Model-Based Design and Analysis of Permission-Based Security

Jan Jürjens, Markus Lehrhuber, and Guido Wimmel
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
{juerjens, lehrhube, wimmel}@in.tum.de

Abstract

To guarantee the security of computer systems, it is necessary to define security permissions to restrict the access to the systems' resources. These permissions rely on certain restrictions based on the workflows the system is designed for. It is not always easy to see if workflows and the design of the security permissions for the system fit together. We address this problem using an approach which embeds security permissions in UML models and supports model-based security analysis by providing consistency checks. The presented formal framework also prepares the ground for an automated analysis of underlying protocols for managing security-critical permissions, for example with the help of first-order logic theorem proving. We explain how the models can be securely implemented in a language such as Java.

1 Introduction

Since IT systems become more and more interconnected, they also become exposed to an increasing number of attacks. In order to develop high quality systems, it is therefore important to consider security aspects in the software development process. Security is a complex non-functional requirement which can only be guaranteed by the interaction of many parts in a system. Leaving security aspects to late stages and not considering them systematically makes their integration extremely difficult and increases the potential for the final product to contain vulnerabilities.

A commonly used security concept is permission-based access control, i.e. associating entities (e.g., users or objects) in a system with permissions and allowing an entity to perform a certain action on another entity only if it owns the necessary permissions. Designing and enforcing a correct permission-based access control policy (with respect to the general security requirements) is very hard, especially because of the complex interplay between the system entities. This is aggravated by the fact that permissions can also be delegated to other objects for actions to be performed on the delegating object's behalf.

In this paper, we present an approach for the integration of permissions into early design models, in particular for object-oriented design using the UML. We both describe static modelling aspects, where we introduce owned and required permissions and capabilities for their delegation into class diagrams, and dynamic modelling aspects. Dynamic modelling aspects are characterized by the use and delegation of permissions within an interaction of the system objects, modelled as a sequence diagram. To gain confidence in the correctness of the permission-based access control policy, we define checks for the consistency of the permission-related aspects within the static and dynamic models and between these models. The formal framework presented here prepares the ground for making use of tool-support such as presented in [7] for these checks using a translation from the UML models to the input notation of a first-order predicate logic automated theorem prover, based on a formal semantics for the used UML diagrams.

We focus on basic consistency checks that are well automatable and allow to quickly detect errors in the models. To achieve a highly trustworthy system, further analyses of the complex properties involved by the defined permissions are required, such as the analysis of the correctness of authorization chains. This is not in scope of the current paper. Besides, we assume that an object-oriented design and a set of appropriate permissions have already been determined. Methodologies for their elaboration are similarly out of scope of this paper.

For the implementation, we show a way to transfer permissions using cryptographic certificates and provide a formal analysis. We also address the realization of the specified permission-based access control policy in a concrete object oriented programming language like Java. We demonstrate our approach at the example of a model of an

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

instant message service.

Our long-term goal is to make formal techniques applicable in the context of industrial software development and thus try to unify the three approaches to software development (formal specifications, automated analysis, and UML models), a challenge discussed for example in [11]. This was motivated by a study which showed that formal methods adoption is still limited [1].

In the next section, we will give the necessary background on permission-based security in object-oriented systems. In Section 3, the above described aspects will be modelled in UML. Section 4 addresses consistency checks of the UML model, whereas Section 5 deals with security aspects of the model in an abstract way. The integration of the modelled concepts into a concrete programming language like Java is described in Section 6. We end with references to related work (Section 7) and conclude in Section 8.

2 Permission-based security in OO systems

Objects in object-oriented systems usually interact with each other in the following way: one object becomes an actor and performs an action on another (passive) object. The passive object is changed or activated by these actions. Activation means that the entity will also become an actor in order to perform actions on other entities.

In security-critical systems, it is crucial to have control over the execution of the actions. For this purpose, the execution of the actions is controlled by permissions. An actor is only allowed to initiate actions on certain objects if he owns the associated permissions.

In the context of such a model, we denote objects which own or define permissions as *permission-secured* objects. Permission-secured objects are the smallest entities on which permissions can be defined. Not every object in a system must be a permission-secured object. The permissions are attached to the actions that can be performed on an object. It is possible to define several permissions, which must all be owned for performing an action. In the following, we write the names of objects, classes and methods in *italics*, and denote permissions and corresponding model annotations in sansserif.

As an example, we consider a simple file. There is the permission-secured object *file*. The file defines two protected actions: *read*, which is protected by the permission read, and *write*, protected by the permission write and the permission read. These permissions are valid for the whole *file* object. We assume the protection of lower levels like lines or characters is not possible.

There are two types of owning permissions: there are permissions which are defined statically, but there is also the possibility to delegate permissions to other objects. Delegation is necessary to enable an activated object to fulfil the jobs an actor has given to it, but where the activated object itself does not have the necessary rights.

In this case, it should be possible for the delegate to act in the name of the actor. For this purpose, it is necessary to restrict the given permissions to the ones actually needed, to limit security risks. It is also necessary that it is always recognizable that the delegate acts in commission. Therefore re-delegation of permissions to other objects is an important issue. It is possible that the delegating object does not know the final delegate at delegation time.

