# An Aspect-Oriented Methodology for Designing Secure Applications

Geri Georg[1], Indrakshi Ray[1], Kyriakos Anastasakis[2], Behzad Bordbar[2], Manachai Toahchoodee[1], Siv Hilde Houmb[3]

[1] Computer Science Department
Colorado State University
Fort Collins, Colorado, USA
{georg,iray,toahchoo}@cs.colostate.edu
[2] School of Computer Science
University of Birmingham
Edgbaston, Birmingham, UK
{B.Bordbar, K.Anastasakis}@cs.bham.ac.uk
[3] Department of Computer Science
University of Twente
Enschede, Netherlands
S.H.Houmb@ewi.utwente.nl

**Abstract:** We propose a methodology, based on Aspect-Oriented Modeling (AOM), for incorporating security mechanisms in an application. The functionality of the application is described using the primary model and the attacks are specified using aspects. The attack aspect is composed with the primary model to obtain the misuse model. The misuse model describes how much the application can be compromised. If the results are unacceptable, then some security mechanism must be incorporated into the application. The security mechanism, modeled as security aspect, is composed with the primary model to obtain the security treated model. The security treated model is analyzed to give assurance that it is resilient to the attack.

**Keywords:** Alloy, aspect-oriented modeling, authentication, secure systems design, security analysis, UML

## 1. Introduction

Developing secure systems is a non-trivial task. Security standards such as the ISO Common Criteria [28] and risk management standards such as the Australian/New Zealand Risk Management standards [4, 5] exist to aid secure systems development. However, these standards generally address system security in the broad sense, and often require extensive resources and expertise to adapt their use to the design of a specific system. These standards also do not address low-level details, such as, how to verify that a system is protected from specific kinds of attacks or how to ensure that a system has a given set of security properties. More importantly, they do not provide a methodology for designing secure systems.

Security mechanisms are typically analyzed in isolation as protocols, and depending on how they are integrated in an application, they may or may not provide adequate protection. In addition, there are often multiple mechanisms that could be used to counter an attack, so choosing a mechanism that best fits design goals may be confusing. It is also the case that solutions to different security concerns may actually conflict, rendering some ineffective against the attack they were supposed to counter. System designers need a way to verify the efficacy of security mechanisms once they have been integrated into an application design, prior to implementation. They also need the ability to include solutions in combination and analyze them against various attacks. In this paper, we propose such a methodology for designing secure applications.

We use aspect-oriented modeling (AOM) techniques [20] in our approach to designing secure systems. Complex software is not developed as a monolithic unit but is decomposed into modules on the basis of functionality. We refer to the models describing functionality as the *primary model*. Security concerns are not limited to one module of the primary model but impacts several of them. For example, an attack typically affects multiple modules. Similarly, a security mechanism that thwarts an attack will have to be incorporated in several modules of the application. The attack and the security mechanisms are localized in a separate model, which we call the *aspect*. Modeling security mechanisms and attack models as aspects has several benefits -- it allows designers to understand the attacks and the mechanisms independently, which makes it easier to manage and change these models. Designers can use techniques for composing aspects with the primary model, followed by analysis of the resulting system, to understand the effect of the attack or the effect of the security mechanism on the application. Another advantage is that analyzing using different attack models or different security aspects is easier since all a designer must do is to re-compose the primary model with a new attack model or new security aspect prior to performing a new analysis.

An aspect in our work is similar to the concept of aspects used in other AOM or AOP (Aspect Oriented Programming) approaches [2, 13, 14, 31, 34, 57] in that they represent a non-functional concern, e.g. security, and they are cross cutting and must be integrated at different places in the primary model. The differences lie in how the aspects are specified, whether they are reusable, and the manner in which the aspects are integrated with the application.

We define two types of aspects: *generic aspects* and *context-specific aspects.* Generic aspects are application-independent and reusable. For instance, an attack pattern can be represented as a generic aspect. Similarly, a security protocol or a security mechanism can be modeled as a generic aspect. An application developer can create his own generic aspect or use an existing one from the library of generic aspects. Generic aspects can be independently analyzed to ensure that the properties of the attack or the mechanism have been adequately captured. Generic aspects must be instantiated in the context of a given application. The instantiation is referred to as a context-specific aspect. We use parameterized Unified Modeling Language (UML) to represent generic aspects. Context specific aspects are represented as UML models. The instantiation occurs by binding parameters in the generic aspect to elements in the primary model. Specifying aspects using UML allows our approach to be used at different levels of abstraction.

To understand the impact of a security attack on the primary model, it is necessary to compose the context-specific attack aspect with the primary model. The composition produces the *misuse model*. Analysis of the misuse model will help determine whether the protected resources are compromised by the attack. If the results are unacceptable, a security mechanism must be integrated with the primary model. We refer to this model as the *security treated model*. To understand the efficacy of the security mechanism, the security treated model is composed with the context-specific attack aspect. The result is the *security treated misuse model*. The security treated misuse model is analyzed to ensure that the given attack is mitigated in the security treated model.

Manual analysis is error-prone and tedious. Towards this end, we investigated how this analysis can be partially automated. The tools for verifying UML models, such as, OCLE [47] and USE [25], are useful when we want to check if a specific model instance conforms to the constraints of the model. Although theorem provers are effective for analyzing properties, but they require a lot of expertise and are unlikely to be used by application developers. We chose to use the Alloy Analyzer because it is easy to use and has been used for verifying many real-world applications.

We illustrate the basic operation of our approach using an example e-commerce platform called ACTIVE [17]. ACTIVE provides services for electronic purchasing of goods over the Internet. The IST EU-project CORAS performed three risk assessments of ACTIVE in the period 2000-2003. The project looked into security risks of the user authentication mechanism, secure payment mechanism, and the agent negotiation mechanisms of ACTIVE. Our example consists of the user authentication mechanism of ACTIVE's login service. In order to keep the example tractable, we only show how to apply our methodology to one of its risks and one of the possible treatments for that risk.

The paper makes several contributions. First, it provides a methodology for designing secure applications. Second, it shows how to analyze the impact of a security attack on an application and how effective the security solutions are against a given attack. Third, it allows one to compare the efficacies of the different security solutions with respect to one or more given attacks. Fourth, it shows how to formally analyze a model and get assurance about the security properties. Fifth, it demonstrates feasibility that the approach can be used for real-world applications.

The rest of the paper is organized as follows. Section 2 describes ACTIVE. Section 3 shows an example attack to the login service. We also show how to compose the attack model with the primary model to create a misuse model. Section 4 presents a security mechanism we use to prevent the attack and illustrates how we integrate it with the primary model to create a security treated model. This section also shows how we generate the misuse model for the security treated model. Section 5 shows how we can analyze this model to ensure the satisfaction of the security properties. Section 6 discusses related work. Section 7 concludes the paper with some pointers to future directions. The Appendix gives the detailed Alloy models.

## 2. Overview of Our Approach

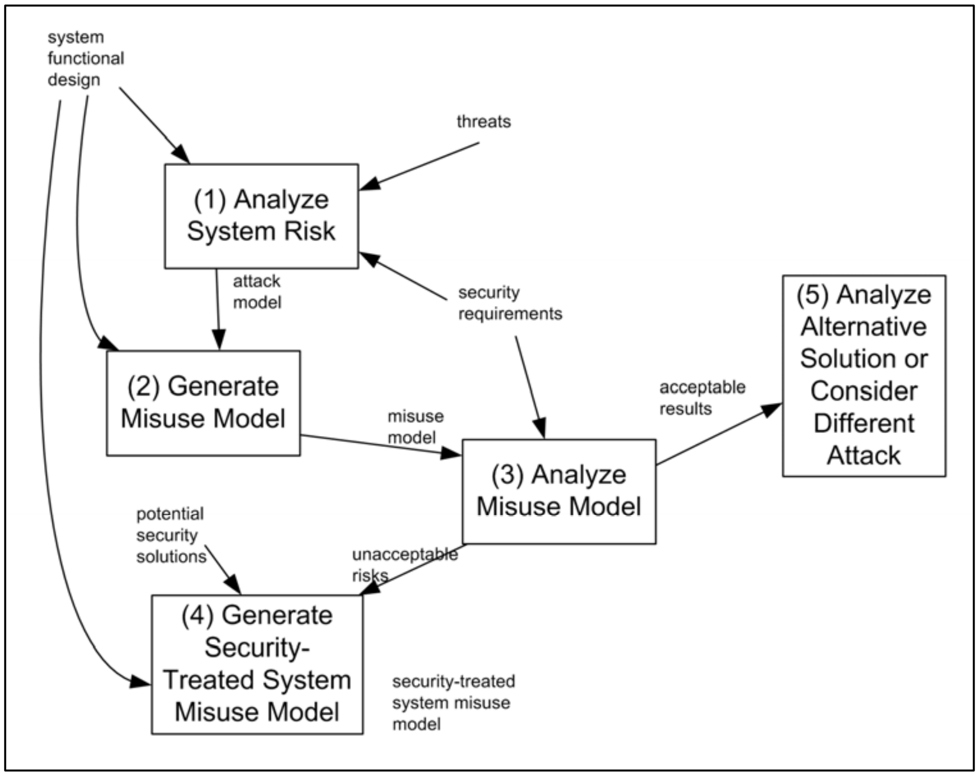
system functional design
threats
(1) Analyze System Risk
attack model
security requirements
(2) Generate Misuse Model
misuse model
(3) Analyze Misuse Model
acceptable results
(5) Analyze Alternative Solution or Consider Different Attack
potential security solutions
unacceptable risks
(4) Generate Security-Treated System Misuse Model
security-treated system misuse model

Figure 1. Secure system design methodology.

An overview of our methodology is given in Figure 1. Step (1) analyzes the system to identify the threats to the resources. The inputs to this step are the primary model, possible threats, and the security requirements. Threats become attacks on the system when they compromise protected resources. Since an attack impacts various parts of the primary model, we abstract the specification of the attack in an aspect. To distinguish them from the other aspects used in our work, we refer to them as *attack aspects*. Step (2) involves composing the attack aspects with the primary model to create *misuse* models. Step (3) analyzes the misuse model to understand the impact of the attack. If the results are not acceptable, potential security solutions (or mechanisms) that counter the attack are incorporated into the primary model to obtain the *security-treated model*. The security-treated model is combined with the specific attack to create a *security-treated misuse model*. This is done in Step 4. The security-treated misuse model is analyzed as in Step (3), and if the results are still unacceptable, an alternate security solution must be integrated, and the new security-treated system misuse model re-generated and re-analyzed. When the analysis results are acceptable, a different attack and its potential solutions can be considered. This is done in Step (5). It is important to continue integrating security mechanisms and analyzing the resulting security-treated system against previously considered attack models since some mechanisms may interfere with each other. When such conflicts arise, the designer can integrate alternative solutions until a usable combination is identified through achieving acceptable analysis results. We next discuss each step of the methodology in more details.

## Step 1: Analyze system risk.

There are many different risk analyses methodologies that can be used in the first step of the methodology, and we use the CORAS framework [15, 17, 49]. CORAS is model-based, and uses UML diagrams and textual usage scenarios as part of a risk assessment. This fits well with existing design processes since UML is the de-facto modeling language used in the software industry. CORAS takes advantage of techniques developed for the safety domain, and has a platform of supporting tools. Using CORAS, a portion of the system to be analyzed is identified as the context for the analysis, and assets associated with particular stakeholders are identified within that context. UML use case, static class, and dynamic behavior diagrams are used to specify the system design that we refer to as the primary model.

The CORAS framework use Hazard and Operability (HAZOP) analysis to identify threats to the assets of interest, and Failure Mode Effect Analysis (FMEA) to identify system vulnerabilities. It then uses Fault Tree Analysis, along with the threats and vulnerability analysis results to identify unwanted incidents that can lead to attacks on assets. The consequences and frequencies of these incidents determine the value of the risks with which they are associated. Designers prioritize risks with respect to the system security requirements, and assess potential treatments using these priorities and the risk values.

This detailed assessment identifies the context in which specific attacks could occur and the assets that could be affected. Part of the output of a CORAS analysis is therefore the exact locations in the system design that are vulnerable to attacks and the exact forms that such attacks would take. This information is used in Steps 2 and 4 of our methodology.

Designers identify treatments that can: 1) reduce the frequency of an unwanted incident, 2) reduce its consequences, 3) transfer the risk elsewhere, or 4) leave the risk unaffected. Potential treatments, like threats, are developed from a variety of sources, including experience, domain expertise, and governmental sources.

Treatments may take the form of incorporating an existing well-defined mechanism (e.g. SSL in web applications), or they may take the form of "good practices" such as proper quoting of input values to remove the possibility of database attacks in form-based web/database applications. The process of identifying threats and developing treatments is beyond the scope of this paper, however we do note that our methodology relies upon prior knowledge of potential threats, and we cannot discover previously unknown threats using these techniques.

Our methodology uses the output of the CORAS process in two ways: 1) we use UML to model unwanted incidents leading to high priority risks in what we call *attack models*, and 2) we use UML to model potential treatments to create *security aspects*. Both types of models consist of diagrams such as use cases, static, and dynamic diagrams. We add constraints written in the Object Constraint Language (OCL) to specify security properties. (OCL [44] is based on set theory and logic.)

## Step 2: Generate misuse model.

The second step in our methodology is to generate a misuse model. We create this model by instantiating a generic attack model (defined in the risk analysis step), and composing it with the primary model. The misuse model represents the system under the specific attack, and illustrates the degree to which the application can be compromised by the given attack. Please note that this model could also be a direct output of the CORAS analysis since system attacks are assessed in the context of the system. We describe creation from a generic model to make clear that other risk assessment techniques that do not directly produce system-specify attack models can also be used with our methodology.

## Step 3: Analyze misuse model.

We analyze the misuse model and compare the analysis results with the security requirements to determine whether the risk leading to the misuse is adequately mitigated by the system. Since the misuse model contains OCL constraints, rigorous formal analysis is possible. Theorem provers, model checkers, and executable models can be used to analyze dynamic behavior. In this paper, we use the Alloy Analyzer for the purpose of analyzing the models.

Analysis can demonstrate that the original functional design sufficiently protects system assets. If this is the case, a designer can compose a different attack model with the primary model to create a new misuse model ready for analysis. More often, however, the misuse model analysis results are unacceptable, and a security mechanism must be incorporated into the system to mitigate the risk to its assets.

### Step 4: Generate security-treated system misuse model.

We compose potential treatments with the primary model to create a security-treated system model. This model specifies the system in which the security mechanism has been incorporated. Instantiation of the generic security aspect and composition with the primary model use the same techniques as described above. (As mentioned in Step 2, the system-specific security aspect model could be a direct output of Step 1.) We compose the attack model with this new system model to create the security-treated system misuse model. We then analyze this model just as the original misuse model was analyzed, and we use the results to give assurance that the application is indeed resilient to the given attack.

