Model-based Security Engineering with UML: Introducing Security Aspects

Jan Jürjens*

Dep. of Informatics, TU Munich, Germany

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. In this tutorial exposition, we concentrate on an approach to develop and analyze security-critical specifications and implementations using aspect-oriented modeling.

1 Introduction

Constructing security-critical systems in a sound and well-founded way poses high challenges. To support this task, we defined the extension UMLsec of the UML [Jür01, Jür02, Jür04]. 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. In

* `http://www4.in.tum.de/~juerjens`

this tutorial exposition, we present in particular an Aspect-Oriented Modeling approach which separates complex security mechanisms (which implement the security aspect model) from the core functionality of the system (the primary model) in order to allow a security verification of the particularly security-critical parts, and also of the composed model.

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. In Sect. 4, we explain how one can specify security aspects in UMLsec models and how these are woven into the primary model using our approach. Sect. 5 explains our code analysis framework. Throughout the paper we demonstrate our approach using a variant of the Internet protocol Transport Layer Security (TLS). In Sect. 6, we report on experiences from using our approach in an industrial setting. After comparing our research with related work, we close with a discussion and an outlook on ongoing research.

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 offers three main "light-weight" language extension mechanisms: stereotypes, tagged values, and constraints [UML03].[1] 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 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 [UML03]. 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 below.

[1] In the following, we use UML 1.5 since the official DTD for UML 2.0, which would be needed by our tools, has not yet been released at the time of writing.

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 [Jür05b]. 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).

The semantics for the fragment of UML used for UMLsec is defined formally in [Jür04] using so-called *UML Machines*, which is a kind of state machine with input/output interfaces similar to the Focus model [BS01], whose behavior can be specified in a notation similar to that of Abstract State Machines [Gur00], and which is equipped with UML-type communication mechanisms. Because of space restrictions, we cannot recall the definition of UML Machines nor the formal semantics here completely. Instead, we use an abstraction of UML Machines in terms of their associated input/output functions to define precisely and explain the interfaces of the that part of our UML semantics that we need here to define the UMLsec profile. Our semantics includes simplified versions of 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. For our approach to be as widely applicable as possible and for the purposes of the current paper, we use a general definition of the notion of a component by which a component is a part of the system which interacts with the rest of the system (and possibly the system environment) through a well-defined interface (this definition is inspired for example by the view taken in [BS01]).

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, 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

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

Fig. 1. Abstract Crypto Operations

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. **Data** may also include *nonces* and other secrets.

To define the *algebra of cryptographic expressions* **Exp** one considers a term algebra generated from ground data in **Var**∪**Keys**∪**Data** using the symbolic operations in Fig. 1. In that term algebra, one defines the equations $\mathcal{D}ec_{K^{-1}}(\{E\}_K) = E$ and $\mathcal{E}xt_K(\mathcal{S}ign_{K^{-1}}(E)) = E$ (for all $E \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$) and the usual laws regarding concatenation, $\mathsf{head}()$, and $\mathsf{tail}()$. We write $\mathbf{fst}(E) \stackrel{\text{def}}{=} \mathbf{head}(E)$, $\mathbf{snd}(E) \stackrel{\text{def}}{=} \mathbf{head}(\mathbf{tail}(E))$, and $\mathbf{thd}(E) \stackrel{\text{def}}{=} \mathbf{head}(\mathbf{tail}(\mathbf{tail}(E)))$ for each $E \in \mathbf{Exp}$.

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. Several such adversary types are predefined in UMLsec and can be used directly (see Sect. 3). Advanced users can also define adversary types themselves by making use of threat sets defined below. If it is unknown which adversary type should be considered, one can use the most general adversary that has the capabilities of all possible adverary types. 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. They are predefined for the standard UMLsec adversary types, as explained in Sect. 3, but can also be defined by the advanced users themselves. 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

As an example, the formal definitions of the important security property secrecy is considered in this section following the standard approach of [DY83]. 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).

Stereotype	Base Class	Tags	Constraints	Description
Internet	link			Internet connection
encrypted	link			encrypted connection
LAN	link			LAN connection
secure links	subsystem		dependency security matched by links	enforces secure communication links
secrecy	dependency			assumes secrecy
integrity	dependency			assumes integrity
secure dependency	subsystem		«call», «send» respect data security	structural interaction data security
critical	object	secrecy, integrity		critical object
data security	subsystem		provides secrecy	basic datasecurity requirements

Table 1. UMLsec stereotypes (excerpt)

3 The UMLsec extension

We can only shortly recall part of the UMLsec notation here for space reasons. A complete account can be found in [Jür04]. In Table 1 we give some of the stereotypes from UMLsec, in Table 2 the associated tags, and in Table 3 the threats corresponding to the default adversary (which define the "security semantics" of the UMLsec models). The constraints connected to the stereotypes are explained in detail below. They can be formalized in first-order logic and thus verified by an automated first-order logic theorem prover, which is part of our UML analysis tool suite.