An example for a re-delegation is the use of an account statement printer. The account owner wants to get his account statement and initiates the process. As he is not able to enquire the banking host system himself, he charges the account statement printer with doing this. Therefore, he gives the permission for reading the account information to the machine, in order for it to get the information on behalf of the account owner.

As we regard this example in more detail, it turns out that it is suitable to restrict this authorization because of two aspects:

- The printer should be able to make use of the authorization only once. If it is possible to use the authorization more than once, the printer can print the owner's account balance to every other customer.
- There should be a timeout, after which the authorization expires. If there is no timeout, it is possible that the printer makes use of the authorization after the customer has left the bank.

3 Security permissions in UML

To model the permission-based security aspects of a system, we must identify the permission-secured objects. The smallest entities on which actions may be executed are the objects. Thus, every object that is defined in a system may be a permission-secured object.

The next step is to define the protected actions. The usual way to change an object's state from the outside is to invoke a corresponding method. So it is necessary to treat method invocations as security-critical actions and thus to protect methods by permissions.

Another way of changing an object's state is to read or write public attributes directly. Although public attributes are not a good way of object design, reading and writing them must also be regarded as an action. While it is possible to restrict access to public variables at the modelling level of a system, it is not possible to do so in common object oriented programming languages. The best way to cope with this problem is to use only private variables in combination with *get* and *set* methods.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

Let us reconsider the file system example. In this case, we can model two objects: a *file* object and a *line* object, where the *file* object is an aggregation of many *line* objects. The only way to access the *line* objects is to use the methods of the corresponding *file* object. The methods of the *file* object are protected by permissions, whereas the methods the *line* object places at the *file* object's disposal are not. As these methods are only available to the *file* object, it is not necessary to make the *line* object a permission-secured object.

In a first step, we will look at the static description of the permissions in an system model. After that, we describe the permission-related aspects of the dynamic interaction of the system regarding the activities the system is designed for.

Static definitions First, we describe the static aspects of an integration of permission-based security into UML models. For this purpose, we consider class diagrams and deal with the following questions:

- Which classes define permission-secured objects?
- Which permissions will be assigned to these objects at instantiation time? These permissions are the same for all objects instantiated from the same class.
- Which methods (and public attributes) will be protected by permissions?
- What kinds of permissions are these?
- Which of the assigned permissions may be delegated? How can one define to which type of objects they may be delegated?

First of all, the permission-secured objects are identified by marking classes that define or own permission objects with the stereotype «permission-secured».

If an object owns certain permissions on other objects at instantiation time, this is also stated at this place. A tagged value is associated with the «permission-secured» stereotype consisting of a list of tuples structured as follows: $\{\mathsf{permission} = [(class, \mathsf{permission})]\}$. The first parameter of the tuple indicates the class on which the permission is valid. The second parameter names the permission.

Methods and public attributes to which access is restricted are marked with the stereotype «permission check» and an associated tagged value containing the list of permissions needed for access ($\{\mathsf{permission} = [\mathsf{permission}]\}$). This list is only a simple list naming the permissions. The association to classes is given by the class implementing the method or containing the attribute.

To allow objects of certain classes classified as reliable unrestricted access to particular methods and public variables, it is possible to associate a second tag to the stereotype «permission_check». The tagged value $\{\mathsf{no\_permission\_needed} = [class]\}$ indicates that objects of the named classes need no permissions for access.

Although delegation is a dynamic process, which comes into effect at execution time, at this point of view it is of interest which permissions can be delegated at all, and if so, to which class of objects these permissions may be delegated.

Classes that can delegate at least some of their permissions have the following tag: $\{\mathsf{delegation} = [(class, \mathsf{permission}, role/class)]\}$. The first two parameters name the permission which is delegated together with the class it belongs to. The third parameter names the class to which the permission can be delegated.

The last aspect to be regarded in the static class definition is inheritance. Definitions belonging to the modelling of permissions are inherited in the same way as all other definitions are inherited. Redefining a method or an attribute makes it necessary to also redefine the stereotypes and tags for permission modelling.

Now let us describe the example of the Instant Messaging Service which will be used to illustrate the definitions in the remainder of this paper.

The class diagram in Figure 1 shows the *SubscriptionClient* and the *InstantMessenger* on the client side, which define permission-secured objects. The class *SubscriptionClient* contains the permission subscribe on objects of class *SubscriptionServer* and the permission receive on objects of class *InstantMessenger*. In the model, this is reflected by the tagged value {permission=[(*SubscriptionServer*, subscribe),(*InstantMessenger*, receive)]}. The latter permission is marked for delegation to objects of class *Forwarder*. This is defined by the tag $\{\mathsf{delegation} = [(\textit{InstantMessenger}, \mathsf{receive}, [\textit{Forwarder}])]\}$.

On the server side, there are the classes *SubscriptionServer* and *Forwarder*. The class *SubscriptionServer* gets the permission forward on objects of class *Forwarder*. The access to the method *subscribe()* is guarded by the permission subscribe. This is stated by the stereotype «permission_check» and the tag $\{\mathsf{permission} = [\mathsf{subscribe}]\}$. For calling the method *checkLogin()*, the possession of the permission checkLogin is necessary.

The class *Forwarder* defines the method *forward(msg, receiver)*, which is guarded by the permission forward.

Dynamic definitions In this section, the point of view on the system changes to the modelling of interactions between the objects instantiated from the classes defined above. For that purpose, we identify and model workflows.