### Step 5: Analyze alternative solution or consider different attack.

If the analysis results of the security-treated system misuse model are unacceptable, designers can incorporate a different security mechanism into the system, creating a new security-treated system model. This model can then composed with the attack model and analyzed. If the results are acceptable, designers can analyze the security-treated system model with respect to a different attack, incorporating new security mechanisms to mitigate additional risks. Once a solution is found to a new attack, designers should analyze the new security-treated system incorporating the previous attack to provide assurance that the multiple solutions do not interfere with each other, rendering one or the other ineffective.

In an ideal situation, we could automate our entire design methodology. This is particularly attractive since repeated generic aspect instantiation, composition, and analysis can be tedious and error prone. While we have been unable to automate the methodology completely, we have automated certain parts, and identified others that can be partially automated. We discuss automation in the context of our example, and give further details as to on-going work in this area in our conclusions.

## 3. Example E-Commerce System

We illustrate the reasoning about security risk mitigation with the login service of the ACTIVE e-commerce platform This example concentrates on the result from the CORAS project risk assessment of the user authentication mechanism of the login service.

We begin by creating a primary model of the login service. This model consists of both static (structural) and dynamic (behavioral) diagrams. Several classes play a part in the login process. A user who wishes to login to the e-commerce system must run an *ActiveClient* in a web browser on his or her local machine. The browser communicates with a *LoginManager* class that is located on a server across the Internet.

The *LoginManager* has several related classes. An account manager (*UAcctManager*) authenticates users using a simple user name and password provided by the client web browser. A profile manager (*UProfileManager*) keeps track of user profile information. The login service static and sequence diagrams are shown in Figure 2.

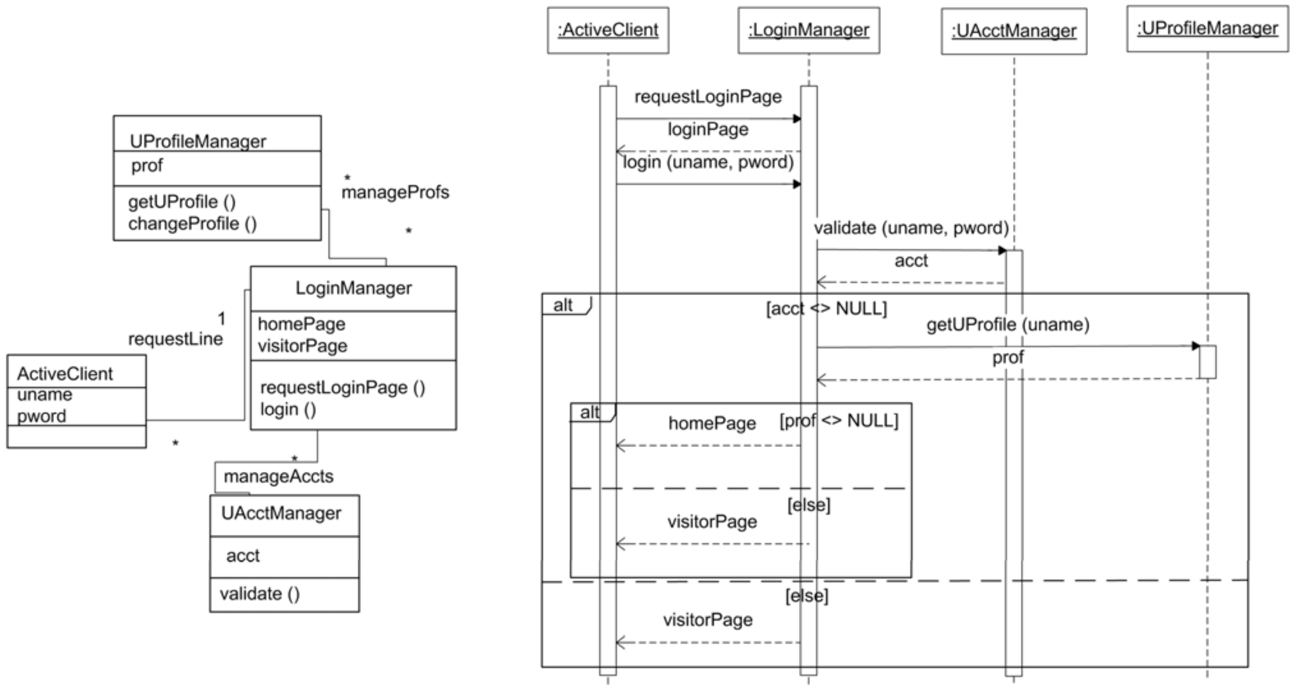

Figure 2. Primary model (E-commerce login service) static and sequence diagrams

The login operation proceeds as follows. First, a user, through a web browser (*ActiveClient*), requests a login page from the e-commerce system by sending *requestLoginPage* to *LoginManager*. *LoginManager* responds with *loginPage*. The user enters a unique user name (*uname*) and password (*pword*), and this information is sent to *LoginManager*. The server then sends *validate* message to *UAcctManager*. The *UAcctManager* returns account information (*acct*), or NULL if the user account does not exist.

If the user is authenticated (i.e. a non-NULL *acct* is returned), the *LoginManager* sends a *getUProfile* message to *UProfileManager*. The *UProfileManager* retrieves the user's profile (*prof*) and sends it to the *LoginManager*. Using this information the *LoginManager* creates an appropriate home page which is returned to the user's web browser. If the user could not be authenticated, or the user's profile could not be obtained, a visitor page is returned to the browser.

The asset that needs to be protected in this system is the user information, specifically the information in a registered user’s profile, which is returned in the *homePage*, and which is accessible anytime after a registered user has successfully logged into the ACTIVE system.

## 4. The Man-in-the-Middle Attack

The risk assessments performed as part of the CORAS project identified the login process as being vulnerable to man-in-the-middle attacks. During this kind of attack, user information can be obtained directly, or an attacker can intercept user names and passwords, to be used at later times to impersonate a valid user.

Attacks can be thought of as aspects because an attack is not confined to one specific module of the application, but impacts multiple modules. We represent these attacks as generic aspects. We represent generic aspects as patterns using UML templates. These templates must be instantiated for each application to obtain a *context-specific attack model*.

In this section, we show how to represent the man-in-the-middle attack as a *generic aspect*. Messages between a requestor and authenticator are intercepted by an attacker. This can only occur if all messages flow through the attacker and not through a direct association between the requestor and authenticator. The attacker either *intercepts* the message intended for the authenticator, or the attacker *eavesdrops* on the communication medium between the requestor and the authenticator.

In interception, the attacker must pose as the authenticator so that any message intended for the authenticator is really sent to the attacker. The attacker then relays messages between the requestor and the authenticator until the private information has been obtained by the attacker. Messages can either be passed on unchanged (*passive* attack), or the attacker can change messages prior to sending them onto the intended recipient (*active* attack).

In eavesdropping, the attacker does not impersonate the authenticator, but rather just listens to the message flow. The attacker may not obtain all of the messages flowing between the requestor and authenticator, but simply sample messages in the hopes of obtaining information. We use the active form of interception in our example, where an attacker can actually participate in complex protocols, and change messages if desired before passing them on to the requestor or authenticator. The static and sequence diagrams of a generic man-in-the-middle authentication attack model is shown in Figure 3.

The static diagram shows four classes. The |*Requestor* communicates with the |*Authenticator*, which uses the help of an authentication helper class, |*AuthHelper* to authenticate the requestor. The communication in both directions passes through an |*Attacker*. The 'X' on the '|*requestDirect*' relation indicates that a direct relationship between the |Requestor and |Authenticator classes is forbidden, and if it exists in a primary model with which this aspect is composed, it will be removed.

The sequence diagram shows all messages between the |*Requestor* and |*Authenticator* passing through the |*Attacker*. Secret information can be changed by the |*Attacker* as shown by the |*checkSecretInfo* message from the |*Requestor* to the |*Attacker*, and the |*checkSecretInfoAt* message passed on to the |*Authenticator*. This generic aspect must be instantiated to create a *context-specific* aspect that can then be composed with the primary model to create a misuse model.

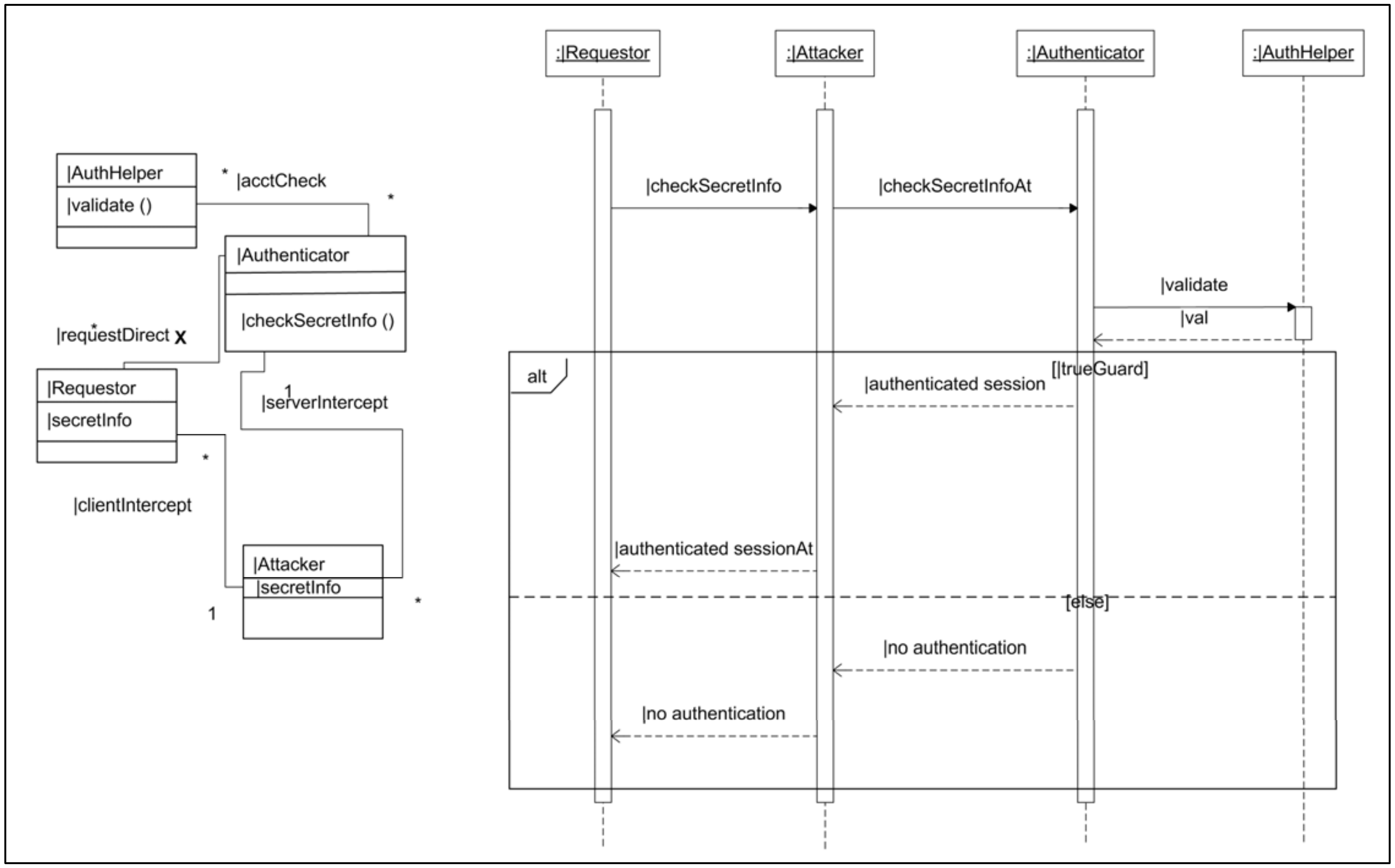

Figure 3. Generic man-in-the-middle attack model static and sequence diagrams.

## 4.1. Generating the Misuse Model

In order to understand the impact the man-in-the-middle attack has on the e-commerce application, we need to generate a misuse model. The misuse model will indicate how much the primary model can be compromised by the attack. Two steps are needed to generate the misuse model:

1. Instantiate the generic attack aspect to obtain the context-specific attack aspect.
2. Compose the context specific attack aspect with the primary model to obtain the misuse model.

Details on instantiation and composition may be found in France et al. [19, 20]. The steps outlined below are intended to provide an overview of the process.

**Instantiating the Generic Aspect:** The generic aspect shown in Figure 3 is application-independent. It is specified using UML templates. These templates must be instantiated for a given application to create a context-specific aspect. Instantiation consists of several steps: 1) determining model element correspondence, 2) creating a binding list, and 3) stamping out aspect template elements using the binding list to create model elements.

Any element in the generic aspect model that has a name beginning with the '|' character is a template parameter and can correspond to an element in the primary model that is of the same construct type. For example, in Figure 3 the '|*Requestor*' lifeline parameter in the sequence diagram can correspond to the '*ActiveClient*' lifeline in the primary model since they are the same construct types (lifelines).

Determining element correspondence is a human-involved task. A designer must determine "where" the generic aspect needs to be integrated into the primary model, and thus, which primary model elements correspond to which aspect parameters. Hints, such as identical primary model and template parameter element names or recognizing patterns in the two models, can be useful in this process, but ultimately a human must decide what parameters correspond to primary model elements. Correspondence is formalized by binding primary model element names to template parameters during aspect instantiation. We call the set of corresponding elements the binding list. It consists of pairs of element names of the form *(<template parameter name>, <primary model element name>)*, for example *(|Requestor, ActiveClient)*. Often there are parameters in a generic aspect that do not correspond to elements in the primary model. Thus, the binding list must be completed by including bindings for the rest of the aspect parameters, using names in the generic aspect with the leading '|' character removed. An example is *(|clientIntersept, clientIntercept)*.

If the CORAS framework is used to perform risk analysis, locations in the primary model where an attack could occur have been identified, and hence bindings to the primary model are also identified. In fact, a complete context-specific attack model can be created from this information as part of the risk analysis step.

The context-specific aspect pattern is then automatically constructed, creating all the elements in the generic aspect model, and substituting the aspect parameter names with their bound names from the binding list, and using the generic aspect model names for the rest of the model elements in the generic aspect. Examples of aspect templates that will simply be stamped out upon instantiation are shown in Figure 6, in the form of most of the messages comprising the TLS protocol – these messages do not contain any parameters. The context-specific aspect diagrams of the man-in-the-middle attack are shown in Figure 4.

