Secure Provision and Consumption in the Internet of Services

FP7-ICT-2009-5, ICT-2009.1.4 (Trustworthy ICT)

Project No. 257876

www.spacios.eu

Deliverable D2.3.1
Definition and Description of Security Goals

Abstract

This deliverable concerns the definition of security goals in SPaCIoS. Its main contributions are the formalization in ASLan++ of the security goals derived from our problem cases, and a feasibility study on verifying the goals, in the context of their corresponding problem cases, using SATMC and other back-ends of the AVANTSSAR Platform. Moreover, we discuss the issues of asynchronous testing and testing with un-observable events.

Deliverable details

Deliverable version: *v1.0* Classification: *public*
Date of delivery: *31.03.2012* Due on: *31.03.2012*
Editors: *all* Total pages: *62*

Project details

Start date: *October 01, 2010* Duration: *36 months*
Project Coordinator: *Luca Viganò*
Partners: UNIVR, ETH Zurich, INP, KIT, UNIGE, SAP, Siemens, IeAT

(this page intentionally left blank)

Contents

List of Tables

1 Introduction

This deliverable concerns the definition of security goals in SPaCIoS. The security goals are derived from our problem cases, which are described in the SPaCIoS Deliverables D.2.1.1 [9] and D5.1 [10]. Here, we give a formal specification of the security goals in ASLan++. These specifications are therefore expressed as formulas in the linear temporal logic (LTL), or using the keywords that are envisioned for defining goals in ASLan++ such as assertions and confidentiality sets; see [2] for more details. The latter specifications can in general be translated into LTL; however, they are sometimes amenable to more efficient verification techniques in the AVANTSSAR back-ends.

After specifying the security goals, we have conducted a feasibility study on verifying the goals, in the context of their corresponding problem cases, using SATMC and other AVANTSSAR back-ends. We report on the feasibility study in this deliverable. The key finding of the study can be summarized as follows. In a number of cases, we observed that our initial formalization of security goals cannot be handled by the verification back-ends. We however realized that it is sometimes possible to transform such goals into "equivalent" goals that can be verified using the AVANTSSAR platform. We discuss such transformations in the following sections.

In addition to the formalization of the security goals and the investigation of the feasibility of their verification, in this deliverable we also discuss the issues of asynchronous testing and testing with un-observable events.

Structure of the document. The deliverable is structured as follows. In Section 2, we elicit the security goals arising from our problem cases. These security goals are formalized in Section 3, where we also study whether they can be verified using the AVANTSSAR platform. In Section 4, we discuss the issues of asynchronous testing and testing with un-observable events. Section 5 concludes the deliverable. In appendix A, we recap the syntax and semantics of LTL. In appendix B, we present the formal model of Client Authentication in ACF flow for OAuth 2.0.

2 Elicitation of the security goals arising from the problem cases

2.1 WebGoat security goals

In this section, we present the different security goals that arise from the WebGoat application scenario.

2.1.1 Authenticity

The two lessons in WebGoat that directly address authentication problems are based on low level vulnerabilities (i.e., vulnerable encryption functions), which are out of the scope of SPaCIoS. However, several lessons rely on a role-based access control (RBAC) system. If someone can bypass the authentication system, then other security properties can be violated. For example, when a malicious user mounts a XSS attack, he is able to execute a script with the victim's privileges, therefore violating an authenticity goal.

2.1.2 Authorization

Several lessons aim at bypassing a role-based authorization system to access confidential data by an unauthorized user. For example, in the stored cross-site scripting (XSS) lesson, a user can store in his profile a script that will be executed by anyone that views this profile. Thus, a malicious user can craft a script that will execute with all the permissions of the victim, who can be anybody that has access to his profile.

In WebGoat, there are two kinds of authorization goals:

- Business layer access control: A user `U` can do a limited set of actions according to which roles he belongs to.
- Data layer access control: a confidential data `D` is accessible only by a user `U` that belongs to a set of authorized users for `D`.

2.1.3 Confidentiality

In most of WebGoat lessons, the purpose of the attacker is to find a way to view confidential data. The ways to achieve this are extremely copious and diverse: bypassing the authentication request, sniffing user credentials or stealing a user session, injecting (SQL or JavaScript) code to lead either the server or the user to reveal confidential data. However, the security goal is always the same: a confidential data `D` must remain secret between a set of authorized users. In other words, if a user `U` access a confidential data `D`, then `U` must belong to the set of authorized users for `D`.

2.1.4 Integrity

Certain values must not be altered in an unintended/unauthorized way. For example, assume that a certain condition depends on the value of some parameter `x`, and that if a test performed with a certain value `x` passes, a particular action, which depends on `x`, is executed. Then, it should not be possible for an attacker to change the value of `x` between the test and the action execution. This notion of atomicity is crucial in Web applications and its violation can lead, for example, to race condition flaws.