For workflow modelling in UML, activity diagrams are used, where activities are assigned to the objects. It is possible to depict the interaction of several objects solving one problem regarding the causal and temporal dependencies.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

For the description of used and needed permissions,we often need more detailed information than activity diagrams can offer. The main problem arises from the possibility to combine a number of single actions into one activity, which is connected with a number of objects. For coping with permissions in an automated way, you need to identify the objects communicating with each other clearly. This means that for every single action we must be able to name the sender and the receiver to coordinate the necessary permissions. This information easily gets lost when aggregating actions to activities. For this reason it is only possible to use activity diagrams to catch the workflow whereas for further use the workflow must be converted to a sequence diagram. In a sequence diagram you can identify caller and callee in every single step of communication, which allows to assign the permissions to the sent messages.

For refinement of the workflow, a sequence diagram is created, allowing to specify the connection between permissions and messages by regarding the exchange of messages between objects.

In a first step, we define which of the objects are permission-secured objects, using the same stereotype «permission-secured» as in the class diagram. To this stereotype, we attach the permissions the object owns on other objects, utilizing tagged values. These tags are defined the same way as in the class diagrams, by $\{\mathsf{permission} = [(object, \mathsf{permission})]\}$. In contrast to the class diagram, here the first parameter of the tuple means no longer a class but a concrete object on which the permission is valid. Additionally, the ability for delegation of certain permissions is stated by a tag as well ($\{\mathsf{delegation} = [(object, \mathsf{permission}, role/class)]\}$).

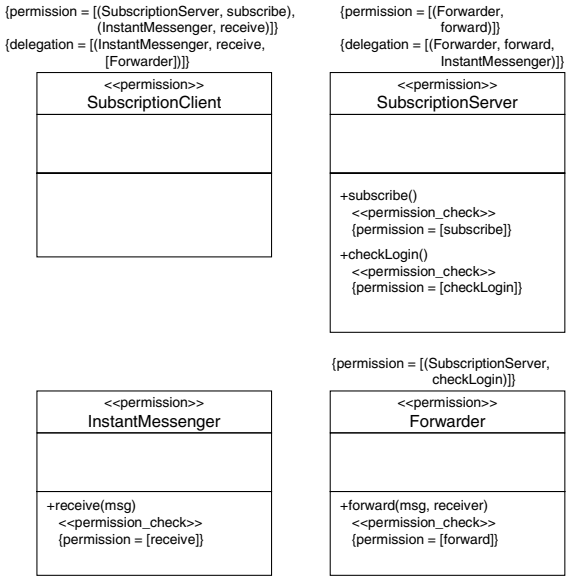

Figure 1. Class diagram for instant messager

Permissions which are needed for executing a method – or in other words for sending a message successfully – are attached directly to the message which is to be protected by these permissions. To signalize that a message is protected by permissions, the message is marked with the stereotype «permission_check», where the permissions are named as tagged values ($\{\mathsf{permission} = [\mathsf{permission}]\}$).

The delegation is performed by emitting and passing on certificates, which are formally defined as 7-tuples

$$\mathsf{certificate} = (e, d, c, o, p, x, s)$$

with emittent e, delegate d, class c of the delegate, object o, permission p which is valid on o, expiration timestamp x and sequence number s.

A certificate contains the following information:

- Who is delegating a permission? The emittent e is named in the certificate; he is signing the certificate.
- To whom is the permission to be delegated? For the definition of the delegate, there are two possibilities, depending on the relation between emittent and delegate. If the emittent knows the delegate at emission time of the certificate, he can name him explicitly (field d in the certificate). Otherwise, he can name the class c the delegate must be an instance of to make use of this certificate. In this case, d has the value *null*. In our example, the emittent never knows the delegate, thus the latter (more general) type of certificate is used.
- Which permission is to be delegated? The permission to be delegated is defined by two parameters: the permission p and the object o on which this permission is valid.
- For how long is the permission to be delegated? As it is not possible to define a contiguous time in sequence diagrams, it is also not possible to make temporal restrictions on the validity of certificates. Time will be approximated by the number of messages to be sent, starting at zero with the first message. If a certificate is valid unrestrictedly, this parameter is set to -1.
- What about the sequence number? The sequence number s is contained in the certificate to avoid that it is used several times. The sequence number of certificates which are defined by the same parameter values must differ. It is also necessary that the number is the same if a certificate is passed along several objects. For defining a certificate which might be used more than once, this parameter is to be set to -1.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

In the sequence diagram, messages where permission certificates are sent are marked by the stereotype «certification», where a 7-tuple representing a certificate is directly attached as a tagged value. The parameters of this tag correspond to the definition above.

Back to the example of the Instant Messaging Service. First of all, in the sequence diagrams in Figure 2 and Figure 3 the instances of the objects are visible with their definitions for permissions and delegation.

In Figure 2, it is shown how the *SubSender* object tries to log on at the authentication server *SubS*, by sending the message *subscribe()*. As this method expects the permission subscribe, the stereotype «permission_check» and the tag {permission = [subscribe]} are attached to this message.

The second message is a confirmation containing a permission certificate for further communication. The certificate is emitted for an object of class *InstantMessenger* and contains the permission forward on the *Forwarder* object *ForS*.