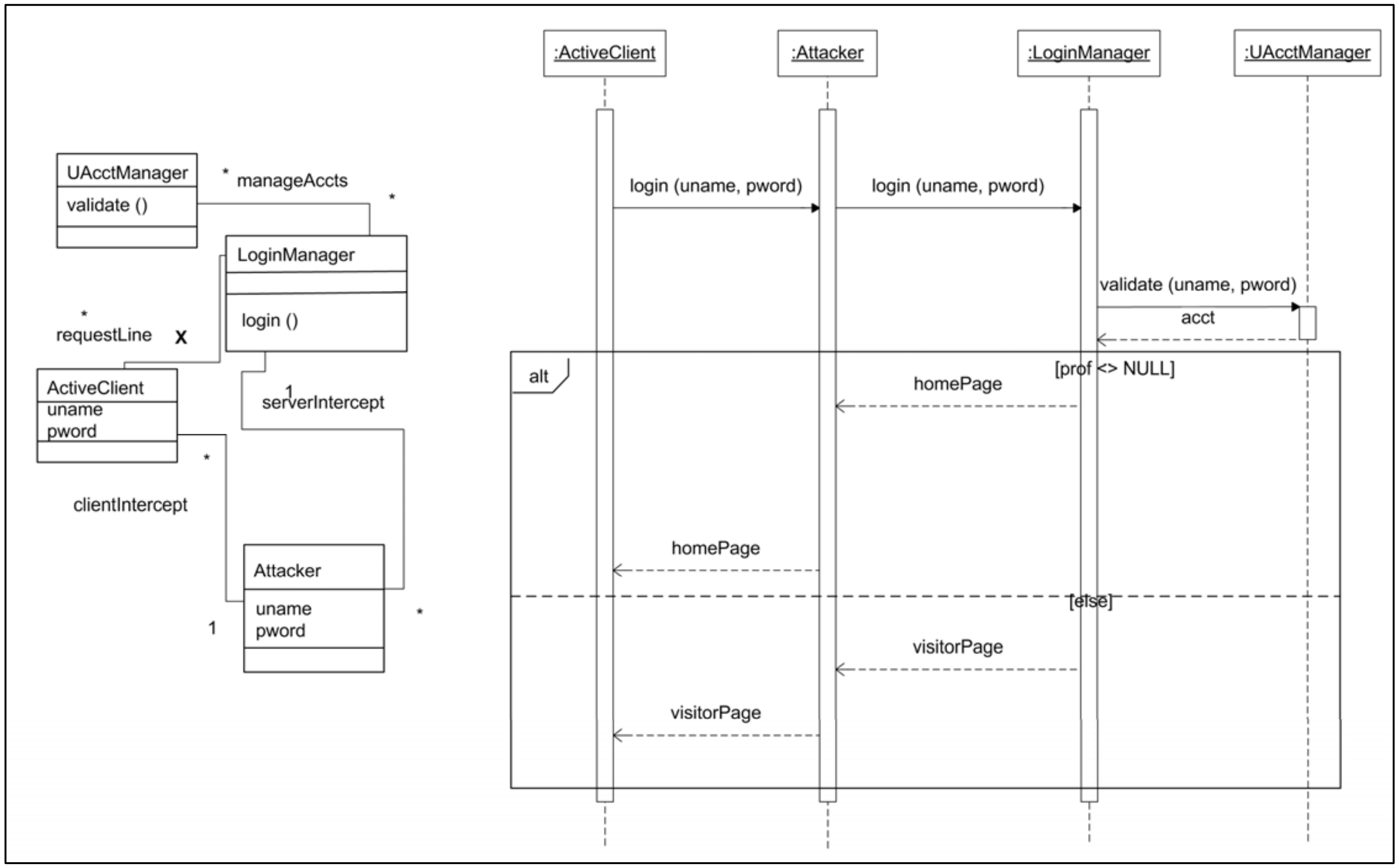

Figure 4. Context-specific man-in-the-middle attack model static and sequence diagrams.

**Obtaining the misuse model:** The context specific aspect is composed with the primary model to obtain the *misuse model*. The composed static diagram is shown in Figure 5, and the composed sequence diagram is shown in Figure 6. The first step is to compose the class diagrams of the attack and primary models.

Class diagram model elements are composed based on their construct type and “signature”. The signature can be simply defined as the name of the element, or it can be defined in more detail, such as a class name, attributes, and methods (perhaps including argument names and types). Composition proceeds by finding elements of the same construct type, with matching signatures in each model and then composing them. Our default algorithm is to simply add model elements that exist in one model or the other, but not both, in the composed model. Another default is to replace elements in the primary model with matching elements in the aspect model. Both default actions can be overridden using composition directives. The presence of an element in the primary model with a matching aspect model element marked for deletion (for example, the *requestLine* relation between *ActiveClient* and *LoginManager* in the context-specific aspect model of Figure 4) results in the element being deleted from the composition. Composition proceeds through all the elements of the class diagram. There are default actions to handle simple conflicts (e.g. different multiplicities in a relation), which can be overridden using composition directives.

Composition of the sequence diagram occurs in a similar fashion, based on matching construct types and their attributes, including name. Stereotypes are used to direct the addition (or deletion) of model elements into the composed sequence. The algorithm defaults again to replacing matching elements in the primary model with their counterparts from the aspect. Composition directives can be used to modify this behavior. Please see our previous work [20, 50] for details on composition.

Model composition is a largely automated task. A human need only be involved to re-direct composition behavior from algorithm defaults, if this is needed, or to decide how to resolve conflicts that cannot be handled by algorithm defaults. An example of such a situation occurs when a composition results in the deletion of an attribute that is needed by another class. To identify such conflicts, a tool must employ various dependency tracing mechanisms. To resolve the conflict though, a human must decide whether to reinstate the attribute, the dependency by moving the location of the data contained in the attribute, or make other changes to eliminate the need for this information.

The misuse class diagram (Figure 5) differs from the primary model class diagram in the following ways: (i) an *Attacker* class is added, (ii) an association between *Attacker* and *ActiveClient* is added, (iii) an association between *Attacker* and *LoginManager* is added, and (iv) direct association between the *ActiveClient* and *LoginManager* is deleted because all communications now go through the attacker class. The composed sequence diagram (Figure 6) shows the addition of the *Attacker* lifeline, and the fact that all communication between *ActiveClient* and *LoginManager* flows through *Attacker*. This sequence diagram will be used to illustrate how much the primary model can be compromised by a man-in-the-middle attack.

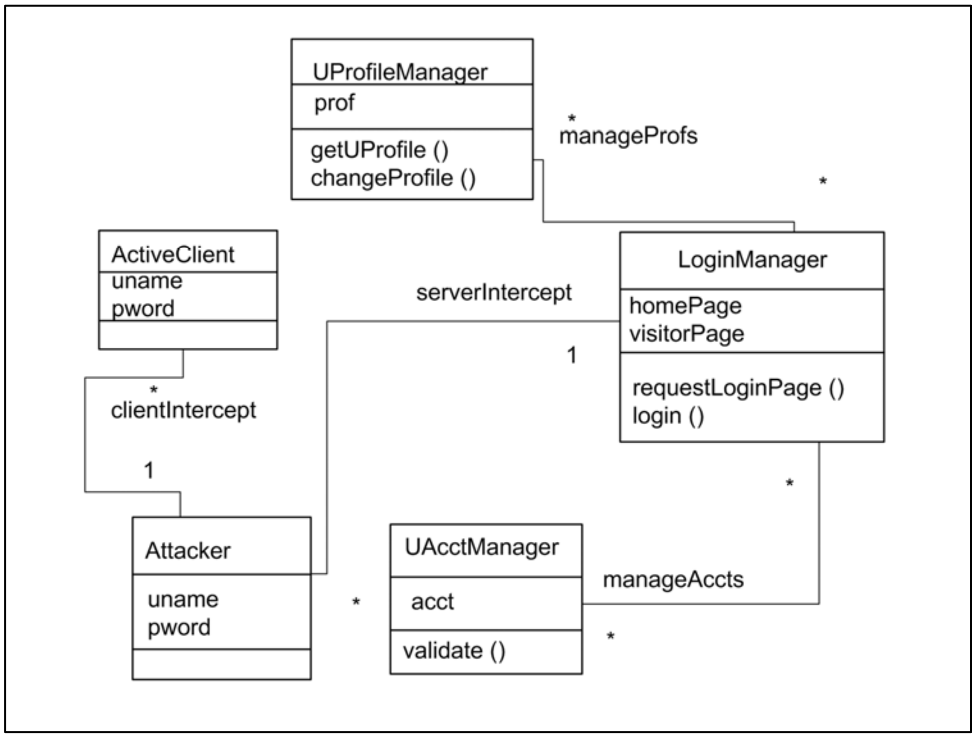

Figure 5. Misuse model (composed primary and man-in-the-middle) class diagram

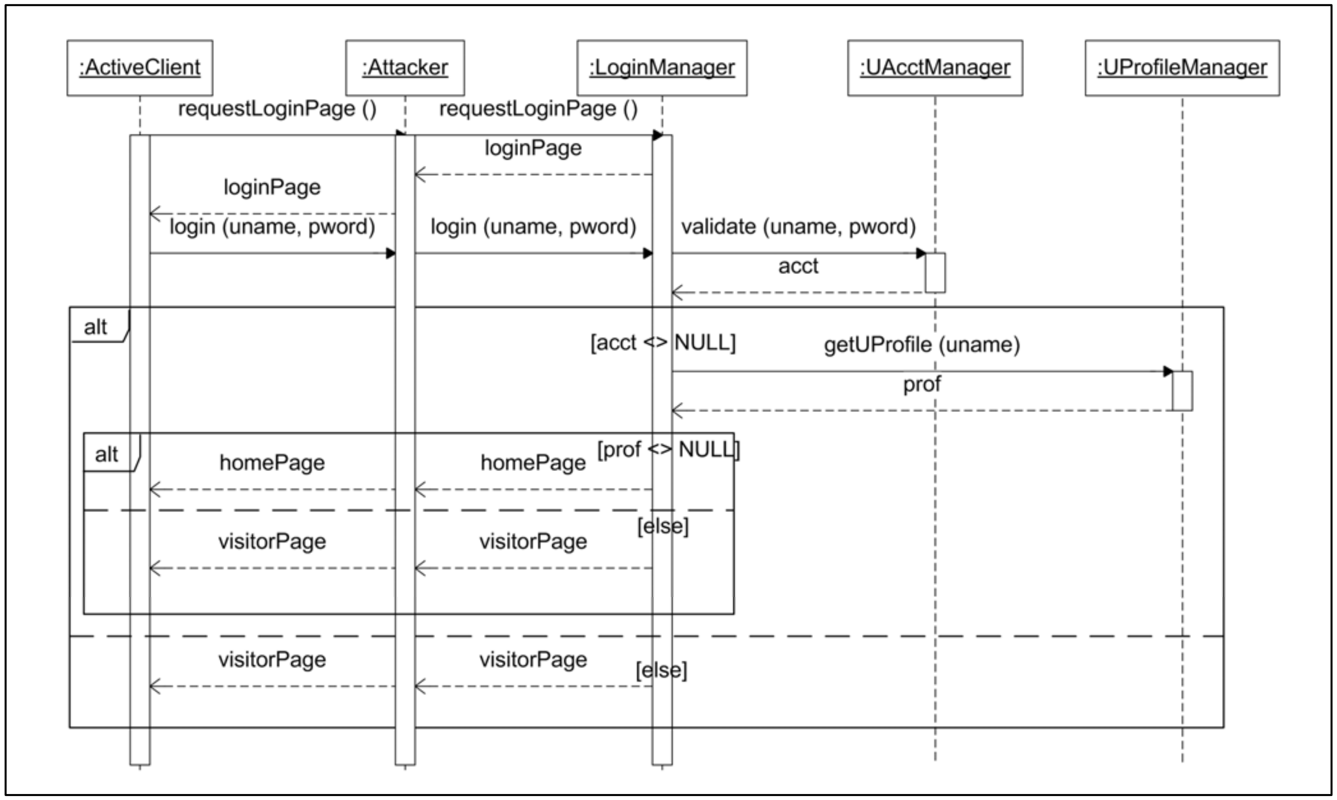

Figure 6.  Misuse model (composed primary and man-in-the-middle) sequence diagram

## 4.2. Evaluating the Impact of an Attack on the Application

Informally, we can reason about the misuse diagram as follows. First, the security properties that should be present are that: 1) *Attacker* should not receive *uname* and *pword* and 2) *Attacker* should not receive *homePage*. However, the sequence clearly shows that the attacker obtains the login name and password of a registered user, which can be used at a later time to impersonate the registered user, and the attacker also receives a *homePage*, if one is sent. The ability of the attacker to extract these secrets can be also formally analyzed using techniques developed by Jürjens [32] or Alloy Analyzer [1, 30], which we use later in this paper.

To counter a man-in-the-middle attack, authentication and confidentiality mechanisms must be incorporated into the login service. The mechanism that we choose is Transport Layer Security (TLS) [51]. We chose to use TLS since it is a follow-on to SSL (Secure Sockets Layer) [52], which is a commonly available authentication mechanism used in web applications. Other mechanisms could also be used to provide a stronger authentication mechanism for the service, including proprietary schemes developed for particular applications. The only requirement for use with our methodology is that the mechanism be specified as a UML model (using static and behavioral diagrams), and that it be specified at a similar level of abstraction as the functional system design primary model.

## 5. Incorporating TLS Authentication in the Application

The security properties of authentication and confidentiality are both at risk with the man-in-the-middle attack, so mechanisms that address authentication and confidentiality are potential risk treatments. We demonstrate the use of TLS to mitigate the man-in-the-middle attack risk. TLS is based on passing certificates between a client and server for authentication purposes, and to establish secret session keys for the encryption of all subsequent messages. In this paper, we use a variant of TLS described by Jürjens [32]. The sequence of the TLS mechanism is shown as a generic aspect diagram in Figure 7. (The class diagram for TLS is quite simple, consisting of only two classes, |Client and |Server, and the methods and attributes used in the sequence diagram. It is not included in this paper.)

The TLS generic aspect contains two classes: |Client and |Server. Certificates shown in Figure 7 (i.e. sCert and cCert) are data that contain a name, e.g. server name or user name, and the public key associated with that name. They are signed by an authority. Signing is essentially an encryption of the certificate with the authority's private key. As a guide to understanding the sequence diagram, the first argument of the *extractKey, extractName, sign, encrypt*, and *decrypt* methods is the key that is used to perform the operation, and the second argument is the data element from which information is to be extracted. For example, the method call *sPublicKey=extractKey(caPubKey,sCert)* indicates that the server's public key, *sPublicKey*, is to be extracted from the certificate authority signed server certificate, *sCert*, using the certificate authority's public key, *caPubKey*. For the purposes of this example, certificate creation and all public and private keys are assumed to be obtained in a secure manner. The client must have the certificate authority's (CA) public key, and the server must have a certificate, signed by the certificate authority, of its name and public key. Other assumptions include the fact that both unique identifier numbers called nonces, and session keys must change each time the protocol is initiated.

A TLS sequence begins with |*Client* sending an *init* message that contains a nonce (*iNonce*), its public key (*cPublicKey*), and its certificate (*cCert*). When |*Server* receives this message, it extracts the public key using the client's public key sent in the message, and the client user name. It checks that the public key in the signed portion of the message is the same as the public key sent in the unsigned portion of the message. If not, the TLS authentication is aborted.