The primary model is a set of UML models and the dynamic aspect are weaved in by including the stereotypes defined above.

Internet, encrypted, LAN: These stereotypes on links (resp. nodes) in deployment diagrams denote the corresponding requirements on communication links (resp. system nodes), namely that they are implemented as Internet links, encrypted Internet links, resp. as LAN links. 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

$$s \in \{\text{«}\,\mathsf{Internet}\,\text{»}, \text{«}\,\mathsf{encrypted}\,\text{»}, \text{«}\,\mathsf{LAN}\,\text{»}\}$$

Tag	Stereotype	Type	Multipl.	Description
secrecy	critical	String	*	data secrecy
integrity	critical	String	*	data integrity

Table 2. UMLsec tags (excerpt)

Stereotype	$\mathsf{Threats}_{default}()$
Internet	{delete,read,insert}
encrypted	{delete}
LAN	$\emptyset$

Table 3. Some threats from the *default* attacker

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

- 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.

As an example the threat sets associated with the default adversary type are given in Table 3.

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 d with stereotype $s \in \{\text{«}\,\mathsf{secrecy}\,\text{»}, \text{«}\,\mathsf{integrity}\,\text{»}\}$ 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 $s = \text{«}\,\mathsf{secrecy}\,\text{»}$, 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. 2, 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

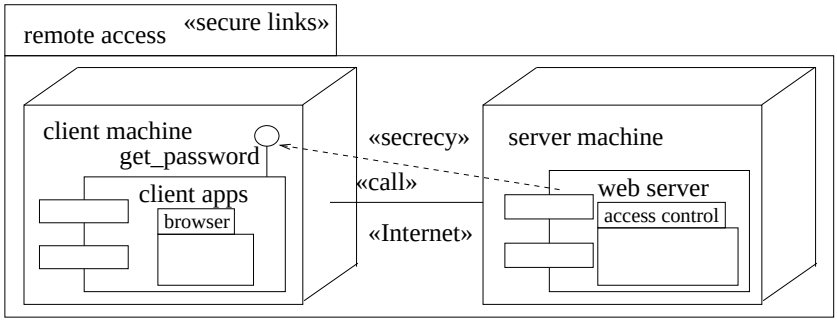

Fig. 2. Example *secure links* usage

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: « call » or « send » dependencies in object or component diagrams stereotyped « secrecy » (resp. « integrity ») are supposed to provide secrecy (resp. integrity) for the data that is sent along 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} and {integrity} 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}) in C if and only if it does so in D.
- If a message name in I appears in the tag {secrecy} (resp. {integrity}) in C then the dependency is stereotyped « secrecy » (resp. « integrity »).

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 3 shows a key generation subsystem stereotyped with the requirement « secure dependency ». The given specification violates the constraint

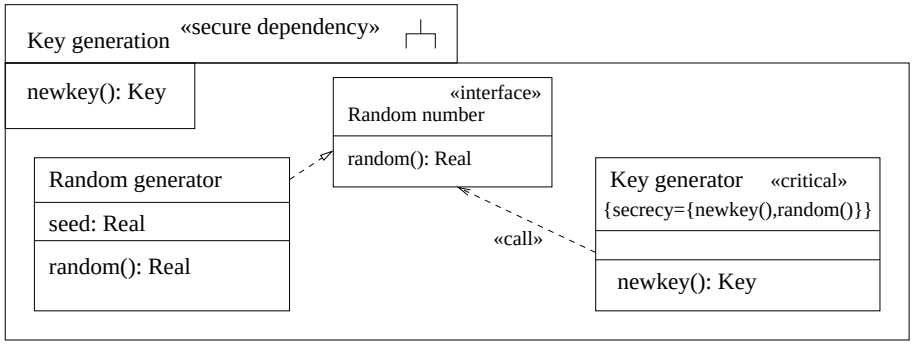

Fig. 3. Key generation subsystem

for this stereotype, since the 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 secret by Key generator (by the tag {secrecy} in Key generator), but it is not guaranteed to be secret by Random generator (in fact there are no secret messages in Random generator and so the tag {secrecy} is missing), and also the communication dependency is not guaranteed to preserve secrecy. Note that the {secrecy} tag is used at the key generator to both require and assure secrecy (for the random() method called at the random generator resp. the newkey() method offered by the key generator).