In the next step, *SubSender* initializes the *InstantMessenger* object *Sender* and sends the certificate to it for further communication.

The rest of the communication now takes place directly between *Sender* and *ForS*. Whenever *Sender* has to send a message to other *InstantMessenger* objects, he does so by calling the method *forward()* on *ForS*, for which he needs the permission forward, delegated by *SubS*.

Although the *SubReceiver* receives no certificate from *SubS*, the scenario on the receiver side is similar (cf. Figure 3). Here, *SubReceiver* delegates to *SubS* his permission receive restricted to objects of type *Forwarder*. Later, this permission certificate will be passed to the *Forwarder* object *ForS*, after it has been checked whether *SubReceiver* has logged on or not. With this permission, *ForS* can pass a message to the *Receiver* object by calling the method *receive()*.

4 Checking the UML model

Consistency between class and sequence diagrams As class diagrams and sequence diagrams are linked very closely to each other regarding the security permissions, it is necessary to check the consistency of the definitions made in these two diagrams.

In the class diagram, classes are assigned permissions on other classes. The definitions made there have to correspond to the definitions of the objects instantiated out of these class definitions. This means that objects must not have been assigned definitions that are not contained in the corresponding class definition. It is only admissible to define less permissions in the sequence diagram than in the class diagram.

The definitions for delegation are treated in a similar way, with some restrictions. In the sequence diagram, only permissions can be delegated for which this possibility is defined in the class diagram. Besides that, it is necessary that the permission which is to be delegated is present, which means that it is not only defined in the class definition, but also in the object definition.

The next thing to check is the definition of methods. The permissions needed to execute a single method are defined in the class diagram. It is necessary that these definitions fit the definitions of the sequence diagrams. The method calls are defined as messages there. Attached to these messages are the permissions which are necessary to force the receiver to execute the message in the desired way. Therefore, it is necessary that these permissions are consistent with the ones defined in the receiver's class definition.

Dynamic checking of the sequence diagram Are all permissions assigned in a system in a way that the processes modeled in the sequence diagram can be completed? This is the next question to solve. If an object should be able to send a message, it must own all permissions necessary for that action. Permissions which are assigned statically are not a problem (addressed by the consistency checks described above), but permissions assigned dynamically by delegation are:

- A permission certificate must be received before it can be used, which means both using the permission included in the certificate and passing on the certificate to other objects.
- The emittent of a certificate must be able to create the certificate. This means that he must own the permission statically and the permission must be released for delegation.
- A certificate must be valid at time of use. The loss of validity will be defined by a time stamp in the certificate.
- A certificate which is defined for being used only once looses validity by being used, so no object can use it again.

In the sequence diagram for the instant messaging service in Figure 2 you can see that the object *Sender* calls the method *forward()* of *ForS* where the permission forward is needed. As the object *Sender* does not own this permission, it is delegated by a certificate which is passed on by the message *create()*:

$$\{certificate = (SubS, null, ForS, \mathsf{forward}, InstantMessenger, -1, -1)\}$$

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 IEEE

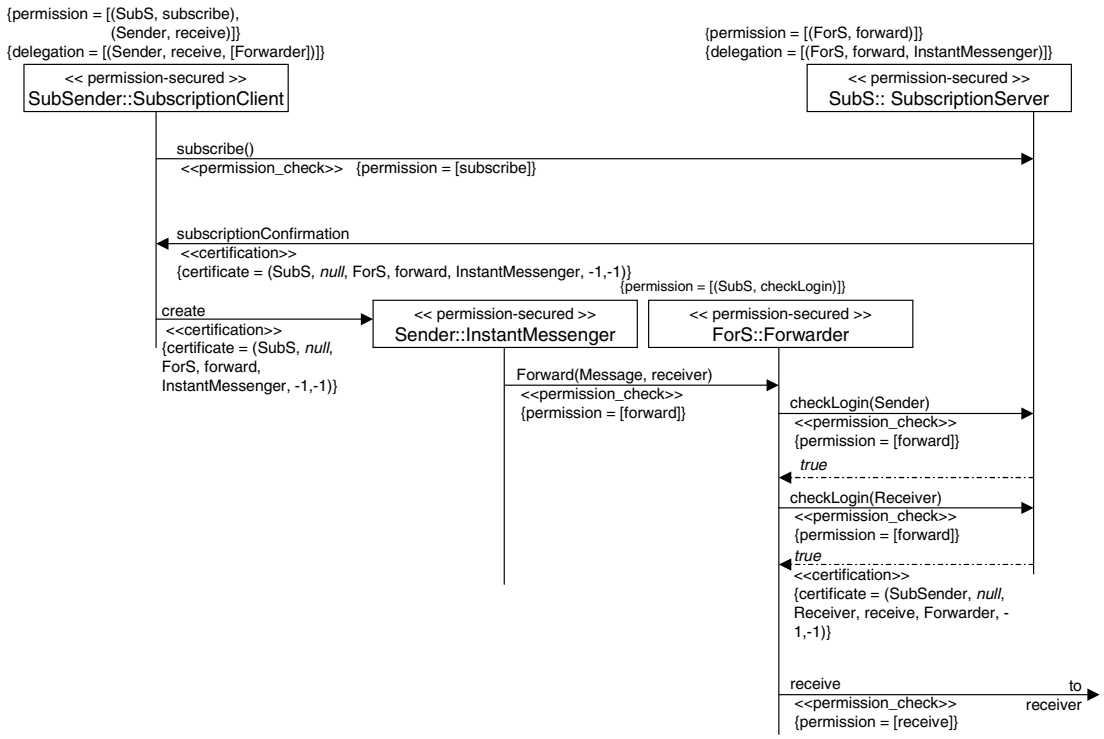

Figure 2. Sequence diagram for the instant messaging service (sender)