If the client public keys match, |*Server* creates a message containing the session key that needs to be used for encryption once the connection is complete, the nonce received in the original client message, and the client public key extracted from the client certificate. This message is then signed using |*Server*'s private key. This signed message is then encrypted using |*Client*'s public key. The result, along with |*Server*'s certificate (which is signed by a trusted certificate authority) is sent to |*Client*.

|*Client* first extracts the server public key from the certificate, using the certificate authority's public key. It then extracts the server name from the certificate. If the name of the server in the certificate (*recName*) matches the name of the server (*sName*) to which the original *init* message was sent, the protocol proceeds. Otherwise |*Client* aborts the authentication. The encrypted portion of the message is decrypted using |*Client*'s private key (*cPrivateKey*), and the items in the resulting signed message are extracted using |*Server*'s public key (using the methods *getKey*, *getNonce*, and *getPkey*). The received nonce value (*reciNonce*) is compared to the nonce originally sent by the client (*iNonce*). If it does not match, this indicates that an attack on the communication has occurred, and |*Client* aborts the operation. If the items match, another check is made against the received client public key (*recPubKey*) and |*Client*'s internal public key

(*cPublicKey*). If these items match, then the communication path is secure, and |*Client* can encrypt its secrets using the session key and transmit them to |*Server*. |*Client* therefore encrypts a continue message of some sort (|*contL*) and sends it to |*Server*.

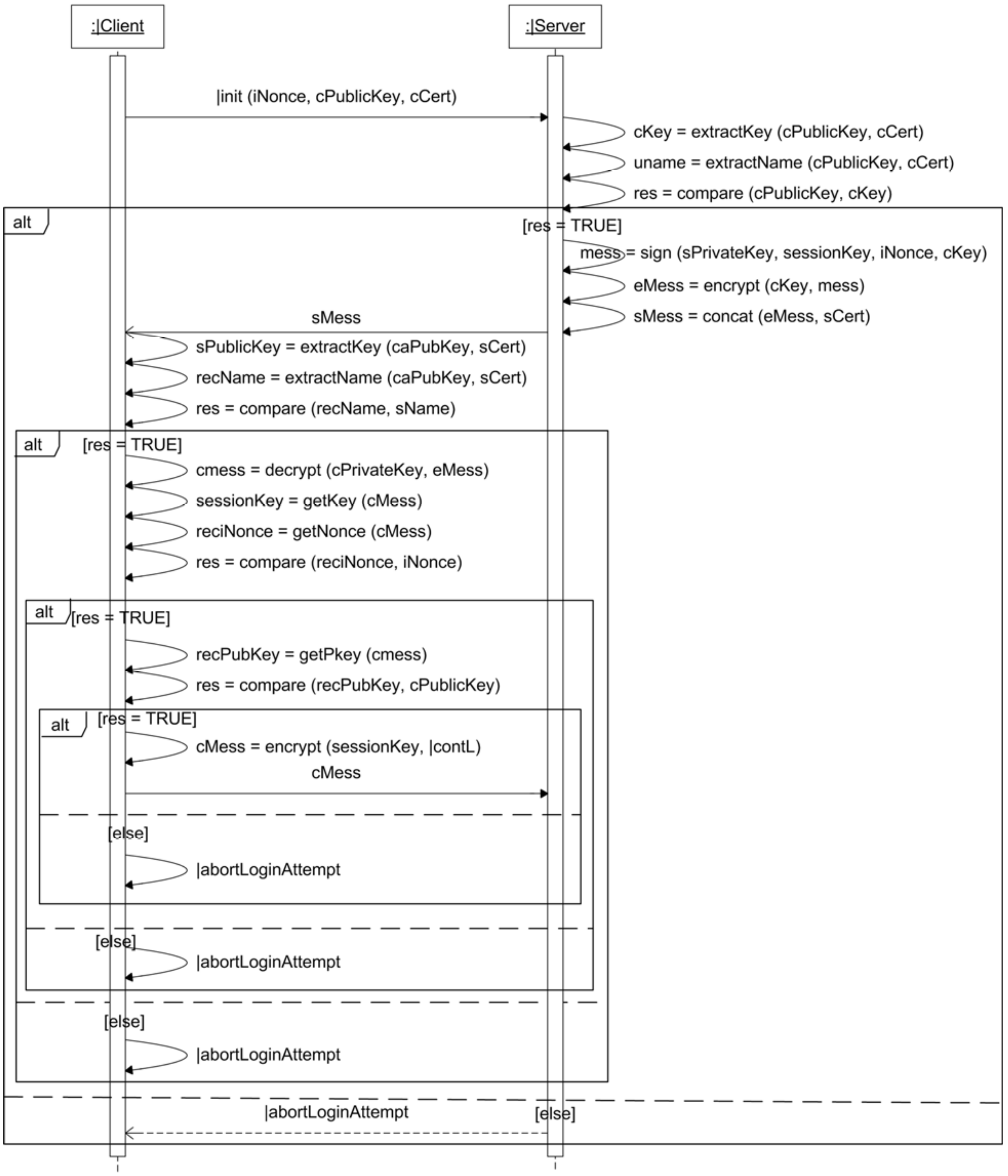
:|Client
:|Server
|init (iNonce, cPublicKey, cCert)
cKey = extractKey (cPublicKey, cCert)
uname = extractName (cPublicKey, cCert)
res = compare (cPublicKey, cKey)
alt
[res = TRUE]
mess = sign (sPrivateKey, sessionKey, iNonce, cKey)
eMess = encrypt (cKey, mess)
sMess = concat (eMess, sCert)
sMess
sPublicKey = extractKey (caPubKey, sCert)
recName = extractName (caPubKey, sCert)
res = compare (recName, sName)
alt
[res = TRUE]
cmess = decrypt (cPrivateKey, eMess)
sessionKey = getKey (cMess)
reciNonce = getNonce (cMess)
res = compare (reciNonce, iNonce)
alt
[res = TRUE]
recPubKey = getPkey (cmess)
res = compare (recPubKey, cPublicKey)
alt
[res = TRUE]
cMess = encrypt (sessionKey, |contL)
cMess
[else]
|abortLoginAttempt
[else]
|abortLoginAttempt
[else]
|abortLoginAttempt
|abortLoginAttempt
[else]

Figure 7. Generic TLS aspect sequence diagram

## 5.1. Generating the Security-Treated Model

The TLS mechanism model can be composed with the e-commerce model in Figure 2 in order to create a security-treated model that incorporates TLS capabilities. The TLS generic model is first instantiated as a context-specific model using bindings defined between it and the primary model, as described previously in Section 4.1. Similar to creating the context-specific attack model, a context-specific security aspect model may be a direct output of the risk analysis step of our methodology. If not, the bindings needed to create a context-specific model will be an output of the analysis, and these can be used to automatically create the context-specify aspect model. The context-specific models are the composed. The resulting composed static diagram is shown in Figure 8, and the composed sequence diagram is shown in Figure 9.

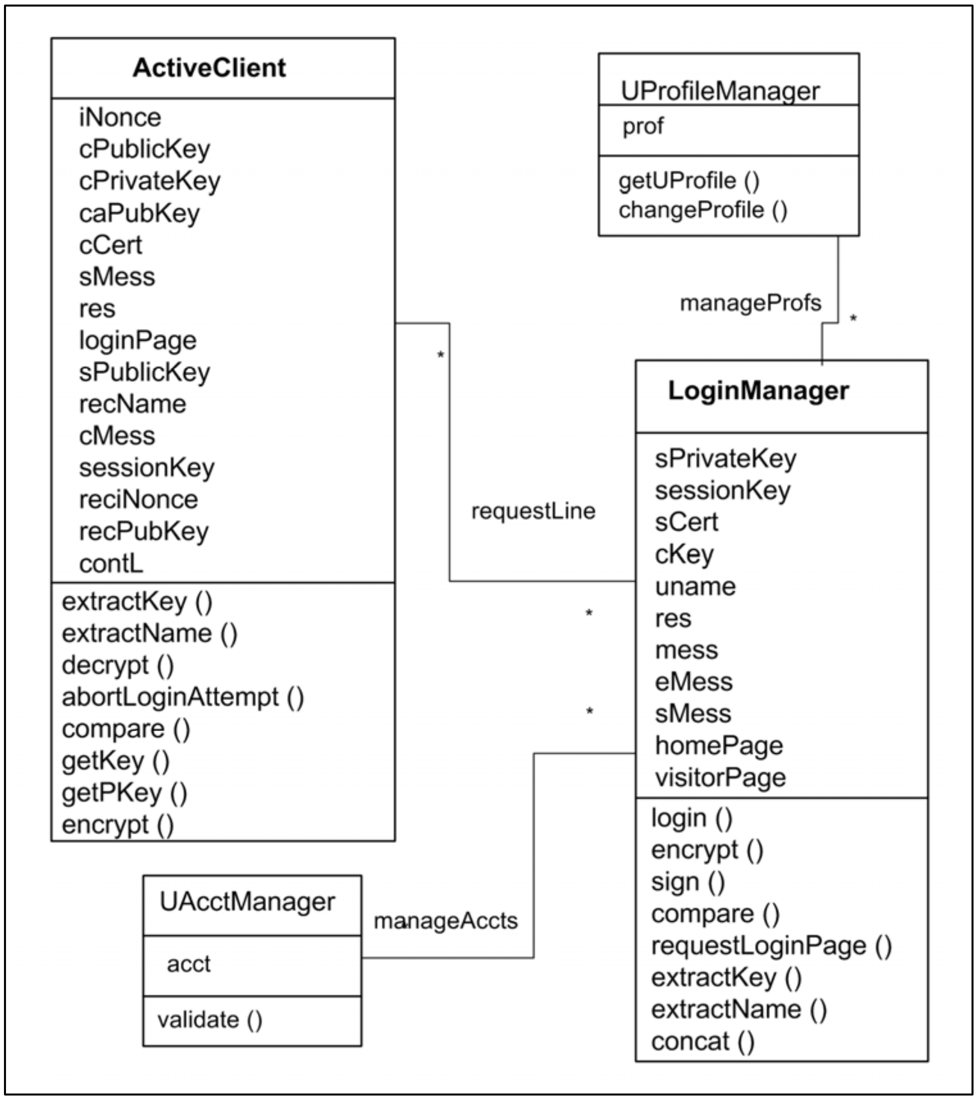

Figure 8. Security-treated static diagram

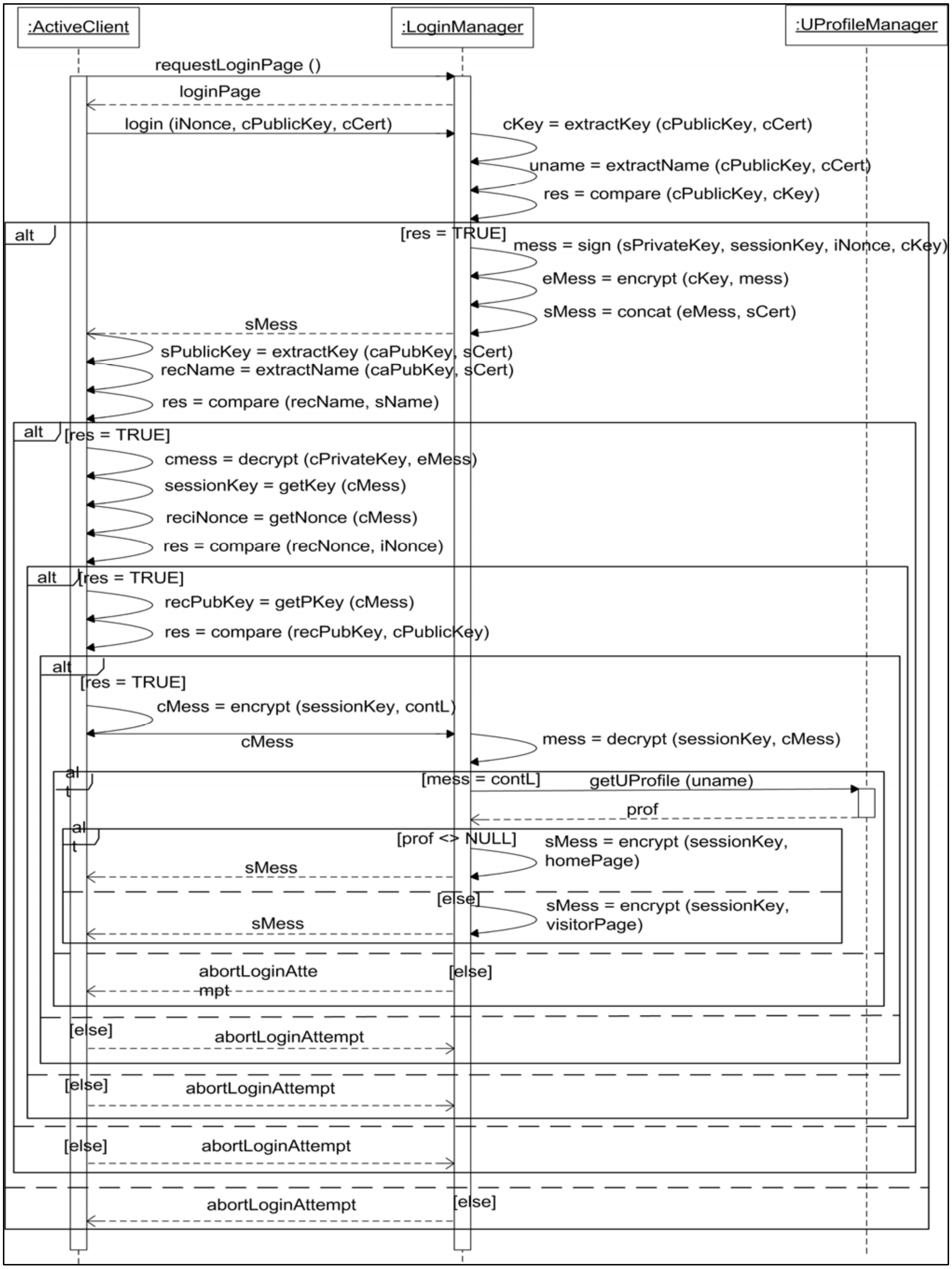
:ActiveClient
:LoginManager
:UProfileManager
requestLoginPage ()
loginPage
login (iNonce, cPublicKey, cCert)
cKey = extractKey (cPublicKey, cCert)
uname = extractName (cPublicKey, cCert)
res = compare (cPublicKey, cKey)
alt
[res = TRUE]
mess = sign (sPrivateKey, sessionKey, iNonce, cKey)
eMess = encrypt (cKey, mess)
sMess = concat (eMess, sCert)
sMess
sPublicKey = extractKey (caPubKey, sCert)
recName = extractName (caPubKey, sCert)
res = compare (recName, sName)
alt
[res = TRUE]
cmess = decrypt (cPrivateKey, eMess)
sessionKey = getKey (cMess)
reciNonce = getNonce (cMess)
res = compare (recNonce, iNonce)
alt
[res = TRUE]
recPubKey = getPKey (cMess)
res = compare (recPubKey, cPublicKey)
alt
[res = TRUE]
cMess = encrypt (sessionKey, contL)
cMess
mess = decrypt (sessionKey, cMess)
alt
[mess = contL]
getUProfile (uname)
prof
alt
[prof <> NULL]
sMess = encrypt (sessionKey, homePage)
sMess
[else]
sMess = encrypt (sessionKey, visitorPage)
sMess
abortLoginAttempt
[else]
[else]
abortLoginAttempt
[else]
abortLoginAttempt
[else]
abortLoginAttempt
abortLoginAttempt
[else]

Figure 9. Security-treated sequence diagram

The sequence shown in Figure 9 begins as the sequence did in Figure 2, with the *ActiveClient* requesting a login page from the *LoginManager*. The *LoginManager* responds with *loginPage*. Now the TLS sequence is inserted; instead of *ActiveClient* sending a login message with a user name (*uname*) and password (*pword*), a different login message is sent. This new login message contains a nonce, the user's public key, and certificate. The logic for the TLS handshake continues as in the TLS aspect model, with model element name changes per the bindings discussed above. Once the TLS handshake completes successfully, the *ActiveClient* sends a continue message to *LoginManager*, which in turns causes the *LoginManager* to get personal profile information (if it exists), and send a *homePage* back to the user via *ActiveClient*. If the profile information does not exist, a *visitorPage* is sent back to the user.