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 include {secrecy} and {integrity}. Their values 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. 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).

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 conditions that 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.

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 Threats() function in a suitable way).

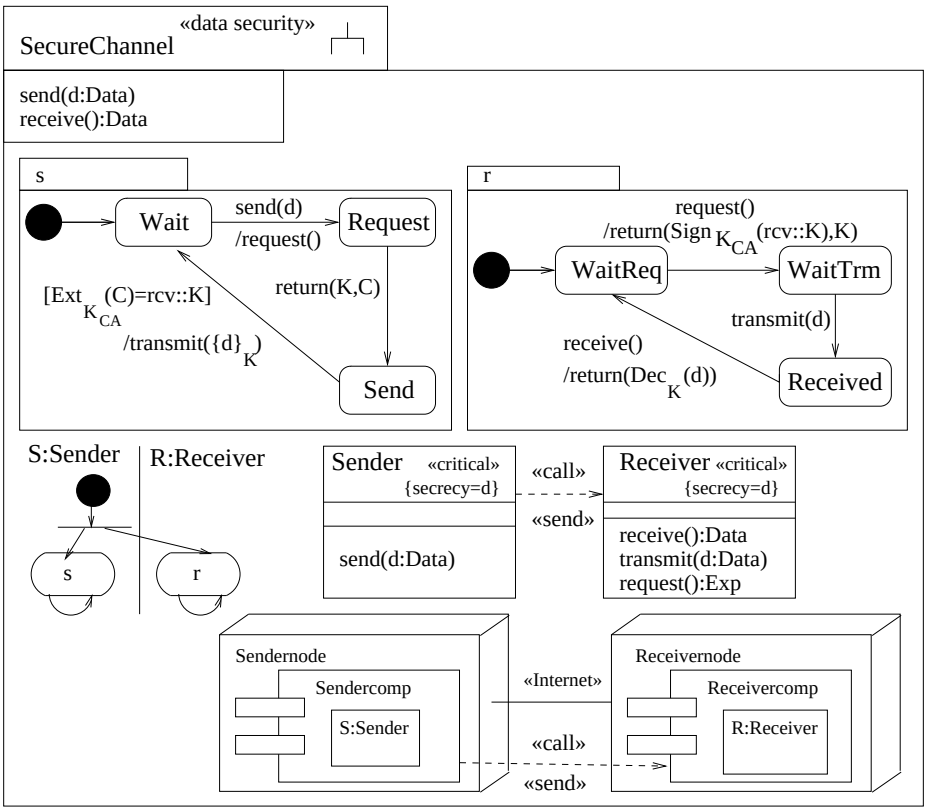

Fig. 4. Security protocol

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. 4 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 {secrecy} and {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 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).

4 Introducing Dynamic Security Aspects

Aspects encapsulate properties (often non-functional ones) which crosscut a system, and we use transformations of UML models to "weave in" dynamic security aspects on the model level. The resulting UML models can be analyzed as to whether they actually satisfy the desired security requirements using automated tools [Jür05b]. Secondly, one should make sure that the code constructed from the models (either manually or by code generation) still satisfies the security requirements shown on the model level. This is highly non-trivial, for example because different aspects may be woven into the same system which may interfere on the code level in an unforeseen way. To achieve it, one has in principle two options: One can either again verify the generated code against the desired security requirements, or one can prove that the code generation preserves the security requirements fulfilled on the model level. Although the second option would be conceptually more satisfying, a formal verification of a code generator of industrially relevant strength seems to be infeasible for the foreseeable future. Also, in many cases, completely automated code generation may not be practical anyway. We therefore followed the first option and extended our UML security analysis techniques from [UML04] to the code level (presently C code, while the analysis of Java code is in development). The analysis approach (of which early ideas were sketched in [JH05]) now takes the generated code and automatically verifies it against the intended security requirement, which has been woven in as dynamic aspects. This is explained in Sect. 5. This verification thus amounts to a *translation validation* of the weaving and code construction process. Note that performing the analysis both at the model and the code level is not overly redundant: the security analysis on the model level has the advantage that problem found can be corrected earlier when this requires less effort, and the security analysis on the code level is still necessary as argued above. Also, in practice

generated code is very rarely be used without any changes, which again requires verification on the code level.