Because of lack of this permission, *SubSender*, the sender of this message, cannot create this certificate but must receive it from *SubS* by the message *subscriptionConfirmation()*. *SubS* owns the permission and is able to delegate it. You can see this by the tags assigned to this object:
$\{\mathsf{permission} = [(ForS, \mathsf{forward})]\}$
$\{\mathsf{delegate} = [(ForS, \mathsf{forward}, InstantMessenger)]\}$

The period of validity has not been considered in this example, because no time stamp is available. Also, the certificates may be used more than once.

Within our tool support, the dynamic checking is carried out by translating the sequence diagrams to Prolog.

5 Modelling permissions on an abstract level

A permission is a message consisting of *permission* and *identifier* (of the object the permission is valid on). The object owning the permission will be specified by appending the object's public key. Therefore it is impossible for any other object to use this permission. A *certificate* is defined as a triple consisting of the identifier followed by the permission and the public key of the user of the certificate.

For signing the permissions, there is an trusted instance in the system called security authority (SA). This instance releases all permissions and passes them on to the objects at their instantiation time. It is not possible to change the definition of a permission once signed by this authority.

Thus, a certificate defining a permission will be formally defined as follows:
Sign(identifier::permission::$K_{legitimate}$*,*$K_{SA}{}^{-1}$*).*

To enable the delegation of permissions, passing on the permission is not enough. The delegating object must issue a certificate containing the permission and restrictions for its use. In addition, the certificate contains the public key of the owner of the permission. This allows other objects to prove that this object originally was the owner of the permission. The certificate is signed with the private key of the permission's owner:

$$Sign(K_{legitimate} :: Sign(identifier :: permission :: K_{legitimate}, K_{SA}^{-1}) :: [properties], K_{legitimate}^{-1})$$

Making use of a delegated permission is only allowed for objects which are implementing the properties of the properties-list.

The next thing to cope with is the question of the security of these definitions. One possible scenario of an attack is that an intruder listens to the messages sent between the objects (man-in-the-middle-attack). So the communication underlies the following threats:

- The intruder gets rid of all messages sent between objects. He can save them for analyzing and further use.
- Messages can be deleted by the intruder, so that the receiver is not able to get a specific message.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

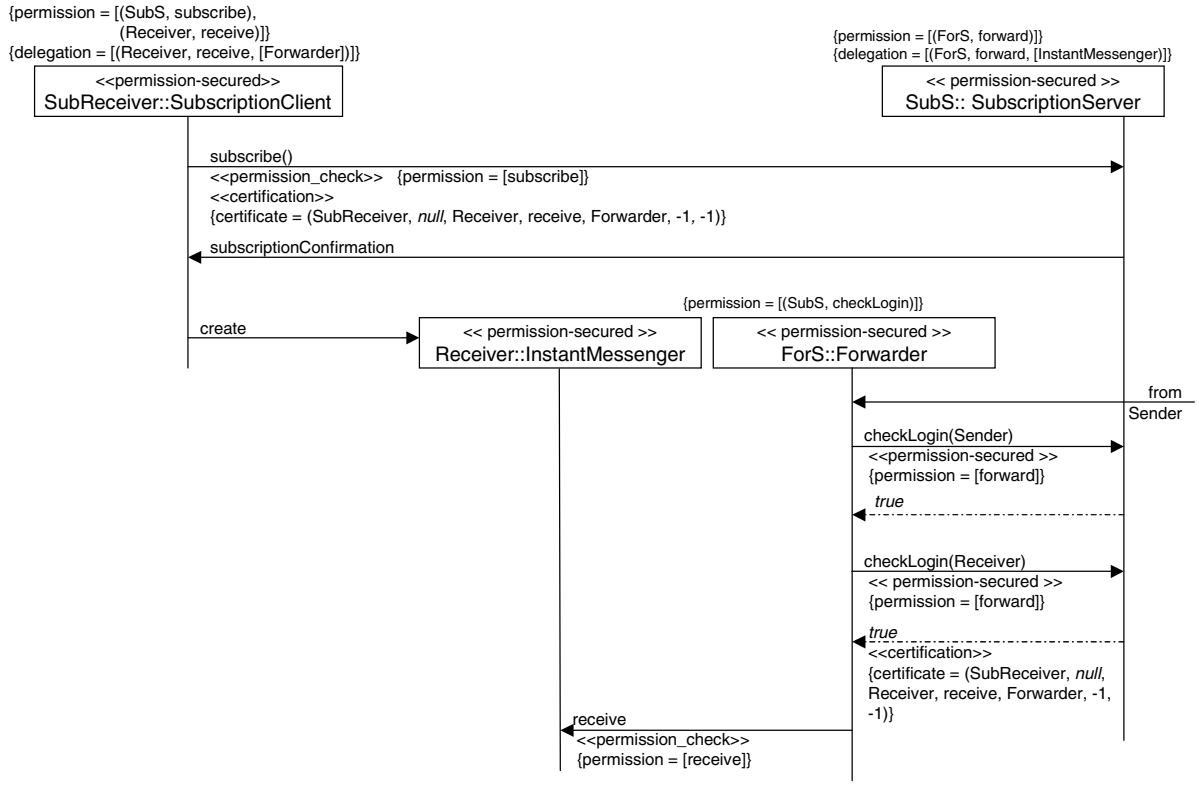

Figure 3. Sequence diagram for the instant messaging service (receiver)

- The intruder is able to insert messages into the communication between objects.
- By combination of these threats, the intruder is able to manipulate messages.