## 5.2. Creating the Security-Treated Misuse Model

Once security mechanisms have been incorporated into a primary model, we need to verify whether the given attack is prevented in this new model. In our example, we need to determine whether the TLS authentication adequately protects the login service from the man-in-the-middle attack. We can reason about the effective security after composing the man-in-the-middle aspect with the security treated primary model and analyzing it for desired security properties.

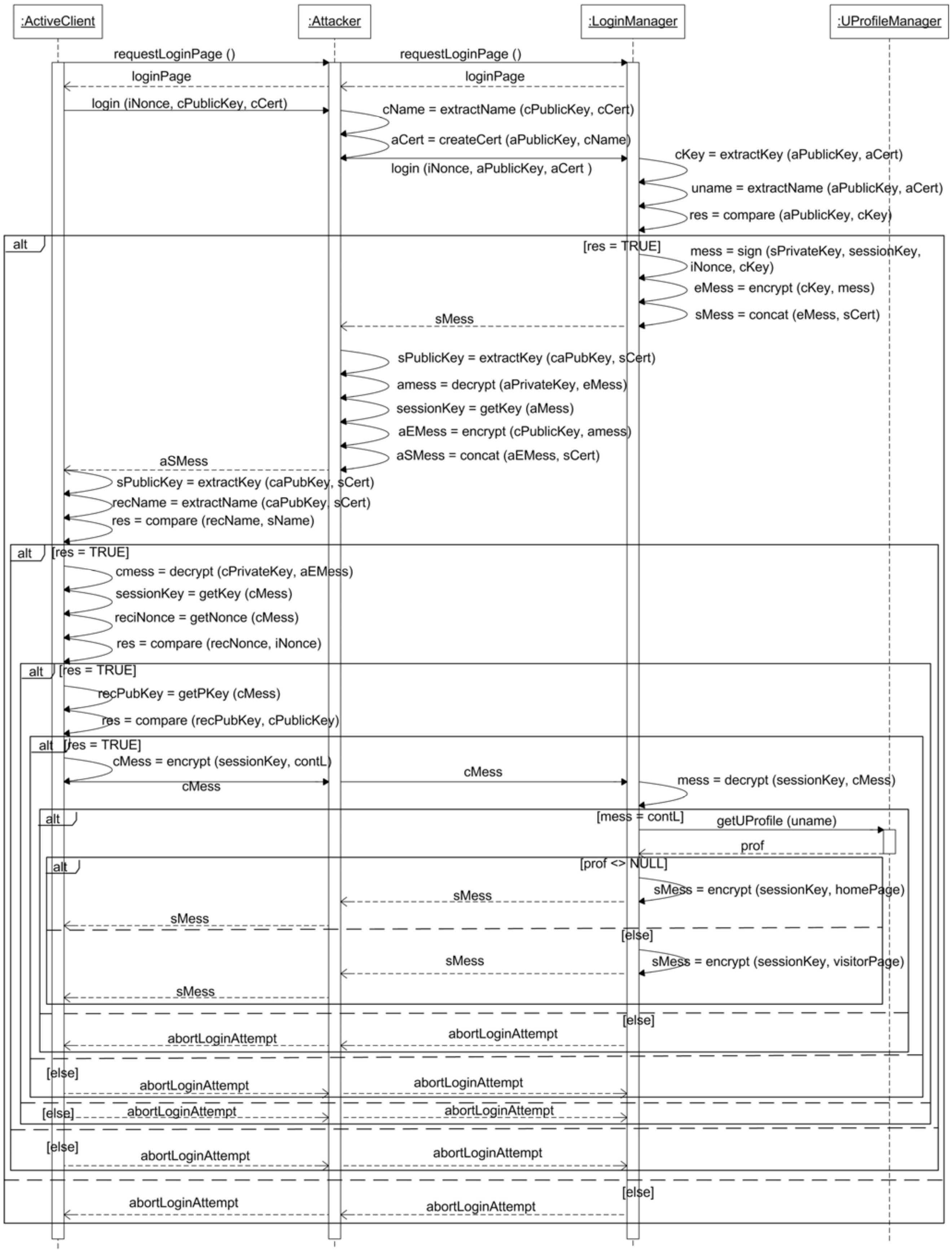
:ActiveClient
:Attacker
:LoginManager
:UProfileManager
requestLoginPage ()
requestLoginPage ()
loginPage
loginPage
login (iNonce, cPublicKey, cCert)
cName = extractName (cPublicKey, cCert)
aCert = createCert (aPublicKey, cName)
login (iNonce, aPublicKey, aCert )
cKey = extractKey (aPublicKey, aCert)
uname = extractName (aPublicKey, aCert)
res = compare (aPublicKey, cKey)
alt
[res = TRUE]
mess = sign (sPrivateKey, sessionKey, iNonce, cKey)
eMess = encrypt (cKey, mess)
sMess = concat (eMess, sCert)
sMess
sPublicKey = extractKey (caPubKey, sCert)
amess = decrypt (aPrivateKey, eMess)
sessionKey = getKey (aMess)
aEMess = encrypt (cPublicKey, amess)
aSMess = concat (aEMess, sCert)
aSMess
sPublicKey = extractKey (caPubKey, sCert)
recName = extractName (caPubKey, sCert)
res = compare (recName, sName)
alt
[res = TRUE]
cmess = decrypt (cPrivateKey, aEMess)
sessionKey = getKey (cMess)
reciNonce = getNonce (cMess)
res = compare (recNonce, iNonce)
alt
[res = TRUE]
recPubKey = getPKey (cMess)
res = compare (recPubKey, cPublicKey)
alt
[res = TRUE]
cMess = encrypt (sessionKey, contL)
cMess
cMess
mess = decrypt (sessionKey, cMess)
alt
[mess = contL]
getUProfile (uname)
prof
alt
[prof <> NULL]
sMess = encrypt (sessionKey, homePage)
sMess
sMess
[else]
sMess = encrypt (sessionKey, visitorPage)
sMess
sMess
[else]
abortLoginAttempt
abortLoginAttempt
[else]
abortLoginAttempt
abortLoginAttempt
[else]
abortLoginAttempt
abortLoginAttempt
[else]
abortLoginAttempt
abortLoginAttempt
[else]
abortLoginAttempt
abortLoginAttempt

Figure 10. Misuse model (security treated model + man-in-the-middle) sequence diagram

Figure 10 shows the detailed sequence when the man-in-the-middle attack is composed with the system protected by TLS. The attacker in this sequence is active, that is, *Attacker* changes messages flowing between *ActiveClient* and *LoginManager*. The first message that is changed is the *login* message, where the attacker creates a certificate with its own public key and *ActiveClient*'s user name. Thus, *LoginManager* has a valid user name, but the attacker's public key, so that any messages from *LoginManager* that have been encrypted using the "client" public key are actually encrypted with the attacker's public key. This encryption means that the attacker can decrypt them using its private key. When *LoginManager* sends back the message with the session key in it, *Attacker* decrypts it using its private key, and re-encrypts it using the real *ActiveClient* public key.

## 5.3. Analyzing the Security-Treated Misuse Model

Recall that the original threat posed by the man-in-the-middle attack is to obtain user information, as returned in a home page. In the original system this occurs when the attacker obtains a user name and password, followed by a *homePage*. The receipt of the *homePage* indicates that the user is registered, and the fact that the attacker has the user name and password means that these items can be used later to obtain more user information.

The addition of TLS to the login sequence changes the situation. First, the user name/password scheme no longer exists, so the attacker cannot simply eavesdrop to obtain information that can later be used to gain access to the system. Eavesdropping also will not work to obtain registered user information from a *homePage*, since all communication between *LoginManager* and *ActiveClient* is encrypted using a session key once the TLS authenticates *ActiveClient*. A successful attack can only occur if the TLS protocol is successful (i.e. a *homePage* or *visitorPage* is sent to *ActiveClient* from *LoginManager*), and *Attacker* obtains the session key.

Therefore the security property that needs to be preserved in the security-treated misuse sequence is that if the protocol succeeds, *Attacker* must not obtain the session key of the *LoginManager* and *ActiveClient*.

This property can be validated using either informal or formal methods. The next section presents the use of Alloy Analyzer to formally validate it. Here we reason informally about the effective security provided by TLS as follows. Since the public key in the *login* message is the same as in the certificate, the first test comparison in *LoginManager* will work. Next *LoginManager* creates a signed message (with its private key) containing the attacker's public key, received nonce value, and session key, and encrypts it using that same public key. This message and *LoginManager's* certificate are sent to the attacker, which decrypts the signed message with its private key and can extract items from the signed message using *LoginManager*'s public key as contained in its certificate. The signed message from the server is then encrypted with the *ActiveClient* public key, and is sent to the *ActiveClient*, along with the server's certificate. Note that the signed message itself cannot be changed since the attacker does not have the *LoginManager*'s private key. Also, a new signed message created with *Attacker*'s private key cannot be created since the certificate included in the message to *ActiveClient* would have to contain the *LoginManager* server name and *Attacker* public key and be signed from a trusted certificate authority. This is required so that the certificate authority public key possessed by *ActiveClient* can be used to obtain the "server" public key.

*ActiveClient* first extracts the server name and public key from the certificate using the CA public key. A comparison is made between the server name the *ActiveClient* has and the server name in the certificate. This test will work. Next, the *ActiveClient* decrypts the signed message from the *LoginManager* using its private key to obtain the session key and nonce value. It then compares the message nonce included in that message with the one it originally sent, and this test will also work. Next the client public key included in that message is extracted, and compared with its own public key. This test will fail because the client key included in the signed message from *LoginManager* is that of the attacker. Therefore the sequence will always move to the third test failure alternative where the *abortLoginAttempt* message will be returned to the user of *ActiveClient* and the sequence ends. Thus, the treatment prevents the attack, and consequently the undesirable properties it allows, from occurring.

## 6. Formally Verifying Authentication Properties in the Misuse Model

The form of informal analysis shown in the previous section is error prone and tedious. Towards this end, we show how such analysis can be done formally with the help of automated tools. We use the Alloy Analyzer to formally reason about the misuse model sequence shown in Figure 10 and the ability of TLS to protect the system from a man-in-the-middle attack. In the following sections we explain how the Alloy Analyzer can be used to verify that the desired security properties do indeed hold.

## 6.1. Alloy

Alloy [30] is a fully declarative first-order logic language that can be used to model complex software. An Alloy model consists of a number of *signatures* and *relation* declarations. A signature denotes a set of atoms, which are the basic entities of models. Relations are sets of tuples of atoms capturing the relationships between entities.

Alloy comes with an accompanying analyzer that is a fully automatic constraint solver. The analyzer operates on implications, for example that a system modeled in Alloy implies a particular property. This assertion is negated, and then translated into a Boolean expression. The analyzer uses a SAT solver to search for a model of the negated assertion. A user-specified scope on model elements bounds the domain, making the problem finite. This makes it possible to create a Boolean formula for the SAT solver. If a model is found that fits the negated assertion, this means the original implication has a counterexample and is not valid. If no counterexample is found, the original implication may still not be valid since the search was bounded by the user-defined scope. However, if a large scope is used, this situation can be made unlikely.

The Alloy Analyzer differs from theorem provers in this sense – if a counterexample is found, the implication is false, but if no counterexample is found the implication is still not necessarily true. The analyzer differs from model checkers in that it finds models of logical formulas whereas model checkers check that a state machine is a mathematical model of a temporal logic formula. Please see the Alloy website for more information on the language and analyzer [1].

## 6.2. Analyzing the Misuse Model for Security Properties

There are two steps involved in analyzing the misuse model in Figure 10 for security properties using the Alloy Analyzer. The first is to simplify the model to remove non-essential elements so that the translation to Alloy produces a model with only the items necessary to reason about the security properties. The second is to translate the UML model to Alloy using the UML2Alloy tool, as described by Bordbar and Anastasakis [8, 9, 54].

The UML2Alloy tool requires that the model be presented as a class diagram, accompanied with OCL specifications of method behavior. Both of these can be derived from the misuse sequence diagram of Figure 10. The class diagram must be complete in that it contains all attributes and their types, along with complete method signatures.

## 6.3. Creating an Abstracted OCL Specification

Abstracting the security-treated misuse model to exclude unused details cannot be fully automated. While portions of the task may be automated (noted in the discussion below), a set of heuristics guiding a human designer was used to create the results in this paper. These heuristics are as follows.

1. A designer must decide what assertions will be tested using Alloy Analyzer. For this example, we need to ensure that:
    (a) if the protocol succeeds, Attacker does not have the session key

The formulation of this assertion is influenced by Alloy Analyzer since the tool works by attempting to find a counterexample to the assertion. Formulating the assertion as in (a) means that the tool needs to search for a case where the protocol succeeds, and the attacker knows the session key. An alternative formulation, that if the attacker knows the session key, then the protocol should abort is harder to test since there are several reasons why the protocol might abort, besides the attacker gaining access to the session key.

2. Every message to a different object lifeline has the potential to become a method in the OCL specification of the receiving object, if the object performs some computation of interest as a result of receiving the message. If the receiving object just passes the message through to another object lifeline, the method will exist in the final receiving object. In order to support this heuristic, it is easiest to construct a message list, including sending and receiving object names. Since messages always exist between at most two object lifelines, this list can be created automatically. Messages that are the result of invoking methods in the same object are not included.
3. Every *alt* box in the sequence diagram of the misuse model represents an if-then-else OCL constraint in the specification, so the next step is to identify each of these tests, and identify the variable dependencies that exist in them. For example, the first *alt* box in Figure 10 has a guard of *[res = True]*. The variable *res* is set as a result of the comparison of *aPublicKey* and *cKey*. The *cKey* variable is obtained by extracting the public key from the certificate, *aCert*. The *aPublicKey* variable is an argument in the *login* method message. The *res* variable thus depends on *aPublicKey* in the *login* message, *cKey*, and therefore *aCert* in the *login* message. Similar dependencies must be identified for each test of each *alt* box.

