referred to as the *tag*. The corresponding notation is $\{tag = value\}$ with the tag name *tag* and a corresponding *value* to be assigned to the tag. If the value is of type Boolean, one

[1] In the following, we consider the UML version 1.5 current at the time of writing; the transition to the upcoming version 2.0 only has a limited impact on the things we discuss here.

usually omits $\{tag = false\}$, and writes $\{tag\}$ instead of $\{tag = true\}$. Another way of adding information to a model element is by attaching logical *constraints* to refine its semantics (for example written in first-order predicate logic).

To construct an extension of the UML one collects the relevant definitions of stereotypes, tagged values, and constraints into a so-called profile [UML01]. For UMLsec, we give validation rules that evaluate a model with respect to listed security requirements. Many security requirements are formulated regarding the behavior of a system in interaction with its environment (in particular, with potential adversaries). To verify these requirements, we use the formal semantics defined in Sect. 2.1.

2.1 Outline of Formal Semantics

For some of the constraints used to define the UMLsec extensions we need to refer to a precisely defined semantics of behavioral aspects, because verifying whether they hold for a given UML model may be mathematically non-trivial. Firstly, the semantics is used to define these constraints in a mathematically precise way. Secondly, we have developed mechanical tool support for analyzing UML specifications for security requirements using model-checkers and automated theorem provers for first-order logic [JS04, JS05, Jür05a]. For this, a precise definition of the meaning of the specifications is necessary. For security analysis, the security-relevant information from the security-oriented stereotypes is then incorporated (cf. Sect. 2.3).

Because of space restrictions, we cannot recall our formal semantics here completely. Instead, we define precisely and explain the interfaces of this semantics that we need here to define the UMLsec profile. More details on the formal semantics of a simplified fragment of UML and on previous and related work in this area can be found in [Jür02, Jür04]. The semantics is defined formally using so-called *UML Machines*, which is an extension of Mealy automata with UML-type communication mechanisms. It includes the following kinds of diagrams:

Class diagrams define the static class structure of the system: classes with attributes, operations, and signals and relationships between classes. On the instance level, the corresponding diagrams are called *object diagrams*.

Statechart diagrams (or *state diagrams*) give the dynamic behavior of an individual object or component: events may cause a change in state or an execution of actions.

Sequence diagrams describe interaction between objects or system components via message exchange.

Activity diagrams specify the control flow between several components within the system, usually at a higher degree of abstraction than statecharts and sequence diagrams. They can be used to put objects or components in the context of overall system behavior or to explain use cases in more detail.

Deployment diagrams describe the physical layer on which the system is to be implemented.

Subsystems (a certain kind of *packages*) integrate the information between the different kinds of diagrams and between different parts of the system specification.

There is another kind of diagrams, the use case diagrams, which describe typical interactions between a user and a computer system. They are often used in an informal way for negotiation with a customer before a system is designed. We will not use it in the following. Additionally to sequence diagrams, there are *collaboration diagrams*, which present similar information. Also, there are *component diagrams*, presenting part of the information contained in deployment diagrams.

The used fragment of UML is simplified to keep automated formal verification that is necessary for some of the more subtle security requirements feasible. Note that in our approach we identify system objects with UML objects, which is suitable for our purposes. Also, as with practically all analysis methods, also in the real-time setting [Wat02], we are mainly concerned with instance-based models. Although, simplified, our choice of a subset of UML is reasonable for our needs, as we have demonstrated in several industrial case-studies (some of which are documented in [Jür04]).

The formal semantics for subsystems incorporates the formal semantics of the diagrams contained in a subsystem. It

- models actions and internal activities explicitly (rather than treating them as atomic given events), in particular the operations and the parameters employed in them,
- provides passing of messages with their parameters between objects or components specified in different diagrams, including a dispatching mechanism for events and the handling of actions, and thus
- allows in principle whole specification documents to be based on a formal foundation.

In particular, we can compose subsystems by including them into other subsystems.

Objects, and more generally system components, can communicate by exchanging messages. These consist of the message name, and possibly arguments to the message (which will be assumed to be elements of the set **Exp** defined in Sect. 2.2). Message names may be prefixed with object or subsystem instance names. Each object or component may receive messages received in an input queue and release messages to an output queue.

In our model, every object or subsystem instance O has associated multi-sets inQu_O and outQu_O (*event queues*). Our formal semantics models sending a message $msg = op(exp_1, \ldots, exp_n) \in \mathbf{Events}$ from an object or subsystem instance S to an object or subsystem instance R as follows:

(1) S places the message $R.msg$ into its multi-set outQu_S.
(2) A scheduler distributes the messages from out-queues to the intended in-queues (while removing the message head); in particular, $R.msg$ is removed from outQu_S and msg added to inQu_R.
(3) R removes msg from its in-queue and processes its content.

In the case of operation calls, we also need to keep track of the sender to allow sending return signals. This way of modeling communication allows for a very flexible treatment; for example, we can modify the behavior of the scheduler to take account of knowledge on the underlying communication layer (for example regarding security issues, see Sect. 2.3).

At the level of single objects, behavior is modeled using statecharts, or (in special cases such as protocols) possibly as using sequence diagrams. The internal activities contained at states of these statecharts can again be defined each as a statechart, or alternatively, they can be defined directly using UML Machines.

Using subsystems, one can then define the behavior of a system component C by including the behavior of each of the objects or components directly contained in C, and by including an activity diagram that coordinates the respective activities of the various components and objects.

Thus for each object or component C of a given system, our semantics defines a function $[\![C]\!]()$ which

- takes a multi-set I of input messages and a component state S and
- outputs a set $[\![C]\!](I, S)$ of pairs (O, T) where O is a multi-set of output messages and T the new component state (it is a *set* of pairs because of the non-determinism that may arise)

together with an *initial state* S_0 of the component.

Specifically, the behavioral semantics $[\![D]\!]()$ of a statechart diagram D models the run-to-completion semantics of UML statecharts. As a special case, this gives us the semantics for activity diagrams. Any sequence diagram $\mathcal{S}$ gives us the behavior $[\![\mathcal{S}.C]\!]()$ of each contained component C.

Subsystems group together diagrams describing different parts of a system: a system component $\mathcal{C}$ given by a subsystem $\mathcal{S}$ may contain subcomponents $\mathcal{C}_1, \ldots, \mathcal{C}_n$. The behavioral interpretation $[\![\mathcal{S}]\!]()$ of $\mathcal{S}$ is defined as follows:

(1) It takes a multi-set of input events.
(2) The events are distributed from the input multi-set and the link queues connecting the subcomponents and given as arguments to the functions defining the behavior of the intended recipients in $\mathcal{S}$.
(3) The output messages from these functions are distributed to the link queues of the links connecting the sender of a message to the receiver, or given as the output from $[\![\mathcal{S}]\!]()$ when the receiver is not part of $\mathcal{S}$.

When performing security analysis, after the last step, the adversary model may modify the contents of the link queues in a certain way explained in Sect. 2.3.

2.2 Modeling Cryptography

We introduce some sets to be used in modeling cryptographic data in a UML specification and its security analysis.

We assume a set **Keys** with a partial injective map $(\ )^{-1} : \mathbf{Keys} \to \mathbf{Keys}$. The elements in its domain (which may be public) can be used for encryption

and for verifying signatures, those in its range (usually assumed to be secret) for decryption and signing. We assume that every key is either an encryption or decryption key, or both: Any key k satisfying $k^{-1} = k$ is called *symmetric*, the others are called *asymmetric*. We assume that the numbers of symmetric and asymmetric keys are both infinite. We fix infinite sets **Var** of *variables* and **Data** of *data values*. We assume that **Keys**, **Var**, and **Data** are mutually disjoint and that **ASMNames**∪**MsgNm** ⊆ **Data**. **Data** may also include *nonces* and other secrets.

The *algebra of cryptographic expressions* **Exp** is the quotient of the term algebra generated from the set **Var** ∪ **Keys** ∪ **Data** with the operations

- $\_ :: \_$ (concatenation),
- **head**(_) and **tail**(_),
- $\{\_\}\_$ (encryption)
- $\mathcal{D}ec\_(\_)$ (decryption)
- $\mathcal{S}ign\_(\_)$ (signing)
- $\mathcal{E}xt\_(\_)$ (extracting from signature)
- $\mathcal{H}ash(\_)$ (hashing)

by factoring out the equations

- $\mathcal{D}ec_{K^{-1}}(\{E\}_K) = E$ (for $K \in$ **Keys**),
- $\mathcal{E}xt_K(\mathcal{S}ign_{K^{-1}}(E)) = E$ (for $K \in$ **Keys**),
- and the usual laws regarding concatenation, **head**(), and **tail**():
  - $(E_1 :: E_2) :: E_3 = E_1 :: (E_2 :: E_3)$ for all $E_1, E_2, E_3 \in$ **Exp**,
  - **head**$(E_1 :: E_2) = E_1$ and **tail**$(E_1 :: E_2) =$ **tail**(E_2) for all expressions $E_1, E_2 \in$ **Exp** such that there exist no E, E' with $E_1 = E :: E'$.

We write **fst**$(E) \stackrel{\text{def}}{=}$ **head**(E), **snd**$(E) \stackrel{\text{def}}{=}$ **head**(**tail**(E)), and **thd**$(E) \stackrel{\text{def}}{=}$ **head**(**tail**(**tail**(E))) for each $E \in$ **Exp**.

This symbolic model for cryptographic operations implies that we assume cryptography to be perfect, in the sense that an adversary cannot "guess" an encrypted value without knowing the decryption key. Also, we assume that one can detect whether an attempted decryption is successful. See for example [AJ01] for a formal discussion of these assumptions.

Based on this formalization of cryptographical operations, important conditions on security-critical data (such as freshness, secrecy, integrity) can then be formulated at the level of UML diagrams in a mathematically precise way (see Sect. 3).

In the following, we will often consider *subalgebras* of **Exp**. These are subsets of **Exp** which are closed under the operations used to define **Exp** (such as concatenation, encryption, decryption etc.). For each subset E of **Exp** there exists a unique smallest (wrt. subset inclusion) **Exp**-subalgebra containing E, which we call **Exp**-*subalgebra generated by* E. Intuitively, it can be constructed from E by iteratively adding all elements in **Exp** reachable by applying the operations used to define **Exp** above. It can be seen as the knowledge one can obtain from a given set E of data by iteratively applying publicly available

operations to it (such as concatenation and encryption etc.) and will be used to model the knowledge an attacker may gain from a set E of data obtained for example by eavesdropping on Internet connections.