The model transformation resulting from the "weaving in" of a dynamic security aspect p corresponds to a function f_p which takes a UML specification $\mathcal{S}$ and returns a UML specification, namely the one obtained when applying p to $\mathcal{S}$. Technically, such a function can be presented by defining how it should act on certain subsystem instances[3], and by extending it to all possible UML specifications in a compositional way. Suppose that we have a set S of subsystem instances such that none of the subsystem instances in S is contained in any other subsystem instance in S. Suppose that for every subsystem instance $\mathcal{S} \in S$ we are given a subsystem instance $f_p(\mathcal{S})$. Then for any UML specification $\mathcal{U}$, we can define $f_p(\mathcal{U})$ by substituting each occurrence of a subsystem instance $\mathcal{S} \in S$ in $\mathcal{U}$ by $f_p(\mathcal{S})$. We demonstrate this by an example.

We consider the data secrecy aspect in the situation of communication over untrusted networks, as specified in Fig. 5. In the subsystem, the Sender object is supposed to accept a value in the variable d as an argument of the operation send and send it over the « encrypted » Internet link to the Receiver object, which delivers the value as a return value of the operation receive. According to the stereotype « critical » and the associated tag {secrecy}, the subsystem is supposed to preserve the secrecy of the variable d.

A well-known implementation of this aspect is to encrypt the traffic over the untrusted link using a key exchange protocol. As an example, we consider a simplified variant of the handshake protocol of the Internet protocol TLS in Fig. 6. This can be seen as a refinement of the generic scenario in Fig. 5 and is a similar but different protocol from the one in Fig. 4. The notation for the cryptographic algorithms was defined in Sect. 2.2.

The goal of the protocol is to let a sender send a secret over an untrusted communication link to a receiver in a way that provides secrecy, by using symmetric session keys.[4] The sender S initiates the protocol by sending the message $\mathsf{request}(\mathsf{N}, \mathsf{K_S}, \mathcal{S}ign_{\mathsf{K_S}^{-1}}(\mathsf{S} :: \mathsf{K_S}))$ to the receiver R. If the condition $[\mathbf{snd}(\mathcal{E}xt_{\mathsf{K}'}(\mathsf{c_S}))=\mathsf{K}']$ holds, where K' and $\mathsf{c_S}$ are the second and third arguments of the message received earlier (that is, if the key K_S contained in the signature matches the one transmitted in the clear), R sends the return message $\mathsf{return}\big(\{\mathcal{S}ign_{\mathsf{K_R}^{-1}}(\mathsf{K} :: \mathsf{N}')\}_{\mathsf{K}'}, \mathcal{S}ign_{\mathsf{K_{CA}}^{-1}}(\mathsf{R} :: \mathsf{K_R})\big)$ back to S (where N' is the first argument of the message received earlier). Then if the condition

$$[\mathbf{fst}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_R}))=\mathsf{R} \wedge \mathbf{snd}(\mathcal{E}xt_{\mathsf{K}''}(\mathcal{D}ec_{\mathsf{K_S}^{-1}}(\mathsf{c_k})))=\mathsf{N}]$$

holds, where $\mathsf{c_R}$ and $\mathsf{c_k}$ are the two arguments of the message received by the sender, and $\mathsf{K}'' ::= \mathbf{snd}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_R}))$ (that is, the certificate is actually for R and the correct nonce is returned), S sends $\mathsf{transmit}(\{\mathsf{d}\}_\mathsf{k})$ to R, where $\mathsf{k} ::=$

[3] Although one could also define this on the type level, we prefer to remain on the instance level, since having access to instances gives us more fine-grained control.

[4] Note that in this simplified example, which should mainly demonstrate the idea of dynamic security aspect weaving, authentication is out of scope of our considerations.