After this step is complete, messages that do not affect these variables are removed from the message list. This step removes the *requestLoginPage* and *loginPage* messages. Classes that are not involved in the remaining messages can also be removed. Messages involving the conditions that will be tested with Alloy

Analyzer need to be retained. In the case of our example, this means that the messages involving returning a web page to *ActiveClient* need to be retained. We will also have to have some way to tell that the protocol has not aborted, in this case we choose a simple Boolean variable, *loginAborted*. This variable will be used in step 5, below.

A further simplification that needs to occur is to replace variables that are hierarchical with their constituent parts, discarding parts that are not needed. For example, certificates contain many things, but all we care about in this example is the name of the certificate owner and the public key. We make this simplification for the *ActiveClient*, *Attacker*, and *LoginManager* certificates. Similarly, the messages sent between *ActiveClient*, *Attacker*, and *LoginManager* have names like *sMess*, *aSMess*, and *cMess*. These are hierarchical variables, so we replace them with the variables that we identified in the *alt* box dependency development.

The next steps create the OCL specifications of methods used in the UML2Alloy transformation.

4. Classes are specified with methods named for the messages received by the class. For example, *Attacker* has a method called *recLoginFromAC* (corresponding to the *login()* message sent from *ActiveClient* to *LoginManager*, but intercepted by *Attacker*, which changes it and forwards it onto *LoginManager*). *ActiveClient* has a method called *abortLoginAttempt* that corresponds to the *abortLoginAttempt* messages from *LoginManager* to *ActiveClient* (via *Attacker*, which simply passes them through). A method called *main* must be added prior to the first method, to start the scenario during analysis. This method is added to *ActiveClient* in our example, and the last thing this method does is to invoke *recLoginFromAC* in *Attacker*. Variables that are used in the callee method need to be set next. So for example, in Figure 12, the OCL specification of the *main* method is shown, and this includes setting the initial nonce value, the server name, the client name, and then invoking the *recLoginFromAC* method.
5. Any *alt* boxes that appear in the sequence diagram after a message corresponding to a method call in the OCL specification, but before any other messages corresponding to method calls, will result in an if-then-else constraint in the method body. So, the first *alt* box of the sequence occurs after the *login* message is received in *LoginManager*. The corresponding OCL specification of the method *recLoginFromAttacker* thus has an if-then-else block around a comparison of the key received as part of the message and the key contained in the client certificate. Recall that the client certificate was replaced above by the name and key, e.g.

*ac.certName* in Figure 12. The misuse model shows *Attacker* replacing both the public key in the certificate and the public key in the message with its own public key; these are the values that will be checked by *LoginManager*. Variables that have been added in order to test security properties with Alloy Analyzer (i.e. *loginAborted* in our example) must be set appropriately in the OCL method specifications. For example, *self.loginAborted* is set to true in the *abortLoginAttempt* method of *ActiveClient*.

Variables that exist in other classes are accessed via the relations shown in Figure 11. For example, *LoginManager* can test the values of the public keys by following the relation to the *Attacker* class, *at1*. The public key that was in the certificate is *at1.certKey*, and the public key that was just in the message is *at1.pubKey*. Methods are also accessed through relations. For example, if the public key test fails, *LoginManager* invokes *at1.ac1.abortLoginAttempt* in *ActiveClient*.

Using these heuristics allowed us to develop the class diagram and OCL specification needed by the UML2Alloy tool, which is discussed next.

## 6.4. UML2Alloy Class Diagram and OCL Input

A portion of the security-treated misuse model class diagram used in the transformation from UML/OCL to Alloy is shown in Figure 11.

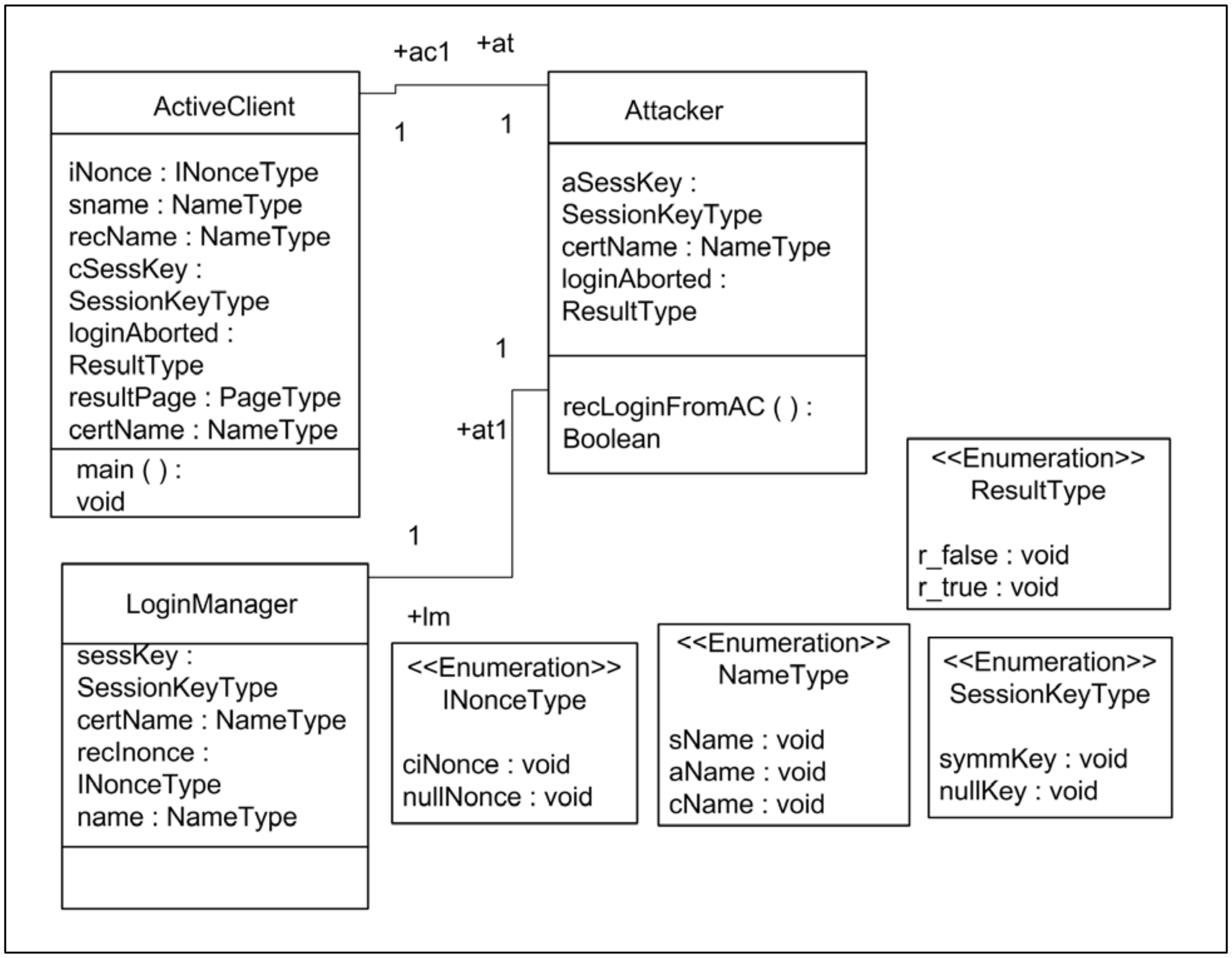

Figure 11. Portion of misuse model class diagram used for UML2Alloy tool translation

The complete class diagram shows the types of all attributes and return types of methods. Figure 11 shows a portion of this diagram to illustrate its form. Since Alloy has no primitive types everything must be declared as a separate type, which will be a set in the Alloy model. Boolean types in the model (e.g. *ResultType*) are defined as an enumerated variable with the value *r_true* or *r_false*. The behavior of the methods is specified with the help of OCL pre- and post-conditions. Pre-conditions are statements that must be satisfied before the invocation of a method. Post-conditions are the declarative outcome of the method execution. The overall specification must have an entry point to be analyzable, so the *ActiveClient* class is augmented with a *main()* method. Navigation is specified with a dot ('.') notation, and the special name *self* refers to the context object. OCL statements return a Boolean type, so boolean return type operations can be invoked from within other OCL operations using the format `object.operationcall()`.

The OCL standard forbids the referencing of non-query operations from within an OCL statement. This is too restrictive for our approach. In order to be able to simulate and reason about UML models we have to extend OCL to be able to reference other non-query OCL specifications from within a pre- or post-condition. To achieve this, we use Nunes' [42] approach, which makes this extension to OCL and provides its formal semantics. The main benefit of the extension is that it makes OCL more expressive. For example, in the OCL statement of Figure 12, we can reference the recLoginFromAC(), which is a non-query operation, from within the OCL specification of the *main()* operation. This enables us to simulate the model using Alloy Analyzer.

Figure 12 shows the OCL definition of the *main()* method of the *ActiveClient* class. The method specifies that the *recLoginFromAC()* method of the *Attacker* class related to the *ActiveClient*, should be invoked. The rest of the methods in the model have similar specifications, in that they specify the values of the attributes of the classes, and invoke class methods.

```
context ActiveClient :: main ( ) : Boolean
-- The main operation imposes that the
recLoginFromAC()
-- operation will apply to all of the
ActiveClients; they will all send a
-- login message to LoginManager (via Attacker)
-- The method also initializes the nonce, server
name, and user
-- name attributes
```

```
post main :
    ActiveClient.allInstances() ->
forAll(ac:ActiveClient |
        ac.iNonce = INonceType::ciNonce and
        ac.sname = NameType ::sName and
        ac.certName = NameType ::cName and
       ac.at.recLoginFromAC())
```

Figure 12. OCL specification of the `main()` method of the *ActiveClient* class.

## 6.5. Invariants and Assertions

Invariants must be created to constrain the Alloy model. For example, the “main” method of the *ActiveClient* must specify initialized attributes and state that the *recLoginFromAC()* operation of the *Attacker* has to hold for all *Attackers* in the system. This is the purpose of the OCL post-condition shown in Figure 12.

Similarly, assertions must be created to verify the security properties of interest. The property discussed in Section 4.3 must be translated into an OCL assertion, then into an Alloy assertion for verification. Recall that this property is: if the TLS protocol succeeds, *Attacker* must not possess the same session key as *LoginManager* and *ActiveClient*. An OCL assertion for this property is:

```
context ActiveClient
ActiveClient.allInstances()                     ->
forAll(ac:ActiveClient |
     ac.loginAborted    =    ResultType::r_false
implies (
     ac.cSessKey = SessionKeyType::symmKey and
     ac.at.lm.sessKey  =  SessionKeyType::symmKey
and
     ac.at.aSessKey <> SessionKeyType::symmKey )
)
```

## 6.6. Creating an Alloy Model

This section sketches the method adopted for conducting the transformation from the UML to Alloy, which draws on Model Driven Architecture (MDA) [35]. This requires creation of a metamodel [43] of the source modeling languages i.e. the UML/OCL and the destination modeling language, Alloy. Then, a transformation is specified via rules that map model-elements of the source metamodel to a destination metamodel.

The UML and OCL metamodels have already been defined in their respective specifications [44, 45], which have been released by the Object Management Group (OMG). We are currently supporting a subset of the UML and OCL metamodels, as the UML metamodel is very large and includes elements that cannot be mapped to Alloy. We have created an Alloy metamodel from the Alloy grammar [30], using the methods proposed by Muller et al. [40] and Wimmer and Kramler [58].

Transformation rules from UML to Alloy are explained by Bordbar and Anastasakis [9]. In particular, UML *classes* are directly translated to *signatures* in Alloy. UML *associations* are translated to fields of *signatures*. When translating an association, additional multiplicity facts are imposed on the Alloy fields to reflect the multiplicity constraints of the association ends that take part in the association. Class attributes are also translated to *signature* fields. UML types and enumerators are also translated to *signatures*. It is also important to note that binary bidirectional associations in UML are translated to symmetric relations in Alloy.

Both OCL and Alloy are based on first-order logic. They are therefore quite similar, and the translation from OCL to Alloy is quite straightforward when dealing with first-order logic statements. As a result, the *forAll* OCL construct is translated to *all* in Alloy and the *exists* OCL construct to *some* in Alloy. For an extended study of the similarities and differences of OCL and Alloy please refer to Vaziri and Jackson [55].

All OCL statements are declared under a context, which is the element of the UML model on which the OCL expression is evaluated. In OCL there is a special name *self*, which refers to the context object. Alloy expressions on the other hand are evaluated globally. Therefore, it is essential to define the notion of context in Alloy models explicitly. This can be achieved by adding an object, which represents the context, as a parameter in the predicate to which the original OCL statement is translated. So, for instance, the OCL statement of Figure 12 is translated to the Alloy predicate of Figure 13. Translation of the rest of the expression items is straightforward. The complete Alloy model of the misuse model containing the flawed TLS-protected login sequence in the presence of a man-in-the-middle attack is given in the Appendix.

```
pred main(){
all ac: ActiveClient | ((ac.iNonce = ciNonce)
&& (ac.sname = sName)
&& (ac.certName = cName)
&& recLoginFromAC (ac.at))
}
```

Figure 13. Alloy code after the translation of the `main()` method of the *ActiveClient* class.

The translation rules have been implemented in a tool called UML2Alloy [18, 19]. Figure 14 depicts the sequence of steps involved in the transformation. The starting point is to create a UML model of the system in a UML CASE tool such as ArgoUML [3]. Most UML tools, including ArgoUML, can export the UML model to an XMI [46] format. XMI, which stands for XML Metadata Interchange is an OMG standard used by UML tools to store, import and export UML models. UML2Alloy implements the transformation and generates an Alloy model from the XMI file. The Alloy model of the system can then be analyzed with the Alloy Analyzer [1].

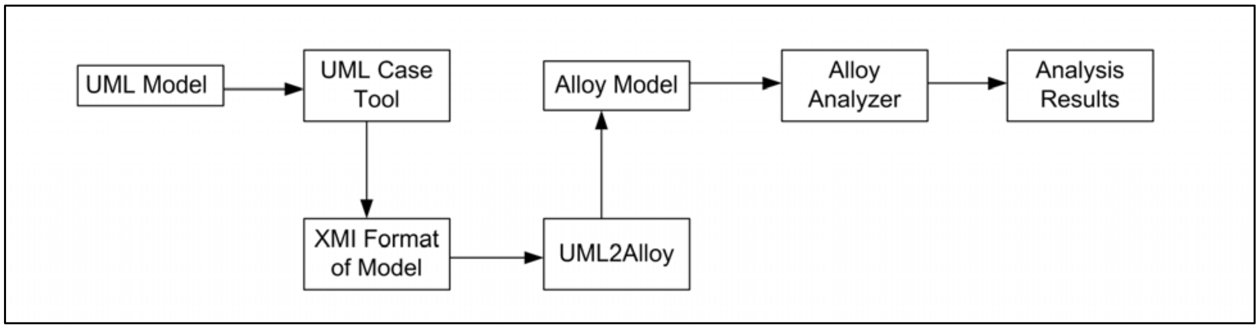

Figure 14. Process of Analysis of UML models via UML2Alloy

## 6.7. Results from the Alloy Analyzer

The first step in analysis is to simulate the model. Simulation generates a random instance that conforms to the *whole* specification, ensuring there are no conflicting statements. The next step is to formulate the OCL assertions as Alloy assertions, capturing properties that we wish to check, as outlined above. The Alloy analyzer automatically checks such assertions and if they fail to produce a counterexample. The Alloy translation of the OCL assertion presented in section 5.5 is:

```
assert sameKeySuccess{
    all ac:ActiveClient | ac.loginAborted = r_false
    implies
       ( ac.cSessKey = symmKey &&
            ac.at.lm.sessKey = symmKey &&
            ac.at.aSessKey <> symmKey ) }
```

The Alloy Analyzer does not present any counterexamples to the above assertion, indicating that this security property is present in the TLS security-treated system model.