As usual, one makes use of cryptography to try to avoid such attacks by encrypting messages. In the case of security permissions it must be ensured that only the legitimate object is able to make use of a permission. Although by the definition of permissions it is guaranteed that only legitimate objects are able to create certificates for granting permissions, it is possible for intruders to obtain such a certificate in order to use the included permission. This threat can only be avoided by using an additional encryption mechanism for transmitting these certificates.

We address this problem by enhancing the UML model by cryptographic functions given in Table 1 for producing a protocol for secure communication between the objects following [6]. The security check for this protocol is done automatically using the first-order predicate logic automated theorem prover e-Setheo. For this, the protocol is converted into predicates in the TPTP-syntax following the formal semantics for UML given in [6].

We explain the modelling of such a protocol by the example of the instant messaging service. For simplification, only the communication between sender and server will be regarded. In the communication with the receiver, it is assumed that the *Forwarder ForS* obtained the permission receive on the receiver-object before using it.

In Figure 4, the corresponding sequence diagram is shown. The notation for cryptographic expressions used in this diagram is given in Table 1. For better readability, the messages contain names of functions (such as subscribe or conf), indicating their purpose. On the receiving side, the components of the received messages are referred to by A_1, A_2 and A_3 (parameters of the subscribe message), respectively by B_i, C_i, and D_i (parameters of the messages conf, init and forward). Note that the protocol in Figure 4 is only considered as an example to demonstrate our approach, not necessarily as an optimal solution for the situation at hand.

As specified in Figure 4, the object *SubSender* connects to the server *SubS* and delivers the necessary certificate *sign(conc(conc(SubS,subscribe),*K_{SC}*),inv(*K_{SA}*))* to the Server, which was signed by the security authority with key *inv(*K_{SA}*)*. It is encrypted with the public key K_{Sub} of *SubS* to ensure that only *SubS* can access the message. When *SubS* gets the message, it checks the permission and the certification of the public key. If the check is successful, a confirmation is sent back to *SubSender* that contains a permission certificate allowing an object of class *InstantMessenger* to send messages to the Forwarder *ForS*. This cer-

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

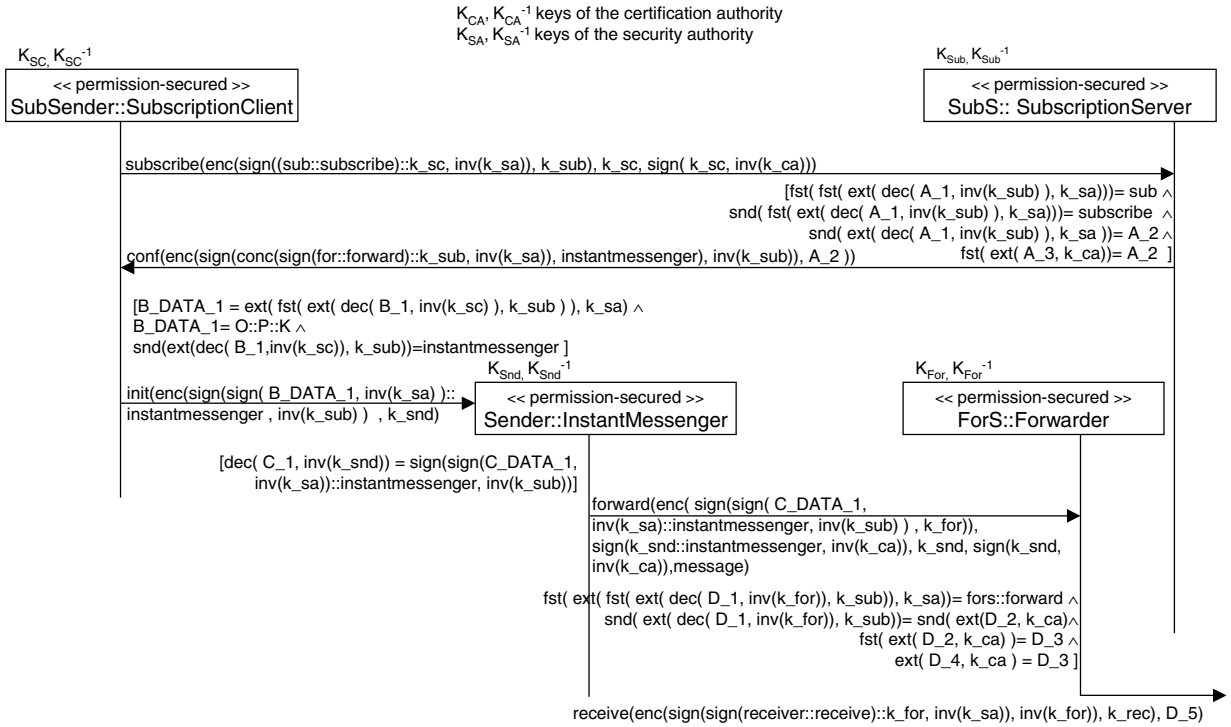

Figure 4. Security protocol

tificate consists of the following parameters:

- the permission, signed by the security authority,
- the name of the class *InstantMessenger*, so that only objects of that class are able to use the permission.