2.3 Security Analysis of UML Diagrams

Our modular UML semantics allows a rather natural modeling of potential adversary behavior. We can model specific types of adversaries that can attack different parts of the system in a specified way. For example, an attacker of type *insider* may be able to intercept the communication links in a company-wide local area network. We model the actual behavior of the adversary by defining a class of UML Machines that can access the communication links of the system in a specified way. To evaluate the security of the system with respect to the given type of adversary, we consider the joint execution of the system with any UML Machine in this class. This way of reasoning allows an intuitive formulation of many security properties. Since the actual verification is rather indirect this way, we also give alternative intrinsic ways of defining security properties below, which are more manageable, and show that they are equivalent to the earlier ones.

Thus for a security analysis of a given UMLsec subsystem specification $\mathcal{S}$, we need to model potential adversary behavior. We model specific types of adversaries that can attack different parts of the system in a specified way. For this we assume a function $\mathsf{Threats}_A(s)$ which takes an *adversary type* A and a stereotype s and returns a subset of $\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}, \mathsf{access}\}$ (*abstract threats*). These functions arise from the specification of the physical layer of the system under consideration using deployment diagrams, as explained in Sect. 3. For a link l in a deployment diagram in $\mathcal{S}$, we then define the set $\mathsf{threats}^{\mathcal{S}}_A(l)$ of *concrete threats* to be the smallest set satisfying the following conditions:

If each node n that l is contained in[2] carries a stereotype s_n with $\mathsf{access} \in \mathsf{Threats}_A(s_n)$ then:

- If l carries a stereotype s with $\mathsf{delete} \in \mathsf{Threats}_A(s)$ then $\mathsf{delete} \in \mathsf{threats}^{\mathcal{S}}_A(l)$.
- If l carries a stereotype s with $\mathsf{insert} \in \mathsf{Threats}_A(s)$ then $\mathsf{insert} \in \mathsf{threats}^{\mathcal{S}}_A(l)$.
- If l carries a stereotype s with $\mathsf{read} \in \mathsf{Threats}_A(s)$ then $\mathsf{read} \in \mathsf{threats}^{\mathcal{S}}_A(l)$.
- If l is connected to a node that carries a stereotype t with $\mathsf{access} \in \mathsf{Threats}_A(t)$ then $\{\mathsf{delete}, \mathsf{insert}, \mathsf{read}\} \subseteq \mathsf{threats}^{\mathcal{S}}_A(l)$.

The idea is that $\mathsf{threats}^{\mathcal{A}}_A(x)$ specifies the *threat scenario* against a component or link x in the UML Machine System $\mathcal{A}$ that is associated with an adversary type A. On the one hand, the threat scenario determines, which data the adversary can obtain by *accessing* components, on the other hand, it determines, which actions the adversary is permitted by the threat scenario to apply to the concerned links. delete means that the adversary may delete the messages in the

[2] Note that nodes and subsystems may be nested one in another.

corresponding link queue, read allows him to read the messages in the link queue, and insert allows him to insert messages in the link queue.

Then we model the actual behavior of an adversary of type A as a *type A adversary machine*. This is a state machine which has the following data:

- a control state $\mathsf{control} \in \mathsf{State}$,
- a set of *current adversary knowledge* $\mathcal{K} \subseteq \mathbf{Exp}$, and
- for each possible control state $c \in \mathsf{State}$ and set of knowledge $K \subseteq \mathbf{Exp}$, we have
  - a set $\mathsf{Delete}_{c,K}$ which may contain the name of any link l with $\mathsf{delete} \in \mathsf{threats}_A^{\mathcal{S}}(l)$
  - a set $\mathsf{Insert}_{c,K}$ which may contain any pair (l, E) where l is the name of a link with $\mathsf{insert} \in \mathsf{threats}_A^{\mathcal{S}}(l)$, and $E \in \mathcal{K}$, and
  - a set $\mathsf{newState}_{c,k} \subseteq \mathsf{State}$ of states.

The machine is executed from a specified initial state $\mathsf{control} := \mathsf{control}_0$ with a specified *initial knowledge* $\mathcal{K} := \mathcal{K}_0$ iteratively, where each iteration proceeds according to the following steps:

(1) The contents of all link queues belonging to a link l with $\mathsf{read} \in \mathsf{threats}_A^{\mathcal{S}}(l)$ are added to $\mathcal{K}$.
(2) The content of any link queue belonging to a link $l \in \mathsf{Delete}_{\mathsf{control},\mathcal{K}}$ is mapped to $\emptyset$.
(3) The content of any link queue belonging to a link l is enlarged with all expressions E where $(l, E) \in \mathsf{Insert}_{\mathsf{control},\mathcal{K}}$.
(4) The next control state is chosen non-deterministically from the set $\mathsf{newState}_{\mathsf{control},\mathcal{K}}$.

The set $\mathcal{K}_0$ of initial knowledge contains all data values v given in the UML specification under consideration for which each node n containing v carries a stereotype s_n with $\mathsf{access} \in \mathsf{Threats}_A(s_n)$. In a given situation, $\mathcal{K}_0$ may also be specified to contain additional data (for example, public encryption keys).

Note that an adversary A able to remove all values sent over the link l (that it, $\mathsf{delete}_l \in \mathsf{threats}_A^{\mathcal{S}}(l)$) may not be able to selectively remove a value e with known meaning from l: For example, the messages sent over the Internet within a virtual private network are encrypted. Thus, an adversary who is unable to break the encryption may be able to delete all messages undiscrimatorily, but not a single message whose meaning would be known to him.

To evaluate the security of the system with respect to the given type of adversary, we then define the *execution of the subsystem $\mathcal{S}$ in presence of an adversary of type A* to be the function $[\![\mathcal{S}]\!]_A()$ defined from $[\![\mathcal{S}]\!]()$ by applying the modifications from the adversary machine to the link queues as a fourth step in the definition of $[\![\mathcal{S}]\!]()$ as follows:

(4) The type A adversary machine is applied to the link queues as detailed above.

Thus after each iteration of the system execution, the adversary may nondeterministically change the contents of link queues in a way depending on the level of physical security as described in the deployment diagram (see Sect. 3).

There are results which simplify the analysis of the adversary behavior defined above, which are useful for developing mechanical tool support, for example to check whether the security properties secrecy and integrity (see below) are provided by a given specification. These are beyond the scope of the current paper and can be found in [Jür04].

One possibility to specify security requirements is to define an idealized system model where the required security property evidently holds (for example, because all links and components are guaranteed to be secure by the physical layer specified in the deployment diagram), and to prove that the system model under consideration is behaviorally equivalent to the idealized one, using a notion of behavioral equivalence of UML models. This is explained in detail in [Jür04].

In the following subsection, we consider alternative ways of specifying the important security properties secrecy and integrity which do not require one to explicitly construct such an idealized system and which are used in the remaining parts of this paper.

2.4 Important Security Properties

The formal definitions of the two main security properties secrecy and integrity considered in this section follow the standard approach of [DY83] and are defined in an intuitive way by incorporating the attacker model.

Secrecy. The formalization of secrecy used in the following relies on the idea that a process specification preserves the secrecy of a piece of data d if the process never sends out any information from which d could be derived, even in interaction with an adversary. More precisely, d is leaked if there is an adversary of the type arising from the given threat scenario that does not initially know d and an input sequence to the system such that after the execution of the system given the input in presence of the adversary, the adversary knows d (where "knowledge", "execution" etc. have to be formalized). Otherwise, d is said to be kept secret.

Thus we come to the following definition.

Definition 1. *We say that a subsystem* $\mathcal{S}$ preserves the secrecy *of an expression* E *from adversaries of type* A *if* E *never appears in the knowledge set* $\mathcal{K}$ *of* A *during execution of* $[\![\mathcal{S}]\!]_A()$.

This definition is especially convenient to verify if one can give an upper bound for the set of knowledge $\mathcal{K}$, which is often possible when the security-relevant part of the specification of the system $\mathcal{S}$ is given as a sequence of command schemata of the form *await event e – check condition g – output event e'* (for example when using UML sequence diagrams or statecharts for specifying security protocols, see Sect. 3).

Note that this formalization of secrecy is relatively "coarse" in that it may not prevent implicit information flow, but it is comparatively easy to verify and seems to be sufficient in practice [Aba00].

Examples

- The system that sends the expression $\{m\}_K :: K \in \mathbf{Exp}$ over an unprotected Internet link does not preserve the secrecy of m or K against attackers eavesdropping on the Internet, but the system that sends $\{m\}_K$ (and nothing else) does, assuming that it preserves the secrecy of K against attackers eavesdropping on the Internet.
- The system that receives a key K encrypted with its public key over a dedicated communication link and sends back $\{m\}_K$ over the link does not preserve the secrecy of m against attackers eavesdropping on and inserting messages on the link, but does so against attackers that cannot insert messages on the link.

Integrity. The property integrity can be formalized similarly: If during the execution of the system, a system variable gets assigned a value intially only known to the adversary, then the adversary must have caused this variable to contain the value. In that sense the integrity of the variable is violated. (Note that with this definition, integrity is also viewed as violated if the adversary as an honest participant in the interaction is able to change the value, so the definition may have to be adapted in certain circumstances; this is, however, typical for formalizations of security properties.) Thus we say that a system preserves the integrity of a variable v if there is no adversary A such that at some point during the execution of the system with A, v has a value i_0 that is initially known only to A.

Definition 2. *We say that a subsystem* $\mathcal{S}$ preserves the integrity *of an attribute* a *from adversaries of type* A *with initial knowledge* $\mathcal{K}_0$ *if during execution of* $[\![\mathcal{S}]\!]_A()$*, the attribute* a *never takes on a value appearing in* $\mathcal{K}_0$ *but not in the specification* $\mathcal{S}$*.*

The idea of this definition is that $\mathcal{S}$ preserves the integrity of a if no adversary can make a take on a value initially only known to him, in interaction with $\mathcal{A}$. Intuitively, it is the "opposite" of secrecy, in the sense that secrecy prevents the flow of information from protected sources to untrusted recipients, while integrity prevents the flow of information in the other direction. Again, it is a relatively simple definition, which may however not prevent implicit flows of information.

Secure Information Flow. We explain an alternative way of specifying secrecy and integrity like requirements, which gives protection also against partial flow of information, but can be more difficult to deal with, especially when handling with encryption (for which further refinements of the notion are possible, but not needed in the following).

Given a set of messages H and a sequence $\boldsymbol{m}$ of event multi-sets, we write

- $\boldsymbol{m}^H$ for the sequence of event multi-sets derived from those in $\boldsymbol{m}$ by deleting all events the message names of which are *not* in H, and
- $\boldsymbol{m}_H$ for the sequence of event multi-sets derived from those in $\boldsymbol{m}$ by deleting all events the message names of which *are* in H.