## 6.8. Discussion

As explained in Section 5.1, analysis in Alloy requires selecting a scope for the execution, that is, putting an upper limit on the number of elements of each signature. The underlying idea of Alloy is to deploy automated analysis to inspire confidence in the correctness of the design. The larger the scope, the more confidence is warranted, but the longer the analysis will take [30, page 163]. Currently, there is no clear guideline or method for identifying a suitable scope for analyzing an Alloy model. However, experience has shown that design flaws are often discovered in smaller scope. This is known as “small scope hypothesis” [30, Section 5.1.3]. Finding a suitable scope for each problem is a practical problem and is a matter of experience.

In this example, we initially analyzed the security-treated misuse model with a scope of 1 for each of the *ActiveClient*, *Attacker* and *LoginManager* and a scope of up to 8 for the rest of the model elements, to analyze the attack by a single *Attacker*. This analysis did not provide any counterexamples and returned the results in less than one second. To increase our confidence in the correctness of the design we increased the scope to sixteen for all model elements. This means that the Alloy Analyzer has searched for a counterexample for all combinations of up to sixteen clients, attackers, servers, public keys, certificate names, etc. Again the Analyzer did not produce a counterexample.

Table 1 captures time required for the analysis of the model in the scope of up to sixteen on a server with two dual core AMD Opteron CPUs and 4GB of RAM. It can be seen that the time required for the analysis increases rapidly, as scope increases. Since there are many relations between the elements of the model, as the scope increases the number of possible cases the Analyzer needs to search, grows dramatically. With a scope of 16 the Analyzer exhaustively searches a very large number of cases, but still does not find a counterexample that violates our assertion.

Table 1. Performance of the Alloy Analyzer

| **Scope** | **Time Required** |
|---|---|

| | |
|---|---|
| Scope of 1 for the ActiveClient, the Attacker, the LoginManager. Scope of 8 for the rest model elements. | ~ 1 second |
| Scope of 8 for all model elements. | ~ 9 seconds |
| Scope of 12 for all model elements. | ~13 seconds |
| Scope of 16 for all model elements. | ~ 43 seconds |

Finally, we wish to point out that this analysis is valid only for the particular attack that is included in the Alloy model, i.e. man-in-the-middle. A new Alloy model must be created for each type of attack that needs to be checked. Of course, the assertions that are checked must correctly reflect how a properly protected system should respond to the attack.

## 7. Related Work

Recently, a lot of work appears in the areas of AOM and AOP [2, 13, 14, 31, 34, 57]. These works are similar to ours in that they represent a view of interest, e.g. security, and they are cross-cutting. In AOM, this means that an aspect model must be integrated in several places with the primary model. In AOP, aspect code must be inserted into multiple components of the implementation. In both AOM and AOP, it is necessary to define *what* an aspect will do, and *where* this action should be done. Many AOP and AOM techniques use the term *advice* for the action an aspect will take and *join points* for where these actions will be inserted in the primary model. *Point cuts* are used to specify more general rules of where to apply an aspect. Often, advice, joint points, and point cuts are specified as one entity, called an aspect.

There are other AOM approaches to adding security template or patterns to applications. Trillo and Rocha [53] describe an approach that describes security patterns as aspect models, and keeps them separate from the main application functionality throughout the design process (using AOP techniques also keeps them separate through implementation). Implicit in such an approach is that interactions between aspects will not be found since the aspects are never combined for analysis. If AOP techniques are used, interactions will only be found in the running system. Our approach composes multiple aspects with the primary model, so that we can analyze the entire system and identify interactions between security mechanisms. Such an analysis at design time is particularly useful when the different mechanisms interact with each other and trade-off decisions must be made because the properties ensured by the different mechanisms cannot be simultaneously enforced.

Other researchers have proposed languages based on UML that allows for the specification and analysis of system and security properties. SecureUML [6, 11] is one such language that specifies system and security models using an UML profile. Model transformations are used to generate system code from the models that includes a security infrastructure. The interactive theorem prover Isabelle is used to analyze and verify the model prior to code generation. Our work differs in three ways. First, SecureUML is specifically meant for access control and authorization properties. Our work is more generalized to check to any sort of security property as long as it can be expressed in first-order logic. Secondly, Alloy Analyzer is not a theorem prover; it does not require any user guidance to generate the result of the analysis. In Alloy a property is specified in first-order logic and the analyzer automatically tries to find an instance of the model that violates the property. If an instance is found, Alloy provides counterexamples that will help the application developer understand the flaws of the protocol. Finally, we are interested in security property validation in abstract specifications of system models and have not investigated code generation from these models. It might be possible to use model transformations to refine the abstract platform independent model to a platform specific one and partially generate code for the system. However this was not the initial aim of this work and remains for future investigation.

UMLsec [32] is a UML profile that allows a developer to specify security requirements and specifications in a system design. It has an accompanying toolset that allows the design to be verified as having the required security properties in the presence of a particular type of attacker. UMLsec provides several common adversary models that can be used to ensure that the system has the necessary security properties. Such verification can be done using a theorem prover such as e-SETHEO [39]. As described earlier, the approach adopted by Alloy, unlike theorem provers, is fully automated. For a detailed description of the differences between Alloy and theorem provers, please refer to Jackson [30, Ch.5.1.1]. Our method also differs from the UMLsec approach in that we require a specific misuse model to be composed with the system model prior to analysis. UMLsec embodies the misuse in a more generalized adversary model.

Researchers have also focused on the analysis of UML models. One method to conduct analysis of complex systems relies on formalizing the UML. Evans et al. [18] propose the use of Z [59] as the underlying semantics of UML models. This is a natural choice, as Z has been used to formally model and verify a wide variety of systems. However, conventional Z does not provide any support for object oriented constructs. Kim [36] uses Object-Z [48], an object oriented extension of Z, which is better suited to formalizing UML models. Kim also makes use of

MDA technology [36] to transform UML models to Object-Z, in order to facilitate analysis of the models.

There are also a number of tools that support analysis of UML models. For example, the UML Specification Environment (USE) [25] is a snapshot generator. USE can check whether a specific instance of a UML model conforms to the OCL constraints of the model. This method requires the manual generation of the instances to be checked. Gogolla et al. [25] propose the use of a scripting language (ASSL), which can be used to automate the instance generation process. Such instances can be checked for the conformance to the model. However, creation of such instances is directed towards validation. On the other hand our approach, which makes use of the Alloy Analyzer, automatically checks the state space exhaustively on all possible valid instances of the model up to user specified scope, for a counterexample, hence resulting in verification.

Another group of UML tools rely on theorem provers for conducting the analysis. For example the KeY tool [7] formalizes OCL with the help of dynamic logic [26] and provides an interactive theorem prover environment for the analysis. HOL-OCL [10] is another tool that transforms OCL to HOL formulas that can be analysed by the Isabelle [41] theorem prover. All these methods require guidance and special expertise to operate the theorem prover environment. Most application developers lack such expertise. On the other hand, our method relies on SAT-solvers and as a result the analysis if fully automated.

Finally there are a number of UML tools, which are oriented towards checking the runtime conformance of an implementation. For example, the Dresden OCL toolkit [27] can generate  java code from UML class diagrams enriched with OCL constraints. The code generated checks at runtime whether the implementation violates any of the constraints. For an extended study of this category of tools the reader is referred to [12]. In contrast to such approaches, our method deals with the analysis of the system at an abstract level before the implementation. As a result our method can expose bugs and security issues of a system, early in the development process before the model is refined enough to be implemented and executed.

Another suitable choice of formalism for UML model is Alloy, which is specifically designed for Object Oriented design. As described in Section 3, our approach is based on using Alloy, by transforming UML class diagrams and OCL into Alloy models. Massoni et al. [37] also transform UML Class diagrams to Alloy in order to analyze structural properties of UML models. However, their work is mainly focused on static aspects and, unlike our method, does not deal with the dynamics of UML models.

Mostefaoui and Vachon [38] also use Alloy to analyze behavioral interactions of aspect-oriented models. Base behavior is transformed into an Alloy model along with joint points, point cut specifications, and aspect behavioral advice. Alloy analyzer is used to verify the presence of the required base behavior, in addition to aspect behavior. The technique was developed to identify interactions or conflicts between multiple aspects being applied to a base behavior at the same time. Aspects in this work related to additional features being added to a base system. The multiple aspects are composed into a single Alloy entity and then woven with the base behavior before, after, or both before and after the join point. Our work differs in that composing a security-treated system model with different attack models prior to transformation and analysis provides assurance that the system design is resilient to particular forms of attack identified through risk analysis. Also, the work we describe provides a methodology for designing secure applications, and the use of Alloy to analyze for security issues is just one part (albeit a crucial one) of the overall methodology.

A number of protocols and systems have been modeled and analyzed in Alloy. The COM architecture [29] and the consistency of the International Naming Scheme (INS) [33] are two of them. There are also a number of systems originally modeled in UML, which have been manually transformed to Alloy for the purpose of analysis. Alloy has also been used for partially analyzing the run-time configuration management of an Asynchronous Transfer Mode/Internet Protocol (ATM/IP) Network Monitoring System [21]. Dennis et al [16] have used Alloy to analyze a radiation therapy machine, exposing major flaws in the original UML design of the system. Unlike these case studies, our work supports fully automated transformation of UML models enriched with OCL constraints, to Alloy, for the purpose of analysis.

Vela et al. [56] focus on model driven development of secure XML databases. Specifically, the authors address authorization and audit properties. While they use model transformations to convert a platform independent model to a platform specific one, our approach is using model transformation to create formalism for analysis. Unlike our work, they do not apply any analysis techniques on the UML design of the system.

The idea of using aspects for designing secure systems has been presented in our earlier works [22, 23, 24]. In our very early work [22, 24] we developed the concepts needed to specify and compose security aspects with a primary model. At that time we utilized UML1.4 static class diagrams and UML dynamic collaboration diagrams to specify behavior. Other work in our group [19, 20, 50)] provided formal notations (using newer versions of UML), specifications, and composition algorithms that we rely on in the research presented in this paper. In our later paper [23], we outlined the steps needed to verify security properties (embodied in an aspect) after composition with a primary design model. The analysis was informal and done manually. Although informal analysis is easy to perform and requires fewer resources, it cannot give adequate assurances especially for complex systems. The work presented in this paper extends the earlier work by showing how a real-world complex system can be formally analyzed to ensure that a given attack does not compromise the system resources. The formal analysis done in this paper is automated and ensures that problems have not been overlooked.

## 8. Conclusion

In this paper, we propose a methodology for developing secure systems that are resilient to given attacks. We first perform risk assessments to identify the types of attacks that are typical for such applications. We show how to evaluate the application against such attacks. If the results of this evaluation indicate that the assets may be compromised, then some security mechanism must be incorporated into the application. The resulting system is then formally analyzed to ensure that it is indeed resilient to the given attack. We validated our approach on a real-world e-commerce application.

Our approach does not detect new vulnerabilities but it can be used for assessing whether a given vulnerability poses sufficient risk that necessitates its mitigation. The main benefit of our approach is that it simplifies the design of complex systems. The primary models and the aspects can be analyzed in isolation to ensure that individually they satisfy the functional and security properties respectively. The models can be composed and the analysis of the composed model will give assurance that the resulting system also satisfies the properties. Another benefit of our approach is that it allows one to experiment with various security mechanisms to see which one is most suitable for preventing a given attack on the application. When a system is required to enforce different security properties, multiple aspects must be integrated with the application. This will allow one to study and formalize the interaction between aspects.

Our on-going and future work concentrates efforts in three areas. We are in the process of developing detailed algorithms to support the abstraction of complex UML diagrams and their conversion to OCL specifications, so that the approach can be automated. This ability will aid developers using the approach by reducing the chances that simplifying abstractions made by the developer leave out crucial items for the analysis. We are also investigating the broader applicability of the approach to other security mechanisms that are more appropriately specified by UML diagrams other than sequence diagrams. Finally, we are also investigating application of the approach to other stages in the development lifecycle of complex software systems, especially to the requirements phase.

## Acknowledgements

This work was partially supported by AFOSR under Award No. FA9550-04-1-0102.

## References

[1] Alloy website. http://alloy.mit.edu.

[2] AOSD-EUROPE, Survey of Analysis and Design Approaches, Document ID AOSD-Europe ULANC-9, http://ww.early-aspects.net, 2005.

[3] ArgoUML website: http://argouml.tigris.org/

[4] Australian/New Zealand Standards. AS/NZS 4360:2004 Risk Management, 2004.

[5] Australian/New Zealand Standards. HB 436:2004 Risk Management Guidelines – Companion to AS/NZS 4360:2004, 2004.

[6] Basin, D., Doser, J., and Lodderstedt, T. Model driven security: From UML models to access control infrastructures. In ACM Transactions on Software Engineering and Methodology, volume 15, issue 1, pages 39-91, January 2006.

[7] Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, Vol. 4334 of LNCS. Springer-Verlag (2007).

[8] Bordbar, B. and Anastasakis, K. MDA and Analysis of Web Applications. In Trends in Enterprise Application Architecture (TEAA) 2005, volume 3888 of Lecture notes in Computer Science, pages 44-55, Trondheim, Norway, 2005.3

[9] Bordbar, B. and Anastasakis, K.. “UML2ALLOY: A tool for light-weight modelling of discrete event systems”. In Nuno Guimarães, Pedro T. Isaías, editors, *AC 2005, Proceedings of the IADIS International Conference on Applied Computing (IADIS 2005)*, 2005, Volume 1, ISBN 972-99353-6-X, pages 209-216.

[10] Brucker, A.D.: An Interactive Proof Environment for Object-oriented Specifications. ETH, Zurich (2007) ETH Dissertation No. 17097.

[11] Brucker, A.D., Doser, J. and Wolff, B. A model transformation semantics and analysis methodology for SecureUML. In Model Driven Engineering Languages and Systems 9th International Conference, MoDELS 2006, volume 4199 of Lecture Notes in Computer Science, pages 306-320, Genova, Italy, 2006. Springer Berlin.

[12] Cabot, J., Teniente, E.: Constraint Support in MDA tools: a Survey. European Conference on Model-Driven Architecture 2006, Vol. 4066 of LNCS. Springer (2006) 256-267.