2.2 SAML 2.0 SSO security goals

The protocol participants are a user using a web browser (C), a service provider (SP), and an identity provider (IdP).

Authentication Authentication involves C and SP. In general, we define that X authenticates Y on Z iff at the end of the protocol run X believes it has been talking with Y and they agree on the value of Z. In this specific case, we define a mutual authentication security goal:

- *Client authentication*: SP authenticates C on the URI of the resource C wants to access to.
- *Service provider authentication*: C authenticates SP on the resource C asked for.

We would like to check whether SAML SSO ensures that if C accesses to a SP's resource within a user session represented by a token *sid* (e.g. an HTTP cookie), then an IdP issued an assertion for C to SP for the same *sid* and *sid* is still valid.

Confidentiality We distinguish between two kinds of confidentiality goals:

- *Confidentiality of resource:* The resource provided by SP to C at the end of the protocol run must be kept secret between the two entities; the address of the resource must be kept secret among C, SP, and IdP.
- *Confidentiality of Authentication data*, namely authentication requests and assertions.

Integrity It must be granted both on the resource provided by SP and on assertion request/response. Regarding the first case, when C asks for a resource, it should not be possible to change the obtained resource without C being able to notice this change. Similarly, assertions must be checked against modifications that can occur during the exchange of requests/responses between the involved entities.

2.3 OpenID security goals

The protocol participants are a user using a web browser (C), a relying party (RP) and the OpenID identity provider (OP).

Authentication OpenID has to guarantee *mutual authentication* between C and RP. Mutual authentication is defined as the conjunction of the following two authentication goals:

- *Client authentication*: a relying party RP authenticates a user C on a resource available at the address URI.
- *Relying party authentication*: a user C authenticates a relying party RP on the resource RP sends to C.

Confidentiality OpenID comprises three instances of confidentiality properties:

- *Confidentiality of resource*: the resource provided by RP to C must be kept secret between the two of them.
- *Confidentiality of authentication data*: the authentication requests and assertions must be kept secret between protocol participants.

Authorization When an OpenID run ends, RP establishes a local session *lid* with C. The authorization goal is defined as follows: whenever a user C accesses a resource r available at RP's site using a local session, it happened in the past that C's OP issued an assertion a for C to access RP, a caused *lid* and *lid* is a valid session cookie of C.

2.4 OAuth 2.0 security goals

OAuth2 security goals are distinguished between the Authorization Code Flow (ACF) and Implicit Flow (IF); see SPaCIoS Deliverable D.5.1 [10].

Authentication Authentication regards two protocol principals. They are the Client (C) and the Resource Owner (RO).

- *Client authentication in ACF.* The Authorization Server EndPoint, referred to as AS_AEP, authenticates the Client C through its full credentials (client identifier and client secret) and they agree on the values of the exchanged values of authorization code.
- *Client authentication in IF.* AS authenticates C on its partial credentials (on the client identifier only) and they agree on the value of the exchanged access token.
- *Resource Owner's authentication in ACF and IF.* The RO must be authenticated by the AS_AEP during the protocol run through RO's credentials.

Authorization OAuth authorizes C to access some protected resources, owned by RO, by providing an access token to C. It is fundamental to verify that if C is granted access to a protected resource through an access token, then an Authorization Server has previously issued that token to C, after having received the access grant from RO.

Confidentiality Confidentiality is related to different values:

- *Tokens and authorization codes.* Authorization code must be kept secret between AS, UA and C, while access and refresh token must be kept secret between the AS and C during the protocol run (only ACF).
- *Client credentials.* In ACF, Credentials must be kept confidential by the client and shared exclusively with the AS on a secure channel.
- *Resource Owner credentials.* They must be shared only with the AS (in both flows).

Integrity Access tokens, refresh tokens and authorization codes delivered by the Authorization Server endpoints (i.e., AS_AEP and AS_TEP respectively for the access code endpoint and the token endpoint) must reach the destination client without being tampered. The integrity of code and tokens is related with authentication goals, since the violation of integrity would take to the impossibility to authenticate participants.

2.5 Pervasive Retail security goals

At the security protocol level, the following security goals are expected in the Pervasive Retail Platform:

- *Mutual Authentication.* The Pervasive Retail Platform shall have mutual authentication with Consumer Client, Retailer Client, and Product Provider Client. During the message passing process, channels with confidentiality and integrity properties are used.
- *Secrecy.* In the Individual Offer scene, the offer provided by the Pervasive Retail Platform shall be a secret between the platform and the consumer, unless it is exposed by the consumer to the retailer. In the coupon scene, the coupon made by Product Provider to the consumer through the Pervasive Retail Platform shall be also a secret, unless it is exposed by the consumer to the retailer.
- *Non-repudiation.* In the individual offer scene, the retailer shall not deny the offer made for the consumer. There is another case of non-repudiation in the Pervasive Retail application scenario. When consumer receives a coupon from Product Provider, Product Provider cannot deny it has issued the coupon. In addition to this, since there is a trust relationship (e.g., established by some agreement, between Product Provider and Retailer), Retailer will accept the coupon presented by the consumer.