Definition 3. *Given a subsystem $\mathcal{S}$ and a set of* high *messages H, we say that*

- *A* prevents down-flow *with respect to H if for any two sequences $\boldsymbol{i}, \boldsymbol{j}$ of event multi-sets and any two output sequences $\boldsymbol{o} \in [\![\mathcal{S}]\!]_A(\boldsymbol{i})$ and $\boldsymbol{p} \in [\![\mathcal{S}]\!]_A(\boldsymbol{j})$, $\boldsymbol{i}_H = \boldsymbol{j}_H$ implies $\boldsymbol{o}_H = \boldsymbol{p}_H$ and*
- *A* prevents up-flow *with respect to H if for any two sequences $\boldsymbol{i}, \boldsymbol{j}$ of event multi-sets and any two output sequences $\boldsymbol{o} \in [\![\mathcal{S}]\!]_A(\boldsymbol{i})$ and $\boldsymbol{p} \in [\![\mathcal{S}]\!]_A(\boldsymbol{j})$, $\boldsymbol{i}^H = \boldsymbol{j}^H$ implies $\boldsymbol{o}^H = \boldsymbol{p}^H$ and*

Intuitively, to prevent down-flow means that outputting a *non-high* (or *low*) message does not depend on *high* inputs (this can be seen as a secrecy requirement for messages marked as high). Conversely, to prevent up-flow means that outputting a *high* value does not depend on *low* inputs (this can be seen as an integrity requirement for messages marked as high).

This notion of *secure information flow* is a generalization of the original notion of noninterference for deterministic systems in [GM82] to system models that are non-deterministic because of underspecification, see [Jür04] for a more detailed discussion.

3 The UMLsec Extension

In Fig. 1 we give some of the stereotypes from UMLsec, together with their tags and constraints. For space reasons, we can only recall part of the UMLsec notation; a complete account can be found in [Jür04]. The constraints are only referred to in the table and formulated (in plain mathematical language) and explained in detail in the remainder of the section. Fig. 2 gives the corresponding tags (which are all DataTags). Note that some of the stereotypes on subsystems refer to stereotypes on model elements contained in the subsystems. For example, the constraint of the «data security» stereotype refers to contained objects stereotyped as «critical» (which in turn have tags {secrecy}). The relations between the elements of the tables are explained below in detail.

We explain the stereotypes and tags given in Figures 1 and 2. The constraints are parameterized over the adversary type with respect to which the security requirements should hold; we thus fix an adversary type A to be used in the following. By their nature, some of the constraints can be enforced at the level of abstract syntax (such as «secure links»), while others refer to the formal definitions in Sect. 2.3 (such as «data security»). Note that even checking the latter can be mechanized given appropriate tool-support, as explained in Sect. 4.

We give short examples for usage of the stereotypes. To keep the presentation concise, we sometimes give only those fragments of (instances of) subsystems

Stereotype	Base Class	Tags	Constraints	Description
Internet	link			Internet connection
encrypted	link			encrypted connection
LAN	link,node			LAN connection
wire	link			wire
smart card	node			smart card node
POS device	node			POS device
issuer node	node			issuer node
secure links	subsystem		dependency security matched by links	enforces secure communication links
secrecy	dependency			assumes secrecy
integrity	dependency			assumes integrity
high	dependency			high sensitivity
secure dependency	subsystem		«call», «send» respect data security	structural interaction data security
critical	object, subsystem	secrecy, integrity, high fresh		critical object
no down-flow	subsystem		prevents down-flow	information flow
no up-flow	subsystem		prevents up-flow	information flow
data security	subsystem		provides secrecy, integrity, freshness	basic datasec requirements
fair exchange	subsystem	start,stop	after start eventually reach stop	enforce fair exchange
provable	subsystem	action, cert	action is non-deniable	non-repudiation requirement
guarded access	subsystem		guarded objects accessed through guards	access control using guard objects
guarded	object	guard		guarded object

Fig. 1. UMLsec stereotypes

that are essential to the stereotype in question. Also, we omit presenting the formal semantics and proving the stated properties formally, since the examples are just for illustration.

Internet, encrypted, LAN, wire, smart card, POS device, issuer node These stereotypes on links (resp. nodes) in deployment diagrams denote the corresponding requirements on communication links (resp. system nodes). We require that each link or node carries at most one of these stereotypes. For each adversary type A, we have a function $\mathsf{Threats}_A(s)$ from each stereotype

$$s \in \{\text{«}\,\mathsf{wire}\,\text{»}, \text{«}\,\mathsf{encrypted}\,\text{»}, \text{«}\,\mathsf{LAN}\,\text{»}, \text{«}\,\mathsf{smart\ card}\,\text{»},\\ \text{«}\,\mathsf{POS\ device}\,\text{»}, \text{«}\,\mathsf{issuer\ node}\,\text{»}, \text{«}\,\mathsf{Internet}\,\text{»}\}$$

to a set of strings $\mathsf{Threats}_A(s) \subseteq \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}, \mathsf{access}\}$ under the following conditions:

Tag	Stereotype	Type	Multipl.	Description
secrecy	critical	String	*	secrecy of data
integrity	critical	String	*	integrity of data
high	critical	String	*	high-level message
fresh	critical	String	*	fresh data
start	fair exchange	String	*	start states
stop	fair exchange	String	*	stop states
action	provable	String	*	provable action
cert	provable	String	*	certificate
guard	guarded	String	1	guard object

Fig. 2. UMLsec tags

- for a node stereotype s, we have $\mathsf{Threats}_A(s) \subseteq \{\mathsf{access}\}$ and
- for a link stereotype s, we have $\mathsf{Threats}_A(s) \subseteq \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$.

Thus $\mathsf{Threats}_A(s)$ specifies which kinds of actions an adversary of type A can apply to node or links stereotyped s. This way we can evaluate UML specifications using the approach explained in Sect. 2.1. We make use of this for the constraints of the remaining stereotypes of the profile.

Examples for threat sets associated with some common adversary types are given in Figures 3 and 4.

Figure 3 gives the *default* attacker, which represents an outsider adversary with modest capability. This kind of attacker is able to read and delete the messages on an Internet link and to insert messages. On an encrypted Internet link (for example a Virtual Private Network), the attacker can delete the messages (without knowing their encrypted content), but not to read the (plaintext) messages or to insert messages (that are encrypted with the right key). Of course, this assumes that the encryption is set up in a way such that the adversary does not get hold of the secret key. The default attacker is assumed not to have direct access to the Local Area Network (LAN) and therefore not be able to eavesdrop on those connections[3], nor on wires connecting security-critical devices (for example, a smart card reader and a display in a point-of-sale (POS) device). Also, smart cards are assumed to be tamper-resistant against default attackers (although they may not be against more sophisticated attackers [And01]). Also, the default attacker is not able to access POS devices or card issuer systems.

Figure 4 defines the *insider* attacker (in the context of an electronic purse system). As an insider, the attacker may access the encrypted Internet link (the assumption is that insiders know the corresponding key) and the local system components.

Secure links. This stereotype, which may label (instances of) subsystems, is used to ensure that security requirements on the communication are met by the physical layer. More precisely, the constraint enforces that for each dependency

[3] With more sophistication, even an external adversary may be able to access local connections, but this is assumed to be beyond "default" capabilities.

Stereotype	$\mathsf{Threats}_{default}()$
Internet	{delete, read, insert}
encrypted	{delete}
LAN	$\emptyset$
wire	$\emptyset$
smart card	$\emptyset$
POS device	$\emptyset$
issuer node	$\emptyset$

Fig. 3. Threats from the *default* attacker

Stereotype	$\mathsf{Threats}_{insider}()$
Internet	{delete, read, insert}
encrypted	{delete, read, insert}
LAN	{delete, read, insert}
wire	{delete, read, insert}
smart card	$\emptyset$
POS device	{access}
issuer node	{access}

Fig. 4. Threats from the insider attacker *card issuer*

d with stereotype $s \in \{\text{«}\,\mathsf{secrecy}\,\text{»}, \text{«}\,\mathsf{integrity}\,\text{»}, \text{«}\,\mathsf{high}\,\text{»}\}$ between subsystems or objects on different nodes n, m, we have a communication link l between n and m with stereotype t such that

- in the case of $s = \text{«}\,\mathsf{high}\,\text{»}$, we have $\mathsf{Threats}_A(t) = \emptyset$,
- in the case of $s = \text{«}\,\mathsf{secrecy}\,\text{»}$, we have $\mathsf{read} \notin \mathsf{Threats}_A(t)$, and
- in the case of $s = \text{«}\,\mathsf{integrity}\,\text{»}$, we have $\mathsf{insert} \notin \mathsf{Threats}_A(t)$.

Example. In Fig. 5, given the *default* adversary type, the constraint for the stereotype «secure links» is violated: The model does not provide communication secrecy against the *default* adversary, because the Internet communication link between web-server and client does not give the needed security level according to the $\mathsf{Threats}_{default}(Internet)$ scenario. Intuitively, the reason is that Internet connections do not provide secrecy against default adversaries. Technically, the constraint is violated, because the dependency carries the stereotype «secrecy», but for the stereotype «Internet» of corresponding link we have $\mathsf{read} \in \mathsf{Threats}_{default}(Internet)$.

Secrecy, Integrity, High. These stereotypes, which may label dependencies in static structure or component diagrams, denote dependencies that are supposed to provide the respective security requirement for the data that is sent along them as arguments or return values of operations or signals. These stereotypes are used in the constraint for the stereotype «secure links».

Secrecy. «call» or «send» dependencies in object or component diagrams stereotyped «secrecy» are supposed to provide secrecy for the data that is sent along

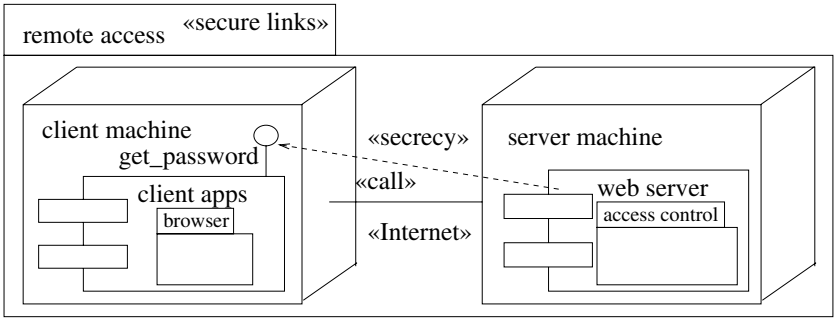

Fig. 5. Example *secure links* usage

them as arguments or return values of operations or signals. This stereotype is used in the constraint for the stereotype « secure links ».