For a secure transmission, the certificate is encrypted with the public key K_{SC} of *SubSender*.

SubSender analyzes the message. It expects a permission and a restriction to the class *InstantMessenger*. If the certificate fulfills these conditions, the object *Sender* is initialized. For transmission, the certificate is encrypted with the public key K_{Snd} of *Sender*.

The *Sender* object uses this permission certificate to send a message to *ForS* in order to transmit it to *Receiver*. For transmission, the certificate to *ForS* is signed with *inv(K_{Snd})* and encrypted afterwards with K_{For}, the public key of *ForS*. The kind of class is also attested using a certificate emitted by the certification authority. This certificate will be attached to the message.

ForS checks the contained permission *conc(ForS, forward)*, and whether the sender of the message identified itself as an object of class *InstantMessenger*, by comparing the declaration in the certificate to the certificate of the certification authority. If these checks are successful, the message is passed on to the *Receiver*.

6 Integration in Java

We now explain how this permission model can be realized in a concrete object oriented programming language such as Java.

Java 2 contains a mechanism to model a security system. With Guarded-, Sealed- and SignedObjects, Java allows to guard, encrypt and sign objects. To deal with security permissions, the GuardedObject is used. In this chapter, the underlying mechanisms are presented, following [4].

The GuardedObject encapsulates the object which should be secured. A guard defines a set of permissions, which are necessary for accessing the object.

To get access to the encapsulated Object, the requesting object calls the method getObject() of the GuardedObject. In a second step, it is checked if the accessing object owns the permissions defined by the GuardObject. If it does, the method returns the reference of the encapsulated Object. The requesting object can now call any method on this object by using this reference (see Figure 5).

The Guard normally checks the permissions by using the Java AccessController. This object reads the class of which the requesting object is an instance off the execution stack. The classes are linked to their code sources and protection domains, to which the permissions are also assigned. This means in particular that all objects of the same class own the

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

same permissions. For permissions assigned at instantiation time this is certainly right, but if one wants to allow the delegation of permissions at run-time (as in our approach), this may lead to different sets of permissions for objects of the same class.

For this reason, it is necessary to enhance this method of permission checking. For delegating permissions dynamically, it is necessary that every object manages the certificates it received for delegation on its own. If such permissions should be considered, they must be given to the GuardedObject as a parameter when invoking the method getObject(). The Guard must thus be enhanced that it not only checks the static permissions but also the permissions contained in the certificates.

It must be ensured that an object cannot use "foreign" certificates to get access to another object. For that reason, the object references that the getObject() method produces may be secured by an asymmetric key.

If there is a permission to be delegated to a certain class, the relevant instance of the class will be referenced in the certificate. For checking that the caller's class and the named class in the certificate coincide, the caller's class is read from the execution stack. If there is at least one certificate which is emitted for a specific object, a reference to this object must be saved in the certificate. To check the permission, the object's public key will be requested, and the reference of the encapsulated object will be encrypted with this key.

Table 1. Notation for cryptographic expressions

inv(k)	Inverse key of *k*; a message, encrypted with key *k* can be decrypted by *inv(k)*.
sign(E,*inv(k)*)	The message E is signed with the inverse key *inv(k)*.
enc(E,*k*)	The message E is encrypted with the key *k*.
conc(E1,E2)	A message consists of two concatenated single messages E1 and E2 (also written E1::E2).
fst(E)	Inversion of conc(E1,E2); gives back the first element E1 of the concatenation.
snd(E)	Inversion of conc(E1,E2); gives back the second element E2 of the concatenation.
ext(E, k)	Extracts the message E out of a message signed message with the inverse key *inv(k)* of *k*.
dec(E, inv(k))	Decrypts the message E out of a message encrypted message with the inverse key *inv(k)* of *k*.

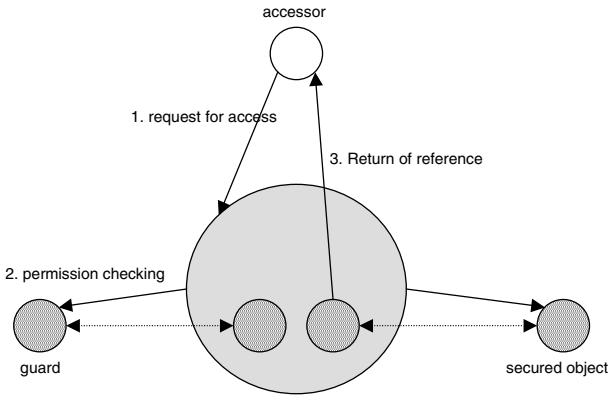

Figure 5. guardedObject

For using the reference, the caller must decode it using the corresponding private key. Since unauthorized objects do not have the appropriate private key, they cannot decode the reference.