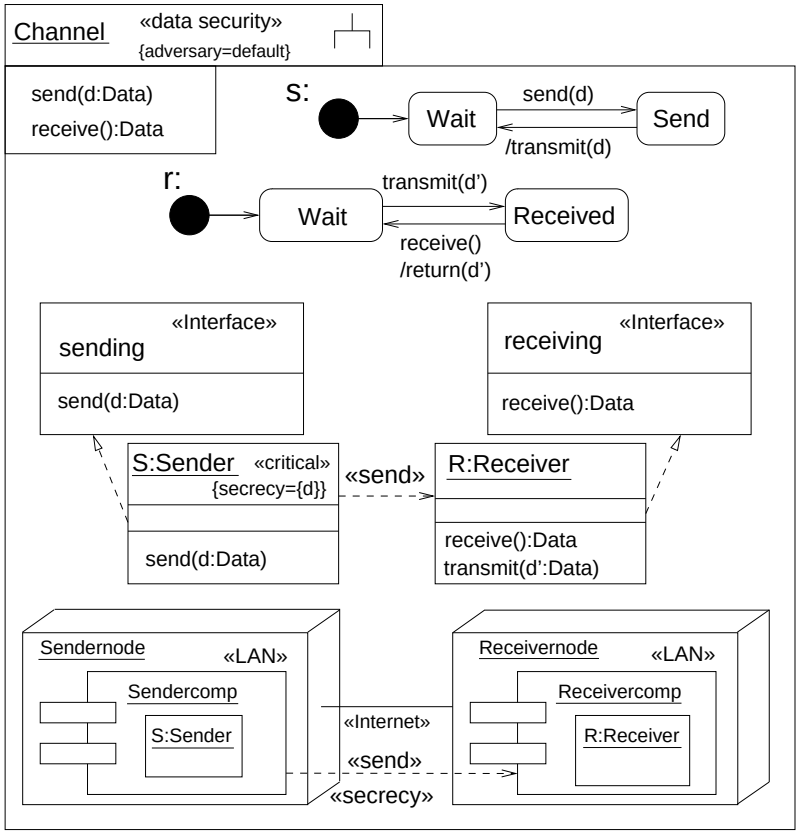

Fig. 5. Aspect weaving example: sender and receiver

$\mathbf{fst}(\mathcal{E}xt_{\mathsf{K}''}(\mathcal{D}ec_{\mathsf{K}_S^{-1}}(\mathsf{c_k})))$. If any of the checks fail, the respective protocol participant stops the execution of the protocol.

Note that the receiver sends two return messages - the first matches the return trigger at the sender, the other is the return message for the receive message with which the receiver object was called by the receiving application at the receiver node.

To weave in this aspect p in a formal way, we consider the set S of subsystems derived from the subsystem in Fig. 5 by renaming: This means, we substitute any message, data, state, subsystem instance, node, or component name n by a name m at each occurrence, in a way such that name clashes are avoided. Then f_p maps any subsystem instance $\mathcal{S} \in S$ to the subsystem instance derived from that given in Fig. 6 by the same renaming. This gives us a presentation of f_p from which the definition of f_p on any UML specification can be derived as indicated above.

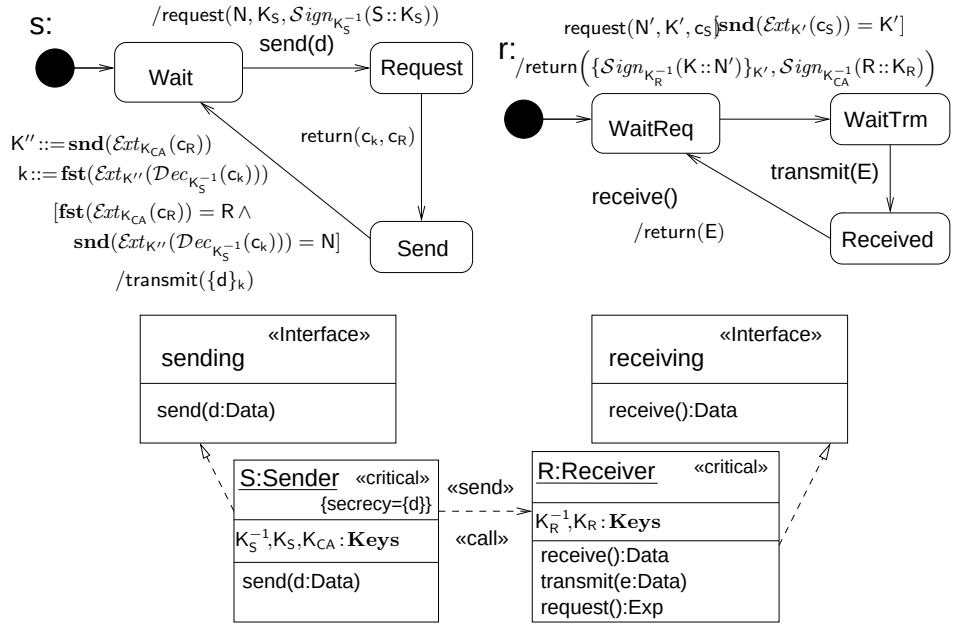

Fig. 6. Aspect weaving example: secure channel

One can do the weaving by defining the transformation explained above using the model transformation framework BOTL developed at our group [BM03]. The overall tool-suite supporting our aspect-oriented modeling approach is given in Fig. 7. The tool-flow proceeds as follows. The developer creates a primary UML model and stores it in the XMI file format. The static checker checks that the security aspects formulated in the static views of the model are consistent. The dynamic checker weaves in the security aspects with the dynamic model. One can then verify the resulting UML model against the security requirements using the analysis engine (an automated theorem prover for first-order logic). One then constructs the code and also verify it against the security requirements using the theorem prover. The error analyzer uses the information received from the static and dynamic checkers to produce a text report for the developer describing the problems found, and a modified UML model, where the errors found are visualized.