Secure Dependency. This stereotype, used to label subsystems containing object diagrams or static structure diagrams, ensures that the « call » and « send » dependencies between objects or subsystems respect the security requirements on the data that may be communicated along them, as given by the tags {secrecy}, {integrity}, and {high} of the stereotype « critical ». More exactly, the constraint enforced by this stereotype is that if there is a « call » or « send » dependency from an object (or subsystem) C to an interface I of an object (or subsystem) D then the following conditions are fulfilled.

- For any message name n in I, n appears in the tag {secrecy} (resp. {integrity} resp. {high}) in C if and only if it does so in D.
- If a message name in I appears in the tag {secrecy} (resp. {integrity} resp. {high}) in C then the dependency is stereotyped « secrecy » (resp. « integrity » resp. « high »).

If the dependency goes directly to another object (or subsystem) without involving an interface, the same requirement applies to the trivial interface containing all messages of the server object.

Example. Figure 6 shows a key generation subsystem stereotyped with the requirement « secure dependency ». The given specification violates the constraint for this stereotype, since Random generator and the « call » dependency do not provide the security levels for random() required by Key generator. More precisely, the constraint is violated, because the message *random* is required to be of high level by Key generator (by the tag {high} in Key generator), but it is not guaranteed to be high level by Random generator (in fact there are no high messages in Random generator and so the tag {high} is missing).

Critical. This stereotype labels objects or subsystem instances containing data that is critical in some way, which is specified in more detail using the corresponding tags. These tags are {secrecy}, {integrity}, {fresh}, and {high}. The values

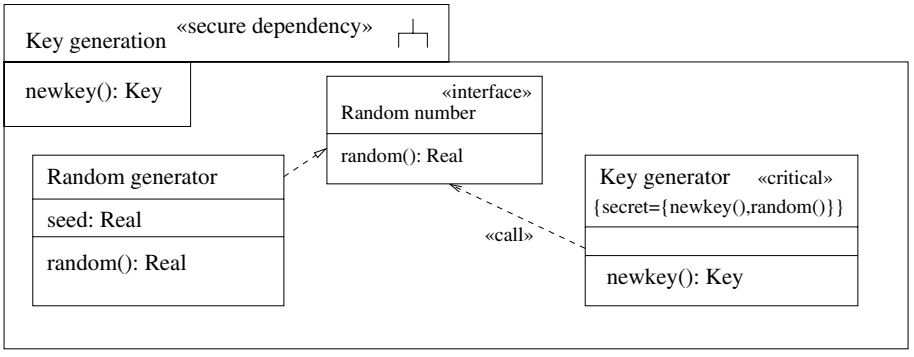

Fig. 6. Key generation subsystem

of the first two are the names of expressions or variables (that is, attributes or message arguments) of the current object the secrecy (resp. integrity) of which is supposed to be protected. The tag {fresh} has data that should be freshly generated as its value. These requirements are enforced by the constraint of the stereotype «data security» which labels (instances of) subsystems that contain «critical» objects (see there for an explanation). The tag {high} has the names of messages as values that are supposed to be protected with respect to secure information flow, as enforced by the stereotypes «no down − flow» and «no up − flow».

No Down-Flow. This stereotype of subsystems enforces secure information flow by making use of the associated tag {high}. According to the «no down − flow» constraint, the stereotyped subsystem prevents down-flow with respect to the messages and their return messages specified in {high}, as defined in Definition 3.

Example. The example in Fig. 7 shows a bank account data object that allows its secret balance to be read using the operation rb() (whose return value is also secret) and written using wb(x). If the balance is over 10000, the object is in a state ExtraService, otherwise in NoExtraService. The state of the object can be queried using the operation rx(). The data object is supposed to be prevented from indirectly leaking out any partial information about *high* data via non-high data, as specified by the stereotype «no down − flow». For example, in a situation where government agencies can request information about the existence of bank accounts of a given person, but not their balance, it may be important that the *type* of the account allows no conclusion about its balance. The given specification violates the constraint associated with «no down − flow», since partial information about the input of the high operation wb() is leaked out via the return value of the non-high operation rx(). To see how the underlying formalism captures the security flaw using Definition 3, it is sufficient to exhibit sequences $\boldsymbol{i}, \boldsymbol{j}$ of event multi-sets and output sequences $\boldsymbol{o} \in [\![A]\!](\boldsymbol{i})$ and $\boldsymbol{p} \in [\![A]\!](\boldsymbol{j})$ of the UML Machine A giving the behavior of the statechart, with $\boldsymbol{i}_H = \boldsymbol{j}_H$ and $\boldsymbol{o}_H \neq \boldsymbol{p}_H$, where H is the set

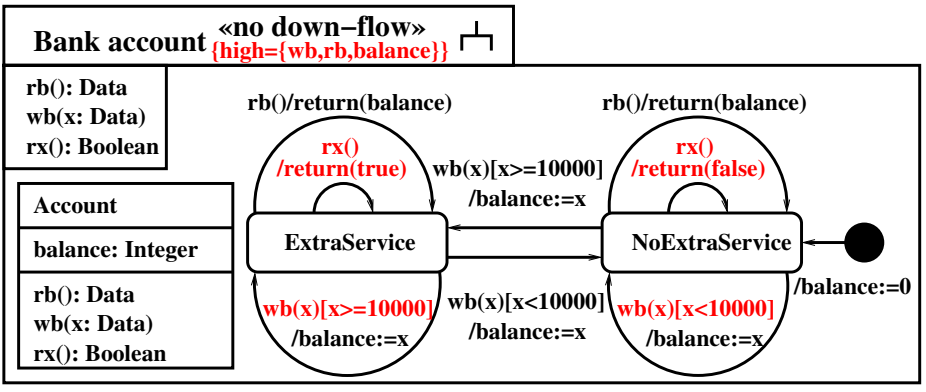

Fig. 7. Bank account data object

of high messages. Consider the sequences $\boldsymbol{i} \stackrel{\text{def}}{=} (\{\!\{ wb(0) \}\!\}, \{\!\{ rx() \}\!\})$ and $\boldsymbol{j} \stackrel{\text{def}}{=} (\{\!\{ wb(10000) \}\!\}, \{\!\{ rx() \}\!\})$. We have $\boldsymbol{i}_H = (\{\!\{ \}\!\}, \{\!\{ rx() \}\!\}) = \boldsymbol{j}_H$. From the definition of the behavioral semantics of statecharts sketched in Sect. 2.1, we can see that $\boldsymbol{o} \stackrel{\text{def}}{=} (\{\!\{ \}\!\}, \{\!\{ return(false) \}\!\}) \in [\![A]\!](\boldsymbol{i})$ and $\boldsymbol{p} \stackrel{\text{def}}{=} (\{\!\{ \}\!\}, \{\!\{ return(true) \}\!\}) \in [\![A]\!](\boldsymbol{j})$. But then $\boldsymbol{o}_H = (\{\!\{ \}\!\}, \{\!\{ return(false) \}\!\}) \neq (\{\!\{ \}\!\}, \{\!\{ return(true) \}\!\}) = \boldsymbol{p}_H$, as required.

Note that, while in the given example, it may be easy to see that the system does not satisfy the «no down − flow» constraint, it is in general not simple to establish that a system does satisfy this constraint, which is why in [Jür04] we provide the formal semantics sketched in Sect. 2.1.

Data security. This stereotype labeling (instances of) subsystems has the following constraint. The subsystem behavior respects the data security requirements given by the stereotypes «critical» and the associated tags contained in the subsystem, with respect to the threat scenario arising from the deployment diagram.

More precisely, the constraint is given by the following three conditions (of which the first two use the concepts of preservation of secrecy resp. integrity defined in Sect. 2.3).

secrecy. The subsystem preserves the secrecy of the data designated by the tag {secrecy} against adversaries of type A.

integrity. The subsystem preserves the integrity of the data designated by the tag {integrity} against adversaries of type A.

freshness. Within the subsystem $\mathcal{S}$ stereotyped «data security» the following condition holds for any subsystem instance or object model $\mathcal{D}$ stereotyped «critical» for any value *data* of the associated tag {fresh}: *data* occurs within $\mathcal{S}$ at most in

- the object model or subsystem instance model representing $\mathcal{D}$ in the static structure diagram contained in $\mathcal{S}$,
- the swim-lanes belonging to $\mathcal{D}$ in the activitiy diagram contained in $\mathcal{S}$,

- the statechart diagrams contained in $\mathcal{S}$ that model parts of the behavior of $\mathcal{D}$, or
- $\mathcal{D}$'s part of the connections in the sequence diagram contained in $\mathcal{S}$.

Note that it is enough for data to be listed with a security requirement in *one* of the objects or subsystem instances contained in the subsystem to be required to fulfill the above conditions.

Thus the properties of secrecy and integrity are taken relative to the type of adversary under consideration. In case of the default adversary, this is a principal external to the system; one may, however, consider adversaries that are part of the system under consideration, by giving the adversary access to the relevant system components (by defining $\mathsf{Threats}_A(s)$ to contain access for the relevant stereotype s). For example, in an e-commerce protocol involving customer, merchant and bank, one might want to say that the identity of the goods being purchased is a secret known only to the customer and the merchant (and not the bank). This can be formulated by marking the relevant data as "secret" and by performing a security analysis relative to the adversary model "bank" (that is, the adversary is given access to the bank component by defining the $\mathsf{Threats}()$ function in a suitable way).

The secrecy and integrity tags can be used for data values as well as variable and message names (as permitted by the definitions of secrecy and integrity in Sect. 2.3). Note that the adversary does not always have access to the input and output queues of the system (for example, if the system under consideration is part of a larger system it is connected through a secure connection). Therefore it may make sense to use the secrecy tag on variables that are assigned values received by the system; that is, effectively, one may require values that are received to be secret. Of course, the above condition only ensures that the component under consideration keeps the values received by the environment secret; additionally, one has to make sure that the environment (for example, the rest of the system apart from the component under consideration) does not make these values available to the adversary.