Another problem of the Guarded Objects in Java is that the caller gets either no or complete access to an object after the permission check. To achieve restricted access to objects, we cannot give back the real reference to an object, but build a wrapper object around the encapsulated object, having only the methods the caller has the permission for calling (see Figure 6). These wrapper objects are the only ones which call the original object. This means that a wrapper class must be created for all possible combinations of methods.

Note that there is one problem not to be solved by these modifications: if one object gets the reference to an encapsulated object, the owner of the reference may pass it to unauthorized objects. This simply means that trusted objects must be developed in a trustworthy way.

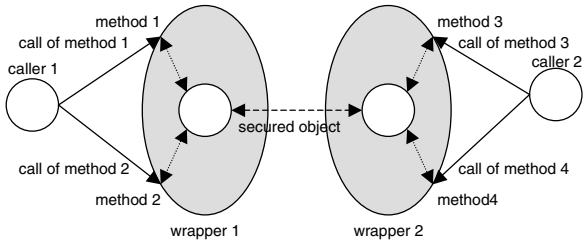

Figure 6. Wrapper objects

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

7 Related work

There are a number of approaches using UML for secure systems development, besides the one this work is based on [6, 7]. The one closest to the work presented in this paper in particular is [2, 8]. It presents work on modeling role-based access control (RBAC) using UML models in a way which allows the UML models to be parameterized. Specifications of RBAC policies are incorporated into UML design models and can thus be specified as patterns and reused. To incorporate RBAC policies into an application specific model, one instantiates the patterns and composes the instantiations with the model. Compared with our work, the paper does not give a method for automatically analyzing the access control specifications for example using an automated theorem prover, as we do. It would however be very interesting to try to join the two approaches. Another example is [3], which explains an approach which allows one to model application requirements and designs separately from security requirements and designs using UML. This approach supports a separation of concerns, in that the security requirements are captured in security use cases and encapsulated in security objects separately from the application requirements and objects. Thereby, system complexity is reduced which would arise when mixing security requirements with business application requirements.

There also exists some work on using logic to analyze object-oriented designs. For example, [9] formalizes Object Oriented Design Frameworks (OOD frameworks), which are groups of interacting objects, using computational logic. The paper uses logic programs in the context of open specification frameworks. The work also considers both static (the specification of constraints and the correctness of queries) and dynamic aspects (by introducing actions that update the current state), but does not consider security aspects.

Compared with the substantial amount of work done in the area of formal methods and security, less work has been done on security and software engineering more generally. One example which is relevant here is [10], which presents a method for modeling and analyzing information system process interactions to enforce security. The work is aimed at the area of complex systems and uses a general modeling methodology.

8 Conclusion

We presented an approach which embeds security permissions in UML models for a model based analysis. In particular, the presented formal framework prepares the ground for an automated security analysis using automated first-order logic theorem provers. We explained how the models can be securely implemented in a language such as Java.

Our approach allows one to define security permissions to restrict the access to the systems' resources based on the workflows the system is designed for. Using the associated tool, one can automatically see if workflows and the design of the security permissions for the system fit together. One can thus consider security aspects in early stages of system development in a systematic way, which decreases the potential for the final product to contain vulnerabilities. We also addressed the realization of the specified permission-based access control policy in a concrete object oriented programming language like Java. Since designing and enforcing a correct permission-based access control policy is very hard, especially in the presence of delegation, our approach is hoped to be a worthwhile contribution to the state of the art in secure software engineering. In future work, we aim to integrate our approach with other approaches to permission modeling using UML such as [8].

References

[1] R. Bloomfield, D. Craigen, F. Koob, M. Ullmann, and S.Wittmann. Formal methods diffusion: Past lessons and future prospects. In F. Koornneef and M. van der Meulen, editors, *Computer Safety, Reliability and Security (SAFECOMP 2000)*, volume 1943 of *LNCS*, pages 211–226, Rotterdam, The Netherlands, Oct. 2000.

[2] G. Georg, I. Ray, and R. B. France. Using aspects to design a secure system. In *ICECCS*, pages 117–. IEEE Computer Society, 2002.

[3] H. Gomaa and M. E. Shin. Modeling complex systems by separating application and security concerns. In *ICECCS* [5], pages 19–28.

[4] L. Gong. Inside Java 2 platform security: Architecture, API design and implementation. In *Addison-Wesley*, 1999.

[5] IEEE Computer Society. *9th International Conference on Engineering of Complex Computer Systems (ICECCS 2004), 14-16 April 2004, Florence, Italy*, 2004.

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

[7] 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.

[8] D.-K. Kim, I. Ray, R. B. France, and N. Li. Modeling role-based access control using parameterized UML models. In M. Wermelinger and T. Margaria, editors, *FASE*, volume 2984 of *LNCS*, pages 180–193. Springer, 2004.

[9] K.-K. Lau and M. Ornaghi. Correct oo systems in computational logic. In M. Bruynooghe, editor, *LOPSTR*, volume 3018 of *LNCS*, pages 34–53. Springer, 2003.

[10] P. Periorellis, O. C. Idowu, S. J. Lynden, M. P. Young, and P. András. Dealing with complex networks of process interactions: A security measure. In *ICECCS* [5], pages 29–36.

[11] B. Steffen. Major threat: From formal methods without tools to tools without formal methods. In *ICECCS* [5], page 15.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**