At the business process level, the following security goals are expected in the Pervasive Retail Platform:

- *Authorization.* All the data access shall be controlled by authorization. For example, the purchasing behavior information, e.g., the relationship between the purchasing decision and the coupon, is accessible for Product Provider. At the same time, the exact selling price information in purchasing behavior is not accessible for Product Provider.
- *Privacy.* In the coupon scene, the consumer can decide whether to expose his/her location information to the Pervasive Retail Platform and Product Provider.
- *Need-to-Know.* In the coupon scene, consumer identity information such as name and gender is not necessary for the Product Provider to make a decision about coupon, so it shall not be sent to and known by the Product Provider.

2.6 eHealth security goals

Since storage and communication of data in an EHR contains sensitive personally identifiable information (PII) within patient data, specific security goals have to be considered for every function.

The eHealth Record scenario can be characterized by several security requirements like *Privacy*, *Confidentiality*, and *Authenticity* of Electronic Health Records (EHRs). In order to be considered compliant to these requirements, the abstract model must meet a set of security properties. Subsequently, those properties will be also tested in the real system by means of test cases.

Although there are no complete models of the EHR case studies at this time, it is nevertheless possible to define a list of relevant goals that any such model should meet.

At a high level, the security requirements for health records and systems are: the confidentiality of patient data must be enforced, the authenticity and integrity of EHRs must be secured, and the system and processes must be reliable and available. Moreover, special scenarios, like the *user consent* to forward information to other parties must be supported. In more detail, our goals are:

- *Confidentiality:* In general, both sensitive medical data transmitted over the network and data stored should be confidential. In particular, patient data should be confidential against third parties unless specific *patient-consent* is given. This is also the case when, for a second opinion, a further medical practitioner obtains access to the EHRs of the patient (except in emergency or in the case of statutory exemptions). Moreover, Patient records must be protected from eavesdropping during communication and this usually implies that communication must be encrypted. Additionally, EHRs should be protected at rest, when stored on servers, on devices or on clients.

  Particularly in the case of eHealth mash-ups, there is a need for special care in order to achieve this goal: since different applications work on the same data space, it is important that the separation techniques in place do enforce, as expected, that the applications cannot access the data of other applications, outside of the defined interfaces.

- *Privacy:* In addition to the more general goal of confidentiality, we plan to consider the following specific and stronger privacy goals. Although it is not clear whether we will be able to fully model all of them in ASLan++, we want to have a diverse collection of goals as a starting point for future formalization efforts.

1. *User Choice and Consent:* In our case, the patient has the choice to approve that some of his medical personal information is being sent to another party for a particular purpose, like a second opinion or for research purposes. Additionally, the patient can allow monitoring devices (or other medical equipment) to autonomously add information to his records for a limited period of time. The required authorization, to be provided by the patient, is called *Patient Consent*. The corresponding goal can be expressed with an LTL goal, using the "Globally" operator (`[]` in ASLan++), having implemented in the model suitable access control policies via Horn clauses and dynamic facts, or directly coding the policy as conditions (guards) in ASLan++.
2. *Notification:* the patient has to be informed about access events to his Electronic Health Records (EHRs). This is a functional goal that is likely to be abstracted in ASLan++, or can be represented as a future-time LTL formula ("always, after some event, eventually, or in a given number of steps, something happens, namely, the patient is notified").
3. *Data Retention:* a doctor that provides a second opinion in a particular case should, of course, be able to read the pertinent EHRs, but should not keep a copy of them after giving his opinion. Additionally it must be ensured that patient information is kept only as long as necessary. This goal has more of a 'usage-control' nature, and is therefore more challenging to be modeled. This goal can probably be abstracted in ASLan++, or can be represented as a future-time LTL formula ("always, after the second opinion has been given, eventually, or in a given number of steps, something happens, namely, the copy is deleted").

Other privacy properties like *Purpose of Use* (the user is informed about the intended usage of his data, and the data is used only for such purposes) and *Data Quality* (the user is able to correct any incorrect information in his data), as well as those related to Procedures, Training, Awareness and Standard of Conduct, Incident Management, Auditing and Enforcement are out of the scope of our investigation (mainly because they lie on a business-process level of abstraction and not in the software level, that is the main scope of this project).

- *Authenticity of EHRs:* It must be verified that the EHRs have been created by the claimed Person at the claimed time (this information is included in the EHR itself) and using the procedures, methods, workflows and systems approved for such purposes. As for other authenticity goals, this is typically represented by an LTL formula.