Example. The example in Fig. 8 shows the specification of a very simple security protocol. The sender requests the public key K together with the certificate $\mathcal{S}ign_{K_{CA}}(rcv :: K)$ certifying authenticity of the key from the receiver and sends the data d back encrypted under K (here $\{M\}_K$ is the encryption of the message M with the key K, $\mathcal{D}ec_K(C)$ is the decryption of the ciphertext C using K, $\mathcal{S}ign_K(M)$ is the signature of the message M with K, and $\mathcal{E}xt_K(S)$ is the extraction of the data from the signature using K). Assuming the *default* adversary type and by referring to the adversary model outlined in Sect. 2.3 and by using the formal semantics defined in [Jür04], one can establish that the secrecy of d is preserved. (Note that the protocol only serves as a simple example for the use of patterns, not to propose a new protocol of practical value.) Recall from Sect. 2.4 that the requirements $\{\mathsf{secrecy}\}$ and $\{\mathsf{integrity}\}$ refer to the type of adversary under consideration. In the case of the default adversary, in this example this is an adversary that has access to the Internet link between the two nodes only. It does not have direct access to any of the components in the specification (this

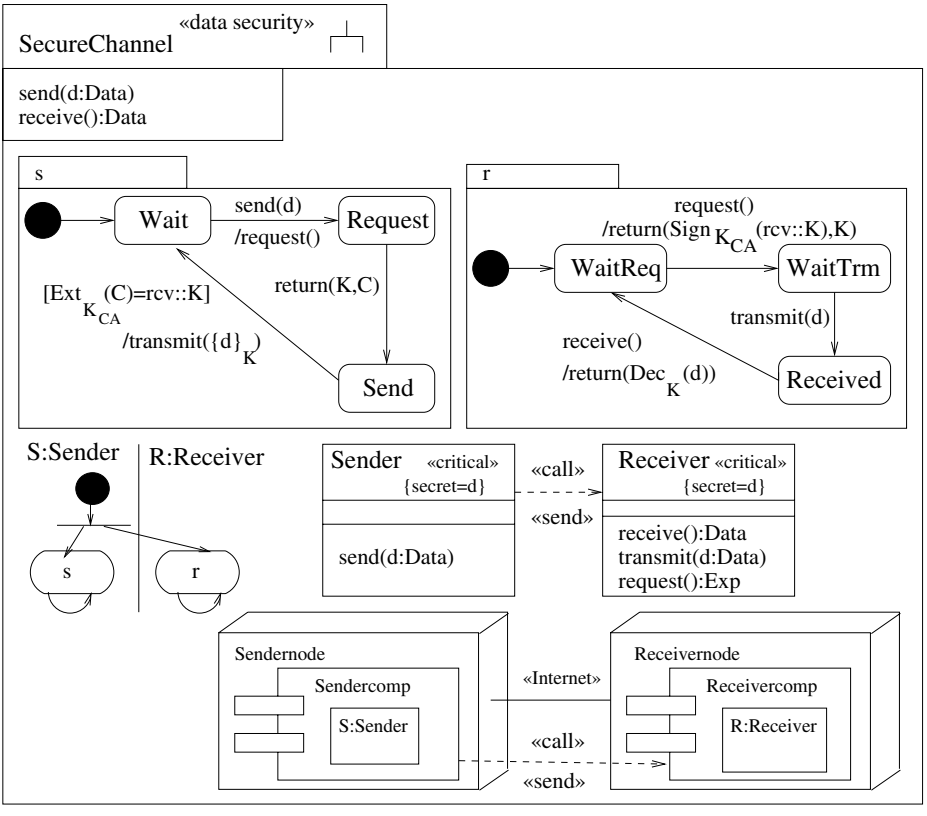

Fig. 8. Security protocol

would have to be specified explicitly using the Threats() function). In particular, the adversary to be considered here does not have access to the components R and S (if it would, then secrecy and integrity would fail because the adversary could read and modify the critical values directly as attributes of R and S).

Again, verifying that a given system satisfies the «data security» constraint is in general non-trivial, even for small specifications as the example above. We therefore provided tool-support for an automated formal verification to assist this task in Sect. 4. Note also that, while it is often possibly to use standard security protocols (such as SSL), which may already be verified, our work in industrial projects has shown that for a variety of reasons, self-designed protocols are still developed and used in industry (see for example [GHJW03]).

The stereotypes «secure links», «secure dependencies», and «data security» describe different conditions for ensuring secure data communication: «secure links» ensures that the security requirements on the communication dependencies between components are supported by the physical situation, relative to the adversary model under consideration. The stereotype «secure dependencies» ensures that the security requirements in different parts of a static structure di-

agram are consistent. Finally, « data security » ensures that security is enforced on the behavior level. – One could for example merge the conditions of « secure links » and « secure dependencies » to give one stereotype; we keep them separate to facilitate understanding and because one might like to use the stereotype « secure dependencies » in situations where no implementation diagram is present.

Fair Exchange. This stereotype of subsystems has associated tags {start} and {stop} taking names of states as values. The associated constraint requires that, whenever a {start} state in the contained activity diagram is reached, then eventually a corresponding {stop} state will be reached. This allows one to formalize the fair exchange requirement as explained in [Jür04]. This is formalized for a given subsystem $\mathcal{S}$ as follows. $\mathcal{S}$ fulfills the constraint of « fair exchange » if for every adversary *adv* of type A and every sequence of input event multi-sets $I_1, \ldots, I_n$, the following implication holds: For any state specified by {start} that the function associated with $\mathcal{S}$ reaches, it subsequently eventually reaches a state specified by {stop}.

Provable. A subsystem instance $\mathcal{S}$ may be labelled « provable » with associated tags {action}, and {cert}, to specify that $\mathcal{S}$ may output the expression $E \in \mathbf{Exp}$ given in {cert} (which serves as a proof that the action at state {action} was performed) only after the state having a name given in {action} is reached. Here the certificate in {cert} is assumed to be unique for each subsystem instance. This is formalized as follows: $\mathcal{S}$ fulfills the constraint if

- for each sequence of event multi-sets $I_1, \ldots, I_k$,
- for each adversary *adv* of type A, and
- for each sequence $(O_1, \ldots, O_k)$ output by $\mathcal{S}$ when executed with an adversary *adv* on input of $(I_1, \ldots, I_k)$,
- and if $(S_1, \ldots, S_k)$ is the corresponding sequence of executed states,

the following implication holds: If there exists an i such that the output O_i equals to the expression in {certificate}, then we have $j \leq i$ such that the state multi-set S_j contains the state specified by action. Again, more explanation can be found in [Jür04].

Example. Fig. 9 gives a subsystem instance describing the following situation: a customer buys a good from a business. The semantics of the stereotype « fair exchange » is, intuitively, that the actions listed in the tags {start} and {stop} should be linked in the sense that if the former is executed then eventually the latter will be. This would entail that, once the customer has paid, either the order is delivered to him by the due date, or he is able to reclaim the payment on that date. To avoid illegitimate repayment claims, one could employ the stereotype « provable » with regards to the state Pay, in order to make sure that the Reclaim payment action checks whether the Customer can provide a proof of payment.

Guarded Access. This stereotype of (instances of) subsystems is supposed to mean that each object in the subsystem that is stereotyped « guarded » can only

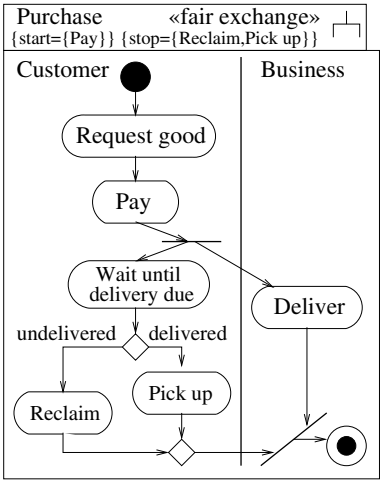

Fig. 9. Purchase activity diagram

be accessed through the objects specified by the tag {guard} attached to the «guarded» object. This way, one can define access control policies, similar the approach taken in the Java 2 security architecture. Formally, we assume that we have $name \notin \mathcal{K}_A^p$ for the adversary type A under consideration and each name $name$ of an instance of a «guarded» object (that is, a reference is not publicly available), and that for each «guarded» object there is a statechart specification of an object whose name is given in the associated tag {guard}. This way, we model the passing of references. This is explained in detail in [Jür04].

3.1 Discussion

We shortly discuss the aspects of security covered by the UMLsec extension.

Security requirements. Formalizations of basic security requirements are provided via stereotypes, such as «secrecy» and «integrity».

Threat scenarios. Threat scenarios are incorporated using the formal semantics and depending on the underlying physical layer via the sets $\mathsf{Threats}_{adv}(ster)$ of actions available to the adversary of kind adv.

Security concepts. We have shown how to incorporate security concepts such as tamper-resistant hardware (using threat scenarios, in this case).

Security mechanisms. As an example, we demonstrated modeling of the Java security architecture access control mechanisms.

Security primitives. Security primitives are either built in (such as encryption) or can be treated (such as security protocols).

Underlying physical security. This can be addressed as demonstrated by the stereotype «secure link» in deployment diagrams.

Security management. This can be considered in our approach by using activity diagrams (as in Fig. 9).

Additional domain knowledge has been incorporated regarding Java security and CORBA applications, as well as smart card security (see [Jür04] for more details).

Note that when adapting a modeling language to security requirements, one needs to make sure that the features used to express security properties on the design level actually map to system constructs on the implementation level which do provide these properties. Since we assume, for example, that attributes can only be accessed through the operations of an object, and that only the explicitly offered operations of a subsystem can be called from outside it, it is generally security-critical that this is enforced on the implementation level.

4 Formal Security Verification of UML Models

We present some work on automated formal verification of the security requirements expressed in the UMLsec notation. This tool-support is embedded in a framework supporting the construction of automated requirements analysis tools for UML diagrams. The framework is connected to industrial CASE tools using data integration with XMI [XMI02] and allows convenient access to this data and to the human user. In this chapter, we will, as an example for a usage of this framework, present verification routines to verify the constraints associated with the stereotypes of UMLsec using automated theorem provers (ATPs).

The goal is, in particular, that advanced users of the UMLsec approach should be able to use this framework to implement verification routines for the constraints of self-defined stereotypes, in a way that allows them to concentrate on the verification logic (rather than on user interface issues).

The usage of the framework as illustrated in Fig. 10 proceeds as follows. The developer creates a model and stores it in the UML 1.5/XMI 1.2 file format.[4] The file is imported by the UML verification framework into the internal MDR repository. MDR is an XMI-specific data-binding library which directly provides a representation of an XMI file on the abstraction level of a UML model through Java interfaces (JMI). This allows the developer to operate directly with UML concepts, such as classes, statecharts, and stereotypes. It is part of the Netbeans project [Net03]. Each plug-in accesses the model through the JMI interfaces generated by the MDR library, they may receive additional textual input, and they may return both a UML model and textual output. The two exemplary analysis plug-ins proceed as follows: The static checker parses the model, verifies its static features, and delivers the results to the error analyzer. The dynamic checker translates the relevant fragments of the UML model into the automated theorem prover input language. The automated theorem prover is spawned by the UML framework as an external process; its results are delivered back to the error analyzer. The error analyzer uses the information received from the static checker and dynamic checker to produce a text report for the developer describing the problems found, and a modified UML model, where the errors

[4] This will be updated to UML 2.0 once the corresponding DTD has been officially released.