5 Analyzing the Code

We define the translation of security protocol implementations to first-order logic formulas which allows automated analysis of the source code using automated first-order logic theorem provers. The source code is extracted as a control flow graph using the aiCall tool [Abs04]. It is compiled to first-order logic axioms

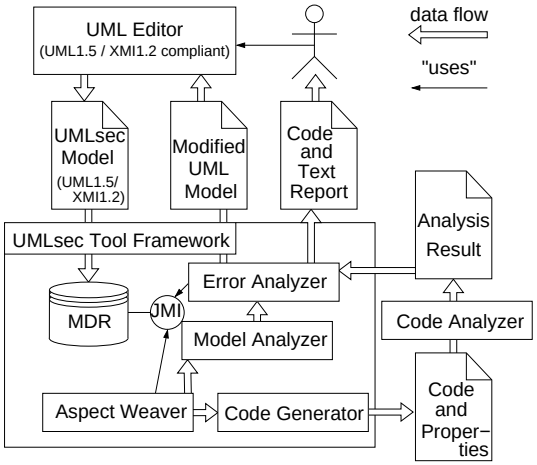

Fig. 7. UML verification framework: usage

giving an abstract interpretation of the system behavior suitable for security analysis following the well-known Dolev-Yao adversary model [DY83]. The idea is that an adversary can read messages sent over the network and collect them in his knowledge set. He can merge and extract messages in the knowledge set and can delete or insert messages on the communication links. The security requirements are formalized with respect to this adversary model. For example, a data value remains secret from the adversary if it never appears in the knowledge set of the adversary. Our approach works especially well with nicely structured code, such as that obtained using our aspect weaving tool from the previous section. For example, we apply an automated transformation which abstracts from pointers before applying our security analysis (which is a standard technique in C code verification, see e.g. [DHH02]).

We explain the transformation from the control flow graph generated from the C program to first-order logic, which is given as input to the automated theorem prover. For space restrictions, we restrict our explanation to the analysis for secrecy of data. The idea here is to use a predicate knows which defines a bound on the knowledge an adversary may obtain by reading, deleting and inserting messages on vulnerable communication lines (such as the Internet) in interaction with the protocol participants. Precisely, knows(E) means that the adversary may get to know E during the execution of the protocol. For any data value s supposed to remain confidential, one thus has to check whether one can derive knows(s). Here we use the algebra of cryptographic expressions defined in Sect. 2.2.

The set of predicates defined to hold for a given program is defined as follows. For each publicly known expression E, the statement knows(E) is derived. To model the fact that the adversary may enlarge his set of knowledge by con-

structing new expressions from the ones he knows, including the use of cryptographic operations, formulas are generated which axiomatize these operations.

We now define how a control flow graph generated from a C program gives rise to a logical formula characterizing the interaction between the adversary and the protocol participants. We observe that the graph can be transformed to consist of transitions of the form $\mathsf{trans}(\mathsf{state}, \mathsf{inpattern}, \mathsf{condition}, \mathsf{action}, \mathsf{truestate})$, where $\mathsf{inpattern}$ is empty and $\mathsf{condition}$ equals true where they are not needed, and where action is a logical expression of the form $\mathsf{localvar} = \mathsf{value}$ resp. $\mathsf{outpattern}$ in case of a local assignment resp. output command (and leaving it empty if not needed). If needed, there may be additionally another transition corresponding to the negation of the given condition, where we safely abstract from the negated condition (for logical reasons beyond this exposition).

Now assume that the source code gives rise to a transition $\mathsf{TR1} = \mathsf{trans}(\mathsf{s1}, \mathsf{i1}, \mathsf{c1}, \mathsf{a1}, \mathsf{t1})$ such that there is a second transition $\mathsf{TR2} = \mathsf{trans}(\mathsf{s2}, \mathsf{i2}, \mathsf{c2}, \mathsf{a2}, \mathsf{t2})$ where $\mathsf{s2} = \mathsf{t1}$. If there is no such transition $\mathsf{TR2}$, we define $\mathsf{TR2} = \mathsf{trans}(\mathsf{t1}, [], \mathsf{true}, [], \mathsf{t1})$ to simplify our presentation, where $[]$ is the empty input or output pattern. Suppose that $\mathsf{c1}$ is of the form $\mathsf{cond}(\mathsf{arg}_1, \ldots, \mathsf{arg}_n)$. For $\mathsf{i1}$, we define $\bar{\mathsf{i1}} = \mathsf{knows}(\mathsf{i1})$ in case $\mathsf{i1}$ is non-empty and otherwise $\bar{\mathsf{i1}} = \mathsf{true}$. For $\mathsf{a1}$, we define $\bar{\mathsf{a1}} = \mathsf{a1}$ in case $\mathsf{a1}$ is of the form $\mathsf{localvar} = \mathsf{value}$ and $\bar{\mathsf{a1}} = \mathsf{knows}(\mathsf{outpattern})$ in case $\mathsf{a1} = \mathsf{outpattern}$ (and $\bar{\mathsf{a1}} = \mathsf{true}$ in case $\mathsf{a1}$ is empty). Then for $\mathsf{TR1}$ we define the following predicate:

$$\mathsf{PRED}(\mathsf{TR1}) \equiv \bar{\mathsf{i1}} \& \mathsf{c1} \Rightarrow \bar{\mathsf{a1}} \& \mathsf{PRED}(\mathsf{TR2}) \tag{1}$$

The formula formalizes the fact that, if the adversary knows an expression he can assign to the variable $\mathsf{i1}$ such that the condition $\mathsf{c1}$ holds, then this implies

```
void TLS_Client (char* secret)
{  char Resp_1 [MSG_MAXLEN];
   char Resp_2 [MSG_MAXLEN];
   // allocate and prepare buffers
   memset (Resp1, 0x00, MSG_MAXLEN);
   memset (Resp2, 0x00, MSG_MAXLEN);
   // C->S: Init
   send (n, k_c, sign(conc(c, k_c), inv(k_c)));
   // S->C: Receive Server's respond
   recv (Resp_1, Resp_2);
   // Check Guards
   if ( (memcmp(fst(ext(Resp_2, k_ca)), s, MSG_MAXLEN) == 0) &&
         (memcmp(snd(ext(dec(Resp_1, inv(k_c)),
            snd(ext(Resp_2, k_ca)))), n, MSG_MAXLEN) == 0) )
   {  // C->S: Send Secret
       send (symenc(secret, fst(ext(dec(Resp_1,
        inv(k_c)), snd(ext(Resp_2, k_ca)))))); }}
```

Fig. 8. Fragment of abstracted client code

```
input_formula(protocol,axiom,(
  ![Resp_1, Resp_2] : (
    ((knows(conc(n, conc(k_c,sign(conc(c,conc(k_c,eol)),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(enc(secret,fst(ext(dec(Resp_1,inv(k_c)),
                                     snd(ext(Resp_2,k_ca))))))))).
```

Fig. 9. Core protocol axiom for client

that $\bar{a1}$ will hold according to the protocol, which means that either the equation localvar = value holds in case of an assignment, or the adversary gets to know outpattern, in case it is send out in a1. Also then the predicate for the succeeding transition TR2 will hold.

To construct the recursive definition above, we assume that the control flow graph is finite and cycle-free. As usual in static code analysis, loops are unfolded over a number of iterations provided by the user. The predicates PRED(TR) for all such transitions TR are then joined together using logical conjunctions and closed by forall-quantification over all free variables contained. It is interesting to note that the resulting formula is a Horn formula. Previous investigations of the interplay between Horn formulas and control flow have been done in [dBKPR89], although with a different motivation.

Figure 8 gives a simplified C implementation of the client side of the TLS variant considered earlier. From this, the control flow graph is generated automatically. The main part of the transformation of the client to the e-SETHEO input format TPTP is given in Fig. 9. We use the TPTP notation for the first-order logic formulas, which is the input notation for many automated theorem provers including the one we use (e-SETHEO). Here & means logical conjunction and ![E1, E2] forall-quantification over E1, E2. The protocol itself is expressed by a for-all quantification over the variables which store the message arguments received.

Given this translation of the C code to first-order logic, one can now check using the automated theorem prover that the code constructed from the UMLsec aspect model still satisfies the desired security requirements. For example, if the prover can derive knows(secret) from the formulas generated by the protocol, the adversary may potentially get to know secret. Details on how to perform this analysis given the first-order logic formula are explained in [Jür05b] and on how to use this approach to analyze crypto-based Java implementations in [Jür06].

6 Industrial Application

We have applied our method in several project with industrial partners. In one of them, the goal was the correct development of a security-critical biometric authentication system which is supposed to control access to a protected resource.

Because the correct design of such cryptographic 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. Our approach has been applied at the specification level in [Jür05b] where several severe security flaws had been found. We have also applied the approach presented here to the source-code level for a prototypical implementation we constructed from the specification [Jür05a]. The security analaysis results achieved so far are obtained with the automated theorem prover within less than a minute computing time on an AMD Athlon processor with 1533 MHz. tact frequency and 1024 MB RAM.