[13] Clarke, S., Extending standard UML with model composition semantics. Science of Computer Programming, 44(1):71.100, 2002.

[14] Clarke, S. and Banaissad, E., Aspect-oriented analysis and design. Addison-Wesley Professional, Boston, 2005.

[15] CORAS (2000–2003). IST-2000-25031 CORAS: A Platform for risk analysis of security critical systems. http://sourceforge.net/projects/coras, Accessed 18 February 2006.

[16] Dennis, G., Seater, R., Rayside, D., Jackson, D.: Automating commutativity analysis at the design level. In: ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, ACM Press (2004) 165-174

[17] Dimitrakos, T. & Ritchie, B. & Raptis, D. & Aagedal, J. O. & den Braber, F. & Stølen, K., & Houmb, S. (2002). Integrating model-based security risk management into Ebusiness systems development: The CORAS approach. In Monteiro, J., Swatman, P., and Tavares, L., editors, Second IFIP Conference on E-Commerce, E-Business, E-Government (I3E 2002), volume 233 of IFIP Conference Proceedings, 159-175. Kluwer.

[18] Evans, A., France, R., Grant, E.: Towards Formal Reasoning with UML Models. In: Proceedings of the OOPSLA'99 Workshop on Behavioral Semantics. (1999)

[19] France, R., Kim, D.-K., Ghosh, S., and Song, E., A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, 2004.

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

[21] Georg, G., Bieman, J., France, R.B.: Using Alloy and UML/OCL to Specify Run-Time Configuration Management: A Case Study. In Evans, A., France, R., Moreira, A., Rumpe, B., eds.: Practical UML-Based Rigorous Development Methods-Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001, in Toronto, Canada. Volume P-7 of LNI., German Informatics Society (2001) 128-141

[22] Georg, G., France, R., Ray, I., "An Aspect-based Approach to Modeling Security Concerns", Proceedings of the Workshop on Critical Systems Development with UML (CSDUML '02), held in conjunction with the 5th International Conference on the Unified Modeling Language (UML '02), pages 107-120, 2002.

[23] Georg, G., Houmb, S. H. and Ray, I., "Aspect-Oriented Risk-Driven Development of Secure Applications", in Damiani, Ernesto; Liu, Peng (Eds.), Proceedings of the 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security, LNCS Vol. 4127, p 282-296, Springer-Verlag, 2006.

[24] Georg, G., Ray, I., France, Rl, "Using Aspects to Design a Secure System", Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '02), pages 117-126, 2002.

[25] Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling 4(4) (2005) 386-398

[26] Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000).

[27] Hussmann, H., Demuth, B., Finger, F.: Modular Architecture for a Toolset Supporting OCL. UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference, Vol. 1939 of LNCS. Springer, York, UK (2000) 278-293.

[28] ISO 15408:1999 Common Criteria for Information Technology Security Evaluation. Version 2.1, CCIMB–99–031, CCIMB-99-032, CCIMB-99-033, August 1999.

[29] Jackson, D., Sullivan, K.: COM revisited:tool-assisted modelling of an architectural framework. In: 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), San Diego, CA (2000)

[30] Jackson, D. Software Abstractions: Logic, Language, and Analysis. The MIT Press, London, England, 2006.

[31] Jacobson, I.: Case for aspects . Parts I, II. Software Development Magazine, pages 32-37, October 2003. Pages 42-48, November 2003.

[32] Jürjens, J.: *Secure Systems Development with UML*. Springer, Berlin Heidelberg, New York, 2005.

[33] Khurshid, S., Jackson, D.: Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In: ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering, Washington, DC, USA, IEEE Computer Society (2000) 13

[34] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W., Getting started with AspectJ. Communications of the ACM, 44(10):59-65, 2001.

[35] Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture - Practice and Promise. The Addison-Wesley Object Technology Series. Addison-Wesley (2003)

[36] Kim, S.K.: A Metamodel-based Approach to Integrate Object-Oriented Graphical and Formal Specification Techniques. PhD thesis, University of Queensland, Brisbane, Australia (2002)

[37] Massoni, T., Gheyi, R., Borba, P.: A UML class diagram analyzer. In: Third Workshop on Critical Systems Development with UML, UML 2004, Lisbon, Portugal (2004) 100-114

[38] Mostefaoui, F. and Vachon, J., Design-level detection of interactions in aspect-UML models using Alloy, Journal of Object Technology Special Issue on Aspect-Oriented Modeling, Vol 6, No. 7, 2007.

[39] Moser, M., Ibens, O., Letz, R., Steinbach, J., Goller, C., Schumann, J., Mayr, K.: SETHEO and E-SETHEO - The CADE-13 Systems.

[40] Muller, P.-A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., and Jézéquel, J.-M.. Model-driven

analysis and synthesis of concrete syntax. In Model Driven Engineering Languages and Systems 9th International Conference, MoDELS 2006, volume 4199 of Lecture Notes in Computer Science, pages 98-110, Genova, Italy, 2006. Springer Berlin.

[41] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, Vol. 2283 of Lecture Notes in Computer Science. Springer-Verlag (2002).

[42] Nunes, I.: An OCL Extension for Low-Coupling Preserving Contracts. “UML” 2003 - The Unified Modeling Language, Vol. 2863 of LNCS. Springer (2003) 310-324.

[43] Object Management Group. Meta Object Facility (MOF) Core v. 2.0. Document Id: formal/06-01-01. http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

[44] Object Management Group. Object Constraint Language Version 2.0. Document id: formal/06-05-01. http://www.omg.org/cgi-bin/doc?formal/06-05-01.

[45] Object Management Group. Unified Modeling Language Infrastructure Specification, v2.0. Document id: formal/05-07-05. http://www.omg.org/cgi-bin/doc?formal/05-07-05.

[46] Object Management Group. XML Metadata Interchange (XMI), v2.0. Document id: formal/05-05-01. http: //www.omg.org/cgi-bin/doc?formal/05-05-01.

[47] OCLE, Object Constraint Language Environment, http://lci.cs.ubbcluj.ro/ocle, accessed Nov 2007.

[48] Smith, G. The Object Z Specification Language. Advances in Formal Methods. Springer (2000)

[49] Stølen, K., den Braber, F., Dimitrakos, T., Fredriksen, R., Gran, B. A., Houmb, S. H., Stamatiou, Y. C., and Aagedal, J. Ø.. Model.based risk assessment in a component-based software engineering process: The CORAS approach to identify security risks. In Franck Barbier, editor, Business Component-Based Software Engineering, pages 189.207. Kluwer, 2002. ISBN: 1.4020.7207.4.

[50] Straw, G., Georg, G., Song, E., Ghosh, S., France, R., and Bieman, J.. Model composition directives. In T. Baar, A. Strohmeier, A. Moreira, and S Mellor, editors, *UML*, volume 3273 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2004.

[51] TLS: Network Working Group. The TLS Protocol Version 1.0, RFC 2246, January 1999.

[52] TLSWG. SSL 3.0 Specification (1996). http://wp.netscape.com/eng/ssl3.

[53] Trillo, C. P.- and Rocha, V., Architectural Patterns to Secure Applications with an Aspect Oriented Approach, Proceedings of the 5th Latin American Conference on Pattern Language of Programming, page 89-105, 2005.

[54] UML2Alloy website: http://www.cs.bham.ac.uk/~bxb/UML2Alloy.html

[55] Vaziri, M. and Jackson, D. Some Shortcomings of OCL, the Object Constraint Language of UML. In Technology of Object-Oriented Languages and Systems (TOOLS 34'00), pages 555{562, Santa Barbara, California, 2000.

[56] Vela, B., Fernandez-Medina, E., Marcos, E., and Piattini, M. Model Driven Development of Secure XML Databases. SIGMOD Record, volume 35, number 3, pages 22-27, September 2006.

[57] Whittle, J. and Araújo, J., Scenario Modelling with Aspects, IEE Software Proceedings, Vol 151, Issue 4, pages 157-171, 2004.

[58] Wimmer, M. and Kramler, G. Bridging grammarware and modelware. In Jean-Michel Bruel, editor, MoDELS Satellite Events, volume 3844 of Lecture Notes in Computer Science, pages 159-168. Springer. 2006.

[59] Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall, Upper Saddle River, NJ, USA (1996) II

## Appendix – Alloy Model and Assertions

```
module FixedTLSMisuseModel

sig ActiveClient{
at: Attacker,
iNonce: INonceType,
reciNonce: INonceType,
sname: NameType,
certName: NameType,
recName: NameType,
cSessKey: SessionKeyType,
msg: EncrMessType,
loginAborted: ResultType,
resultPage: PageType,
cPubKey: PublicKeyType,
recPubKey: PublicKeyType }

sig Attacker{
ac1: ActiveClient,
lm: LoginManager,
aSessKey: SessionKeyType,
pubKey: PublicKeyType,
certKey: PublicKeyType,
certName: NameType,
loginAborted: ResultType,
resultPage: PageType }

sig LoginManager{
at1: Attacker,
upm: UProfileManager,
sessKey: SessionKeyType,
certName: NameType,
recInonce: INonceType,
name: NameType,
cKey: PublicKeyType,
cCertKey: PublicKeyType,
prof: ProfileType,
msg: EncrMessType,
resultPage: PageType }

sig UProfileManager{
lm1: LoginManager,
name: NameType,
prof: ProfileType }

abstract sig ResultType { }
one sig r_true extends ResultType {}
one sig r_false extends ResultType {}

abstract sig SessionKeyType { }
one sig symmKey extends SessionKeyType {}
one sig nullKey extends SessionKeyType {}

abstract sig EncrMessType { }
one sig encrCont extends EncrMessType {}
one sig nullMess extends EncrMessType {}

abstract sig INonceType { }
one sig cINonce extends INonceType {}
one sig nullNonce extends INonceType {}
```

```
abstract sig PageType { }
one sig homePage extends PageType {}
one sig visitorPage extends PageType {}
one sig nullPage extends PageType {}

abstract sig NameType { }
one sig sName extends NameType {}
one sig aName extends NameType {}
one sig cName extends NameType {}

abstract sig PublicKeyType { }
one sig aPublicKey extends PublicKeyType {}
one sig cPublicKey extends PublicKeyType {}

abstract sig ProfileType { }
one sig cProfile extends ProfileType {}
one sig nullProfile extends ProfileType {}

fact ac1_at { ac1 = ~at }
fact at1_lm { at1 = ~lm }
fact lm1_upm { lm1 = ~upm }

fact {My11To11(at ,ActiveClient ,Attacker)}
fact {My11To11(ac1 ,Attacker ,ActiveClient)}
fact {My11To11(lm ,Attacker ,LoginManager)}
fact {My11To11(at1 ,LoginManager ,Attacker)}
fact {My11To11(upm ,LoginManager, UProfileManager)}
fact {My11To11(lm1 , UProfileManager, LoginManager)}

pred My11To11(r:univ -> univ, t: set univ, u: set univ){
all x:t|one y:u|x.r=y
all y:u|one x:t| x.r=y }

pred main(){
all ac: ActiveClient | ac.iNonce = cINonce &&
ac.sname = sName &&
ac.cPubKey = cPublicKey &&
ac.certName = cName &&
recLoginFromAC(ac.at)   }


pred recContLFromAttacker(ac': ActiveClient){
ac'.recName = ac'.at.lm.certName &&
((ac'.recName != ac'.sname) =>
       abortLoginAttempt(ac')
  else(
       ac'.reciNonce = ac'.at.lm.recInonce &&
       ((ac'.reciNonce != ac'.iNonce) =>
              abortLoginAttempt(ac')
              else(
                     ac'.recPubKey = ac'.at.lm.cKey &&
                     ((ac'.recPubKey != ac'.cPubKey) =>
                            abortLoginAttempt(ac')
                            else(
                                   ac'.cSessKey = ac'.at.aSessKey &&
                                   ac'.msg = encrCont &&
                                   recMsgFromAC(ac'.at.lm)))))))}

pred abortLoginAttempt(ac': ActiveClient){
ac'.loginAborted = r_true &&
ac'.resultPage = nullPage &&
ac'.at.loginAborted = r_true &&
ac'.at.resultPage = nullPage }

pred recResFromAttacker(ac': ActiveClient){
ac'.resultPage = ac'.at.lm.resultPage &&
ac'.loginAborted = r_false }

pred recLoginFromAC(at': Attacker){
at'.pubKey = aPublicKey &&
at'.certKey = aPublicKey &&
at'.certName = at'.ac1.certName &&
recLoginFromAttacker(at'.lm) }

pred recContLFromLM(at': Attacker){
at'.aSessKey = at'.lm.sessKey &&
recContLFromAttacker(at'.ac1) }

pred recResFromLM(at': Attacker){
(
  (at'.lm.cKey = aPublicKey) =>
       (at'.loginAborted = r_false &&
       at'.resultPage = at'.lm.resultPage )
  else (
       at'.loginAborted = r_false &&
       at'.resultPage = nullPage ))&&
recResFromAttacker(at'.ac1)}

pred recLoginFromAttacker(lm': LoginManager){
lm'.certName = sName &&
lm'.recInonce = lm'.at1.ac1.iNonce &&
lm'.name = lm'.at1.certName  &&
lm'.cKey = lm'.at1.pubKey &&
lm'.cCertKey = lm'.at1.certKey &&
((lm'.cKey != lm'.cCertKey) =>
       abortLoginAttempt(lm'.at1.ac1)
  else(
              lm'.sessKey = symmKey &&
              recContLFromLM(lm'.at1)))}

pred recMsgFromAC(lm': LoginManager){
lm'.msg = lm'.at1.ac1.msg &&
((lm'.msg = encrCont) =>
       getProfile(lm'.upm)
  else
       abortLoginAttempt(lm'.at1.ac1))}

pred sendResult(lm': LoginManager){
lm'.prof = lm'.upm.prof &&
((lm'.prof = cProfile) =>
       lm'.resultPage = homePage
  else
       lm'.resultPage = visitorPage ) &&
recResFromLM(lm'.at1) }

pred getProfile(upm': UProfileManager){
upm'.name = upm'.lm1.name &&
((upm'.name = cName) =>
       upm'.prof = cProfile
  else
       upm'.prof = nullProfile ) &&
sendResult(upm'.lm1) }

pred exec(){
main() &&
some Attacker &&
some UProfileManager &&
some LoginManager &&
some ActiveClient }

fact{exec()}

assert noLogin{
all ac:ActiveClient | ac.loginAborted = r_true
}

assert assert1{
all ac:ActiveClient |
((ac.loginAborted = r_false) =>
       (ac.cSessKey = symmKey &&
              ac.at.lm.sessKey = symmKey &&
              ac.at.aSessKey != symmKey ))}
```

```
assert assert2{
all ac:ActiveClient |
((ac.loginAborted = r_true) =>
       (ac.resultPage = nullPage &&
              ac.at.resultPage = nullPage))}

assert assert3{
all ac:ActiveClient |
((ac.loginAborted = r_false) =>
       ((ac.resultPage = homePage ||
       ac.resultPage = visitorPage) &&
       ac.at.resultPage = nullPage ))}
```