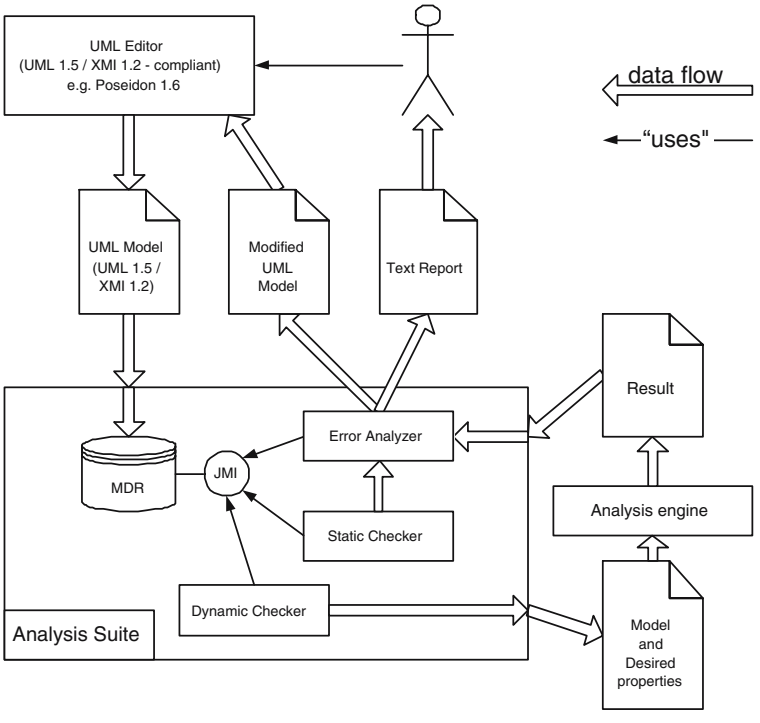

Fig. 10. UML verification framework: usage

found are visualized. Besides the automated theorem prover binding presented in this section there are other analysis plugins including a model-checker binding [JS04] and plugins for simulation and test-sequence generation.

The framework is designed to be extensible: advanced users can define stereotypes, tags, and first-order logic constraints which are then automatically translated to the automated theorem prover for verification on a given UML model. Similarly, new adversary models can be defined.

The user webinterface and the source code of the verification framework is accessible at [UML04].

4.1 Translating UMLsec Diagrams to First-Order Logic Formulas

We explain the automated translation of UMLsec diagrams to first-order logic (FOL) formulas which allows automated analysis of the diagrams using automated first-order logic theorem provers such as e-SETHEO [SW00, MIL$^+$97] or SPASS. More precisely, we assume that we are given a UML package containing the following kinds of diagrams: A deployment diagram specifies the physical layer of the system, such as system nodes and communication links, and the level of security it provides, using UMLsec stereotypes, such as «Internet» denoting an Internet communication link. From this, in the security analysis, the

$$\begin{aligned}\forall E_1, E_2.\ \big(&\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2) \\ &\Rightarrow \mathsf{knows}(E_1 :: E_2) \wedge \mathsf{knows}(\{E_1\}_{E_2}) \wedge\ \mathsf{knows}(\mathcal{S}ign_{E_2}(E_1))\big) \\ \wedge&\big(\mathsf{knows}(E_1 :: E_2) \Rightarrow \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\big) \\ \wedge&\big(\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1}) \Rightarrow \mathsf{knows}(E_1)\big) \\ \wedge&\big(\mathsf{knows}(\mathcal{S}ign_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2) \Rightarrow \mathsf{knows}(E_1)\big)\end{aligned}$$

Fig. 11. Some structural formulas

adversary model is generated in first-order logic who is able to control certain communication links. Secondly, a class diagram describes the data structure of the system, including the security requirements on the system data, for example using the UMLsec tags {secrecy}, {integrity} and {authenticity} which represent the respective requirements. For the security analysis, from this information the conjecture is derived that is to be checked by the automated theorem prover. The package also contains diagrams specifying the intended behavior of the system, which may include an activity diagram coordinating the components or objects in the package, a sequence diagram specifying interaction between them by message exchange (making use of cryptographic operations using the notation from Sect. 2.2), and statecharts specifying the behavior of single components or objects. The behavioral specifications are compiled to first-order logic axioms giving an abstract interpretation of the system behavior suitable for security analysis. In the following, we explain this translation for sequence diagrams. It works similarly for statecharts and activity diagrams. The formalization automatically derives an upper bound for the set of knowledge the adversary can gain. For space restrictions, we can only present a simplified treatment and have to omit issues such as session key generation.

The idea is to use a predicate $\mathsf{knows}(E)$ meaning that the adversary may get to know E during the execution of the protocol. For any data value s supposed to remain secret as specified in the UMLsec model, one thus has to check whether one can derive $\mathsf{knows}(s)$. The set of predicates defined to hold for a given UMLsec specification is defined as follows.

For each publicly known expression E, one defines $\mathsf{knows}(E)$ to hold. The fact that the adversary may enlarge his set of knowledge by constructing new expressions from the ones he knows (including the use of encryption and decryption) is captured by the formula in Fig. 11.

For our purposes, a sequence diagram is essentially a sequence of command schemata of the form *await event e – check condition g – output event e'* represented as *connections* in the sequence diagrams. Connections are the arrows from the life-line of a source object to the life-line of a target object which are labeled with a message to be sent from the source to the target and a guard condition that has to be fulfilled.

Suppose we are given a connection $l = (\mathsf{source}(l), \mathsf{guard}(l), \mathsf{msg}(l), \mathsf{target}(l))$ in a sequence diagram with $\mathsf{guard}(l) \equiv cond(arg_1, \ldots, arg_n)$, and $\mathsf{msg}(l) \equiv exp(arg_1, \ldots, arg_n)$, where the parameters arg_i of the guard and the message

are variables which store the data values exchanged during the course of the protocol. Suppose that the connection l' is the next connection in the sequence diagram with $\mathsf{source}(l') = \mathsf{source}(l)$. For each such connection l, we define a predicate $\mathsf{PRED}(l)$ as in Fig. 12. If such a connection l' does not exist, $\mathsf{PRED}(l)$ is defined by substituting $\mathsf{PRED}(l')$ with true in Fig. 12.

The formula formalizes the fact that, if the adversary knows expressions $exp_1, \ldots, exp_n$ validating the condition $cond(exp_1, \ldots, exp_n)$, then he can send them to one of the protocol participants to receive the message $exp(exp_1, \ldots, exp_n)$ in exchange, and then the protocol continues. With this formalization, a data value s is said to be kept secret if it is not possible to derive $\mathsf{knows}(s)$ from the formulas defined by a protocol. This way, the adversary knowledge set is approximated from above (because one abstracts away for example from the message sender and receiver identities and the message order). This means, that one will find all possible attacks, but one may also encounter "false positives", although this has not happened yet with any practical examples. The advantage is that this approach is rather efficient (see Sect. 4.3 for some performance data).

For each object O in the sequence diagram, this gives a predicate $\mathsf{PRED}(O) = \mathsf{PRED}(l)$ where l is the first connection in the sequence diagram with $\mathsf{source}(l) = O$. The axioms in the overall first-order logic formula for a given sequence diagram are then the conjunction of the formulas representing the publicly known expressions, the formula in Fig. 11, and the conjunction of the formulas $\mathsf{PRED}(O)$ for each object O in the diagram. The conjecture, for which the ATP will check whether it is derivable from the axioms, depends on the security requirements contained in the class diagram. For the requirement that the data value s is to be kept secret, the conjecture is $\mathsf{knows}(s)$. An example is given in Sect. 4.2.

One can define a variation of the formula in Fig. 12 by joining all subformulas $\mathsf{PRED}(l), \mathsf{PRED}(l'), \ldots$ for connections $l, l', \ldots$ in the sequence diagram using the conjunction operator $\wedge$, instead of including the predicate $\mathsf{PRED}(l')$ for next connection l' in the conclusion of the implication in $\mathsf{PRED}(l)$. The effect is that the order of the connections in the sequence diagram is then ignored. This results in a more coarse abstract interpretation of the sequence diagram than that in Fig. 12, which may produce more false positives: allegedly insecure specifications which are in fact secure in reality, because there the order of the connection is in fact observed. However, in particular architectures the order of messages in the sequence diagram is in fact not enforced, and then this variation is useful. For example, this is the case for the industrial application project which we report on in Sect. 6.

$$
\begin{aligned}
&\mathsf{PRED}(l) = \\
&\forall exp_1, \ldots, exp_n \big(\mathsf{knows}(exp_1) \wedge \ldots \wedge \mathsf{knows}(exp_n) \\
&\qquad \wedge\ cond(exp_1, \ldots, exp_n) \\
&\qquad \Rightarrow \mathsf{knows}(exp(exp_1, \ldots, exp_n) \\
&\qquad\quad \wedge\ \mathsf{PRED}(l')\big)
\end{aligned}
$$

Fig. 12. Connection predicate

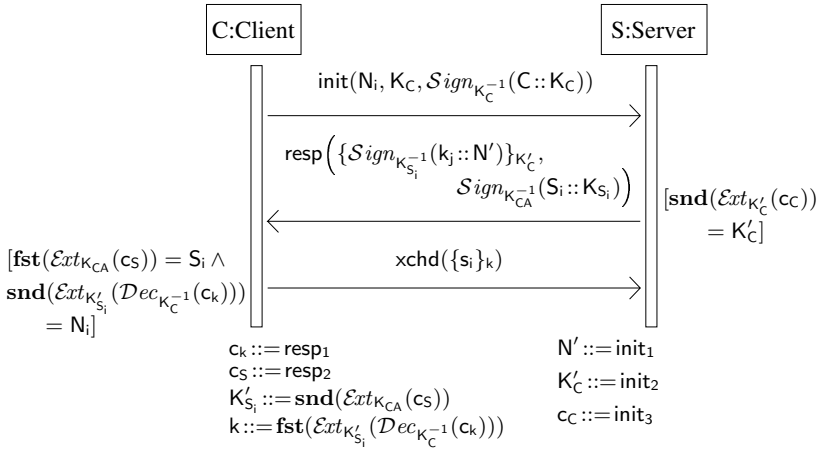

Fig. 13. Variant of the TLS handshake

4.2 A Variant of the TLS Protocol

We will analyze a variant of the handshake protocol of TLS[5] examined in [Jür04] (note that this is not the variant of TLS in common use but a variant proposed at the conference IEEE Infocom 1999). To show applicability of our approach, we demonstrate the flaw from [Jür04], suggest a correction, and verify it. The goal of the protocol is to let a client send a secret over an untrusted communication link to a server in a way that provides secrecy and server authentication, by using symmetric session keys.