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). Work on logical foundations for object-oriented design in general and UML in particular includes for example [FELR98, HS03, ABdBS04, dBBSA04, OGO04, FSKdR05]. There has been a lot of work on formal methods for secure systems, for an overview we have to refer to [Jür04].

[HLN04] represents threats as crosscutting concerns to help determining the effect of security requirements on functional requirements. The approach analysis the interaction between assets and functional requirements to expose vulnerabilities to security threats against the security requirements. In [FRGG04], aspect models are used to describe crosscutting solutions that address quality or non-functional concerns on the model level. It is explained how to identify and compose multiple concerns, such as security and fault tolerance, and how to identify and solve conflicts between competing concerns.

A more complete discussion of related work has to be omitted for space reasons but can be found in [Jür04].

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.

In this tutorial exposition, we concentrated on an approach to develop and analyze security-critical specifications and implementations using aspect-oriented modeling. 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. We also explained how to analyze the source code resulting in the aspect-oriented development approach from the UMLsec diagrams against security requirements with respect to its dynamic behavior, using automated theorem provers for first-order logic.

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].

Acknowledgements The research summarized in this chapter has benefitted from the help of too many people to be able to include here; they are listed in [Jür04]. Helpful comments from the reviewers to improve the presentation are gratefully acknowledged.

References

[ABdBS04] E. Ábráham, M.M. Bonsangue, F.S. de Boer, and M. Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In *ICTAC 2004*, pages 37–51, 2004.

[Abs04] AbsInt. aicall. http://www.aicall.de/, 2004.

[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 *LNCS*, pages 82–94. Springer, 2001.

[BM03] P. Braun and F. Marschall. The BOTL tool. http://www4.in.tum.de/~marschal/botl, 2003.

[BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems.* Springer, 2001.

[dBBSA04] F.S. de Boer, M.M. Bonsangue, M. Steffen, and E. Ábráham. A fully abstract semantics for UML components. In *FMCO 2004*, pages 49–69, 2004.

[dBKPR89] F.S. de Boer, J.N. Koek, C. Palamidessi, and J.J.M.M. Rutten. Control flow versus logic: a denotational and a declarative model for guarded Horn clauses. In *MFCS 1989*, pages 165–176, 1989.

[DHH02] D. Dams, W. Hesse, and G.J. Holzmann. Abstracting C with abC. In *CAV 2002*, pages 515–520, 2002.

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

[FELR98] R. B. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19:325–334, 1998.

[FRGG04] R.B. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings - Software*, 151(4):173–186, 2004.

[FSKdR05] H. Fecher, J. Schönborn, M. Kyas, and W.P. de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In *ICFEM 2005*, pages 52–65, 2005.

[Gur00] Y. Gurevich. Abstract state machines. In T. Rus, editor, *8th International Conference on Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *LNCS*. Springer, 2000.

[HLN04] C. Haley, R. Laney, and B. Nuseibeh. Deriving security requirements from crosscutting threat descriptions. In *3rd International Conference on Aspect Oriented Software Development (AOSD'04)*. ACM, 2004.

[HS03] Ø. Haugen and K. Stølen. STAIRS – steps to analyze interactions with refinement semantics. In P. Stevens, editor, *The Unified Modeling Language (UML 2003)*, volume 2863 of *LNCS*, pages 388–402. Springer, 2003. 6th International Conference.

[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)*, LNCS. Springer, 2005.

[Jür01] J. Jürjens. Towards development of secure systems using UMLsec. In H. Hußmann, editor, *4th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *LNCS*, pages 187–200. Springer, 2001.

[Jür02] J. Jürjens. UMLsec: Extending UML for secure systems development. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *5th International Conference on the Unified Modeling Language (UML 2002)*, volume 2460 of *LNCS*, pages 412–425. Springer, 2002.

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

[Jür05a] J. Jürjens. Code security analysis of a biometric authentication system using automated theorem provers. In *21st Annual Computer Security Applications Conference (ACSAC 2005)*. IEEE, 2005.

[Jür05b] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *27th International Conference on Software Engineering (ICSE 2005)*. IEEE, 2005.

[Jür06] J. Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In S. Easterbrook and S. Uchitel, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. ACM, 2006.

[OGO04] Iu. Ober, S. Graf, and Il. Ober. Validation of UML models via a mapping to communicating extended timed automata. In *SPIN 2004*, pages 127–145, 2004.

[UML03] UML Revision Task Force. OMG UML Specification v. 1.5. OMG Document formal/03-03-01. Available at http : //www.omg.org/uml, March 2003.

[UML04] UMLsec group. Security analysis tool, 2004. http://www.umlsec.org.