The central part of the specification of this protocol is shown in Fig. 13. Parts that have to be left out here are firstly a deployment diagram specifying that the two protocol participants client and server are connected by an Internet connection, using the UMLsec stereotype « Internet ». From this, in the security analysis, the adversary model who is able to control this communication link is generated. Secondly, there is a class diagram which includes various security requirements on the protocol data as UMLsec tags {secrecy}, {integrity} and {authenticity}. For the security analysis, from this information the conjecture is derived that is to be checked by the automated theorem prover. Most importantly, the value s which is exchanged encrypted in the last message of the protocol is required to remain secret.

Depicted in Fig. 13, the protocol proceeds as we explain in the following. The client C initiates the protocol by sending the message $\mathsf{init}(\mathsf{N_i}, \mathsf{K_C}, \mathcal{S}ign_{\mathsf{K_C^{-1}}}(\mathsf{C} :: \mathsf{K_C}))$ to the server S. Suppose that the condition $[\mathbf{snd}(\mathcal{E}xt_{\mathsf{K'_C}}(\mathsf{c_C}))=\mathsf{K'_C}]$ holds, where $\mathsf{K'_C} ::= \mathsf{init_2}$ and $\mathsf{c_C} ::= \mathsf{init_3}$, that is, the key $\mathsf{K_C}$ contained in the signature matches the one transmitted in the clear. Then S sends the message

[5] TLS (transport layer security) is the successor of the Internet security protocol SSL (secure sockets layer).

```
input_formula(protocol,axiom,(
 ![Init_1, Init_2, Init_3, Resp_1, Resp_2, Xchd_1] : (
 % C <-> Attacker
(      ( ( true & true )
       => knows(conc( n,  conc( k_c,  sign(conc(c, k_c)), inv(k_c)) ) ))
     & ( ( knows(Resp_1) & knows(Resp_2)
     &equal(fst(ext(Resp_2,k_ca)),s)&equal(snd(ext(dec(Resp_1,inv(k_c)),
                snd(ext(Resp_2, k_ca)))), n ) )
=>knows(symenc(s,fst(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca))))))
)))
 &  % S <-> Attacker
(      ( ( knows(Init_1) & knows(Init_2) & knows(Init_3)
       & equal( snd(ext(Init_3, Init_2)), Init_2 ) )
  => knows(conc(enc(sign(conc(kgen(Init_2),Init_1),inv(k_s)),Init_2),
                sign(conc(s, k_s), inv(k_ca) ) ))
     & ( ( knows(Xchd_1) & true )
        => true
)) )  ) )).
```

Fig. 14. Protocol part of translation to TPTP

$\mathsf{resp}\big(\{\mathcal{S}ign_{\mathsf{K_S}^{-1}}(\mathsf{k_j} :: \mathsf{N}')\}_{\mathsf{K}'_\mathsf{C}}, \mathcal{S}ign_{\mathsf{K_{CA}}^{-1}}(\mathsf{S} :: \mathsf{K_S})\big)$ back to C, where $\mathsf{N}' ::= \mathsf{init}_1$. Now suppose that the condition

$$[\mathbf{fst}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))=\mathsf{S} \wedge \mathbf{snd}(\mathcal{E}xt_{\mathsf{K}'_{\mathsf{S_i}}}(\mathcal{D}ec_{\mathsf{K_C}^{-1}}(\mathsf{c_k})))=\mathsf{N_i}]$$

holds, where $\mathsf{c_S} ::= \mathsf{resp}_1$, $\mathsf{c_k} ::= \mathsf{resp}_2$, and $\mathsf{K}'_{\mathsf{S_i}} ::= \mathbf{snd}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_S}))$, that is, the certificate is actually for S and the correct nonce is returned. Then C sends $\mathsf{xchd}(\{\mathsf{s_i}\}_\mathsf{k})$ to S, where $\mathsf{k} ::= \mathbf{fst}(\mathcal{E}xt_{\mathsf{K}'_{\mathsf{S_i}}}(\mathcal{D}ec_{\mathsf{K_C}^{-1}}(\mathsf{c_k})))$. If any of the checks fail, the respective protocol participant stops the execution of the protocol.

The main part of the result of the transformation to the e-SETHEO input format TPTP is the protocol definition given in Fig. 14. We have to omit the formulas representing the initial adversary knowledge and the effect of message recombination on the intruder's knowledge predicate knows. The TPTP notation is the de-facto input notation for first-order logic automated theorem provers [SS01], supported, using existing converters, by a variety of provers including also Otter, SPASS, Vampire, and Waldmeister.

Note that in this notation, conjunction is written as &, and forall resp. exists quantification as $![\mathsf{X1}, \ldots, \mathsf{Xm}]$ resp. $?[\mathsf{X1}, \ldots, \mathsf{Xm}]$, where X1, ..., Xm are the quantified variables. Also, encryption, signature, and concatenation are represented respectively as binary functions enc, sign, and conc in TPTP. The private key belonging to the public key K is written as inv(K). Constants, such as the nonce N, have to be written in small letters in TPTP.

The protocol itself is expressed by a forall quantification over variables representing the arguments of messages which are transferred over the communication link. Here, the message variables Resp_1, and Resp_2 represent the messages received by the client. The message variables Init_1, Init_2, Init_3, and Xchd_1 stand

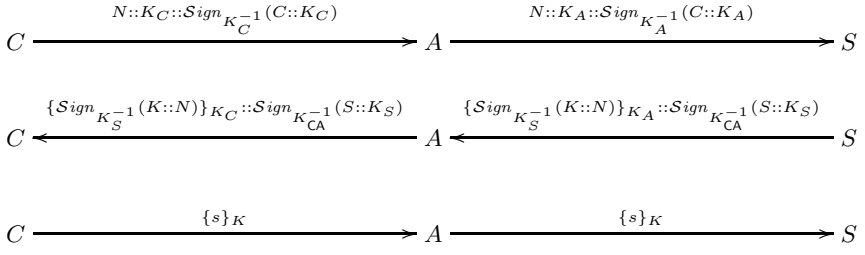

Fig. 15. Attack Visualization: Man-in-the-middle

for the server receiving messages parts. The protocol example includes three messages (cf. Fig. 13), of which the first and third are sent by the client and the second by the server. Each message is expressed by a single implication in the main formula. Therefore three implications occur in Fig. 14 (of which the second is nested in the first). The first,

$$\mathsf{knows(n)} \,\&\, \mathsf{knows(k\_c)} \,\&\, \mathsf{knows(sign(conc(c, k\_c), inv(k\_c)))},$$

is the message sent from the client to the server. It has the precondition true because it is sent unconditionally without previous receipt of any other message. The postconditions of the implications include the messages sent over the communication channel.

4.3 Protocol Analysis with ATPs

We use the ATP e-SETHEO [SW00, MIL+97] for verifying security protocols as a "black box": A TPTP input file is presented to the ATP and an output from the ATP is observed. No internal properties of or information from e-SETHEO is used. This allows one to use e-SETHEO interchangingly with any other ATP accepting TPTP as an input format (such as SPASS, Vampire and Waldmeister) when it may seem fit.

With respect to the security analysis described in Sect. 4.1, the results of the theorem prover have to be interpreted as follows: If the conjecture stating for example that the adversary may get to know the secret can be derived from the axioms which formalize the adversary model and the protocol specification, this means that there may be an attack against the protocol. We then use an attack generation machine programmed in Prolog to construct the attack (also contained in the analysis tool suite [UML04]). If the conjecture cannot be derived from the axioms, this constitutes a proof that the protocol is secure with respect to the security requirement formalized as the negation of the conjecture, because logical derivation is sound and complete with respect to semantic validity for first-order logic. Note that since first-order logic in general is undecidable, it can happen that the ATP is not able to decide whether a given conjecture can be derived from a given set of axioms. However, experience has shown that for a

reasonable set of protocols and security requirements, our approach is in fact decidable.

In our example, e-SETHEO returns as an output that the conjection knows(s) can be derived from the defined rules (within one second). For this example the attack tracking tool needs a few seconds to produce the attack. The derived message flow diagram corresponding to a man-in-the-middle attack is depicted in Fig. 15.

We can fix this problem by substituting $K :: N$ in the second message (server to client) by $K :: N :: K_C$ and by including a check regarding this new message part at the client. Now the new version with the additional signature information about the client key k_c can be verified by the automated theorem prover approach. When e-SETHEO runs on the fixed version of the protocol it now gives back the result that the conjecture knows(s) cannot be derived from the axioms formalizing the protocol. Note that this result, which was delivered within a few seconds, means that the actually exists no such derivation, not just that the theorem prover is not able to find it. This means in particular that the attacker cannot gain the secret knowledge anymore. Note that this statement of course in itself is bound to the particular execution model and the formalizations of the security requirements used here. The security analysis may falsely claim that there may be an attack against the specified system, because of the optimizing abstractions used. This, however, has not so far surfaced as a limitation in practical applications.

5 Source Code Analysis

In recent work, we have applied our analysis techniques explained in the previous section to the security verification of cryptographic protocols implemented in C, making use of control flow graphs generated from the source code. This way, one can find security weaknesses which may have been introduced during the (manual or automated) transition from specifications to code. Such security weaknesses may be introduced not only by programming mistakes, but also because some security-relevant details are abstracted away on the specification level. For space restrictions, details have to be omitted but can be found in [Jür05c, Jür05b]. A link between the specification layer and the source code layer has been established in the context of aspect-oriented development in [JH05].

6 Industrial Case-Study: Biometric Authentication

We applied our methods and tools in an industrial application project with a major German company. The goal of the project was the correct development of a security-critical biometric authentication system which is supposed to control access to a protected resource, for example by opening a door or letting someone log into a computer system. In this system, a user carries his biometric reference data on a personal smart-card. To gain access, he inserts the smart-card in the card reader and delivers a fresh biometric sample at the biometric sensor, for example a finger-print reader. Since the communication links between the

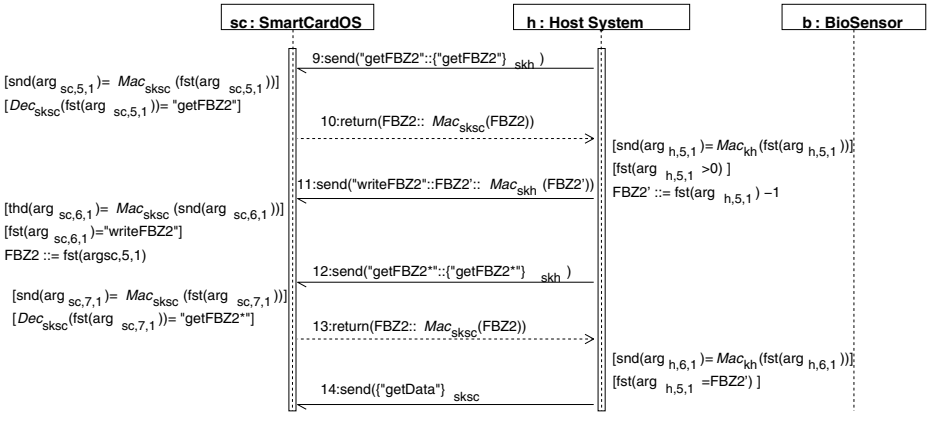

Fig. 16. Excerpt from biometric authentication protocol

host system (containing the bio-sensor), the card reader, and the smart-card are physically vulnerable, the system needs to make use of a cryptographic protocol to protect this communication. Because the correct design of such protocols and the correct use within the surrounding system is very difficult, our method was chosen to support the development of the biometric authentication system.

Within the project, the system was specified using UML diagrams: a deployment diagram describing the architecture, a class diagram defining the data structure, an activity diagram specifying the general workflow, and a sequence diagram giving a detailed specification of the cryptographic protocol. A fragment of the sequence diagram is shown in Fig. 16.

In the next step, this specification was enriched with security-relevant information, according to the UMLsec extension. This includes specifying the level of security provided by the physical layer of the system in the deployment diagram, and formulating security goals on the execution of the system and on the protection of particular data values in the activity and class diagrams.

Then the security of the protocol was analyzed using the automated tool support described in the previous section. The analysis is done with respect to the threat model which is derived from a deployment diagram of the system and the security goals contained in the class diagram, as explained in the previous sections. This way, it turned out that the protocol in fact contains a vital flaw. To prevent an attack where an attacker simply repeatedly tries to match a forged biometric sample, for example, using an artificial finger, with a forged or stolen smart-card, the protocol contains a misuse counter which is decreased from an initial value of 3 to 0, when the card will be disabled. The attack which was found using our tools enables the attacker to prevent the misuse counter from being updated, thereby enabling a brute-force attack.

The relevant part of the attack is displayed in Fig. 17. The attacker is assumed to control the communication between the smart-card and the host system, which is realistic since it is not protected by physical means. He chooses to act as a relay between the smart-card and the host system, until the host

Fig. 17. Attack against biometric authentication protocol

system signals the smart-card to decrease its misuse counter FBZ2 by sending it the message writeFBZ2 which contains the new, decreased value FBZ2′ that the smart-card should assign to its counter. This message is simply dropped by the attacker. Note that it is possible to simply drop the message although the integrity of the message is protected using a Message Authentication Code (MAC) in its third argument $\mathsf{Mac}_{\mathsf{skh}}(\mathsf{FBZ2})$. Here skh is a secret key of the host system, which in a correct protocol is supposed to be equal to the secret key sksc of the smart-card. One should note here that the smart-card does not keep an internal state of the protocol execution history. This means that it accepts any of the messages in the protocol at any point. Therefore, after dropping the message telling the smart-card to decrease its misuse counter, the protocol can simply proceed with the next message from the host system, which is again forwarded by the adversary to the smart-card. This problem had already been detected by our tools at an earlier version of the protocol. To fix it, the protocol was extended with the message getFBZ2 by which the host system tries to make sure that the misuse counter has actually been decreased, as shown in Fig. 16. The return value then expected by the host system from the smart-card is the misuse counter FBZ2, protected in its integrity by also sending the MAC $\mathsf{MAC}_{\mathsf{sksc}}(\mathsf{FBZ2})$, which is supposed to be correctly decreased to give the value FBZ′. Unfortunately, this value had already been sent in the previous message writeFBZ2, since the keys skh and sksc are supposed to be the same, so the adversary only needs to replay the value from that message to the host system.

Based on our findings, the protocol was corrected by using a different one of the coding modes suggested in the specification that makes sure that the return message from the getFBZ2 message cannot be a replay of earlier messages, using a freshly generated random value. This corrected version of the protocol is currently subject to ongoing analysis using our tools.

Since UML was used in the development of this system anyhow, the only extra effort needed was to extend the UML diagrams with the security-critical

information describing the level of physical security, and the security goals to be achieved, as explained above. Considering the gain from using our methods, namely detecting several mistakes in various versions of the protocol, and making sure that the final version is correct, this modest extra effort seems to be worthwhile. In conclusion, experiences from this industrial application have been quite positive.

7 Related Work

So far, there seems to be no comparable approach which allows one to include a comparable variety of security requirements in a UML specification which is then, based on a formal semantics, formally verified for these requirements using tools such as automated theorem provers and model-checkers, and which comes with a transition to the source code level where automated formal verification can also be applied.

There has, however, been a substantial amount of work regarding some of the topics we address here (for example formal verification of security-critical systems or secure systems development with UML). A detailed comparison with related work has to be omitted here for space reasons, but can be found in [Jür04]. Many related approaches can also be found in the CSDUML workshop series [JFFuCH04].

8 Conclusion and Future Perspectives

We gave an overview over the extension UMLsec of UML for secure systems development, in the form of a UML profile using the standard UML extension mechanisms. Recurring security requirements are written as stereotypes, the associated constraints ensure the security requirements on the level of the formal semantics, by referring to the threat scenario also given as a stereotype. Thus one may evaluate UML specifications to indicate possible vulnerabilities. One can thus verify that the stated security requirements, if fulfilled, enforce a given security policy.

At the hand of small examples, we demonstrated how to use UMLsec to model security requirements, threat scenarios, security concepts, security mechanisms, security primitives, underlying physical security, and security management.

As demonstrated, UMLsec can be used to encapsulate established rules on prudent security engineering, also by applying security patterns, and thereby makes them available to developers not specialized in security. While UML was developed to model object-oriented systems, one can also use UMLsec to analyze systems that are not object-oriented (assuming that the underlying assumptions, such as controlled access to data, are ensured).

We also explained how to analyse the UMLsec diagrams against security requirements with respect to their dynamic behavior, using automated theorem provers for first-order logic. We briefly reported on further work to apply this

formal security verification to the source code level of implementations derived from the UMLsec specifications.

The definition and evolvement of the UMLsec notation has been based on experiences from in industrial application projects. We reported on the use of UMLsec and its tool-support in one such application, the formal security verification of a biometric authentication system, where several security weaknesses were found and corrected using our approach during its development.

For space restrictions, we could only present a brief overview over a fragment of UMLsec. The complete notation with many more examples and applications can be found in [Jür04].

Although there exists a solid core UMLsec notation now, together with automated formal verification tools, which has proven its usefulness in several industrial application projects, there are a number of interesting open foundational and practical questions still to consider. Because our underlying formal system model is largely independent from UML specifics, it provide a suitable platform for such investigations also independently from UML. For example, one could use the UMLsec framework to formally explore ...

- ... which security properties are preserved under which conditions when composing or decomposing systems in modular components,
- ... the consistency of different security properties expressed as stereotypes when appearing in combination,
- ... which security properties are preserved under which conditions when refining a specification to a more detailed specification or eventually to the source-code,
- ... how to achieve a coherent level of security throughout the abstraction levels of a computing system,
- ... how to evaluate propose standards such as secure reference architectures,
- ... how security requirements can be achieved in the presence of other non-functional requirements such as dependability.

Acknowledgements

The research summarized in this chapter has benefitted from the help of too many people to include here; they are listed in [Jür04].

References

[Aba00] M. Abadi. Security protocols and their properties. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 39–60. IOS Press, Amsterdam, 2000. 20th International Summer School, Marktoberdorf, Germany.

[AJ01] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software (4th International Symposium, TACS 2001)*, volume 2215 of *Lecture Notes in Computer Science*, pages 82–94. Springer-Verlag, 2001.

[And01] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, New York, 2001.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.

[GHJW03] J. Grünbauer, H. Hollmann, J. Jürjens, and G. Wimmel. Modelling and verification of layered security-protocols: A bank application. In *SAFECOMP 2003*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Security and Privacy (S&P)*, pages 11–20. IEEE Computer Society, 1982.

[JFFuCH04] J. Jürjens, E.B. Fernandez, R.B. France, and B. Rumpe und C. Heitmeyer. Critical systems development using modelling languages (CSDUML'04): Current development and future challenges (report on the third international workshop). In N. Jardin Nunes, B. Selic, A. Silva, and A. Toval, editors, *UML Modeling Languages and Applications. UML 2004 Satellite Activities, Lisbon, Portugal, October 11–15, 2004, Revised Selected Papers*, volume 3297 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[JH05] J. Jürjens and S.H. Houmb. Dynamic secure aspect modeling with UML: From models to code. In *ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS / UML 2005)*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[JS04] J. Jürjens and P. Shabalin. Automated verification of UMLsec models for security requirements. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2004 – The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 412–425. Springer-Verlag, 2004.

[JS05] J. Jürjens and P. Shabalin. Tools for secure systems development with UML: Security analysis with ATPs. In *FASE 2005*, Lecture Notes in Computer Science, Edinburgh, 2-10 April 2005. Springer-Verlag.

[Jür02] J. Jürjens. Formal semantics for interacting UML subsystems. In B. Jacobs and A. Rensink, editors, *5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pages 29–44. International Federation for Information Processing (IFIP), Kluwer Academic Publishers, 2002.

[Jür04] J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2004.

[Jür05a] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *27th International Conference on Software Engineering (ICSE 2005)*. IEEE Computer Society, 2005.

[Jür05b] J. Jürjens. Understanding security goals provided by crypto-protocol implementations. In *21st International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society, 2005.

[Jür05c] J. Jürjens. Verification of low-level crypto-protocol implementations using automated theorem proving. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005)*. IEEE Computer Society, 2005.

[MIL+97] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – The CADE-13 Systems. *Journal of Automated Reasoning (JAR)*, 18(2):237–246, 1997.

[Net03] Netbeans project. Open source.
Available from `http://mdr.netbeans.org`, 2003.

[SS01] G. Sutcliffe and C. Suttner. The TPTP problem library for automated theorem proving, 2001. Available at `http://www.tptp.org`.

[SW00] G. Stenz and A. Wolf. E-SETHEO: An automated[3] theorem prover. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000)*, volume 1847 of *Lecture Notes in Computer Science*, pages 436–440. Springer-Verlag, 2000.

[UML04] UMLsec tool, 2002-04. Open-source. Accessible at http://www.umlsec.org.

[UML01] UML Revision Task Force. OMG UML Specification v. 1.4. OMG Document ad/01-09-67. Available at http://www.omg.org/uml, September 2001.

[Wat02] B. Watson. The Real-time UML standard. In *Real-Time and Embedded Distributed Object Computing Workshop*. OMG, July 15–18 2002.

[XMI02] Object Management Group. *OMG XML Metadata Interchange (XMI) Specification*, January 2002.