- *Integrity of EHRs:* Data stored in the EHR must remain complete and unaltered unless update operations are performed by authorized users. Their logic (workflows and processes) should also be maintained. As for other authenticity goals, this is typically represented by an LTL formula. Usually this goal is subsumed to authenticity (that is, when one proves that the data has been created by the claimed person, at the same time it is also verified that it has not been manipulated since then).

- *Accountability:* In the case of eHealth, every access to an EHR must be logged. This is particularly important when a medical practitioner accesses an EHR with relaxed access control policies in the case of emergency. The special provisions for logging activities take place can be expressed through an LTL goal. It may be sufficient to check that if an emergency occurs (implying relaxed access policies to a record), then extra logging activities must be performed (this may be represented by a special fact), and this can be modeled with an LTL goal. In addition, each time an access occurs, a log entry is also immediately created.

2.6.1 Tentative formalization

As mentioned above, currently we do not have a complete formalization of the eHealth problem case. Therefore, in the following we give a tentative formalization of the goals mentioned above. For the time being, we therefore ignore the fact that a formalization of goals requires a formal model of the problem case. The following tentative formalization nonetheless shows that the security goals of the eHealth problem case can be specified in LTL, hence expressible in ASLan++.

We will use a *key* to identify and access the EHR, and the following predicates and constructor functions will be used in modeling the desired security properties:

- $ehr(patient, key)$: the EHR of patient, referenced by the key (or index) key.

- $isdoctor(anyone)$: the entity *anyone* is a doctor.

- *stores*(*anyone*, *ehr*(*patient*, *key*)): the entity *anyone* (typically, a doctor or a nurse, but in some cases it can be also a device) creates the EHR of a patient and and stores it in the system.

- *access*(*anyone*, *ehr*(*patient*, *key*)): the entity *anyone* (typically, a doctor or a nurse) accesses the EHR of a patient with key *key*.

- *witness*(*anyone*, *key*, *ehr*(*patient*, *key*)): this event is used only for modelling purposes, in order to be able to verify (in the model) the authenticity of an EHR. For the use of the logical predicate *witness*, see also [3]. The event is generated when a health-care professional or administrative person, or a device (*anyone*) creates ("issues") an EHR, as part of his normal workflow. Thus, the event is generated when an authorized entity creates an EHR in the ways that they should be created. This is not the case if the attacker directly creates an EHR or induces another entity to create one. Moreover, the EHR *ehr*(*patient*, *key*) in the *witness* event is correctly signed by the entity *anyone* at the moment of creation.

- *consent*(*patient*, *anyone*, *key*): patient's consent for a doctor, a nurse, a health-care administrator or even a device to access his/her EHR identified with key *key*. This also includes the authorization of creating a new EHR. When a patient is registered in a hospital, automatically he gives consent to selected personnel of the hospital to access his EHRs as long as he is in the hospital. To authorize further personnel or devices an explicit extra consent of the patient is required (except for emergency situations).

- *timeout*(*patient*, *anyone*, *key*): a timeout, that limits the amount of time a doctor or a device may access an EHR after a patient has consented the access. We assume that the event of providing a patient consent implicitly creates a timeout that will fire in the future, limiting the validity of the consent. This is modeled as a fact, rather than a real time event.

- *emergency*(*patient*, *doctor*): state of emergency for a patient, issued by a doctor. During the emergency, the doctor is permitted to access the EHRs of the patient. For modelling purposes, it is easier to assume that *emergency* is not a state of the system (the system does not track any emergency situation of a patient) but it is a *reason or purpose* to access the EHR. Thus a doctor may enter the reason "emergency" to access the EHR. With this view in mind, *emergency*(*patient*, *doctor*) and *access*(*anyone*, *ehr*(*patient*, *key*)) happen simultaneously.

- $notification(patient, doctor, key)$: a patient is notified that a doctor has accessed his/her EHR with key key.
- $logged(doctor, ehr(patient, key))$: the event of logging a doctor's access to an EHR.

Instead of requiring that if a doctor knows an EHR, he *has* permission to know it, we should require that *in the moment he accessed it* he *had* the corresponding permissions. The reason is that the conditions to access a certain EHR may cease to be true, while the fact that an agent knows a fact is monotone (if true, remains true forever). Thus, a doctor may access an EHR in an emergency situation, and after the emergency has been resolved, the doctor can not access any more the EHR of the patient, but of course he may still know the EHRs of the patient that he access during the emergency. Thus, our security conditions do not talk about a person *knowing* but rather *accessing* (or creating) an EHR.

The main security goals can be written as:

- $\forall anyone, patient, key\ (\Box(access(anyone, ehr(patient, key))) \implies (\neg timeout(patient, anyone, key)\ \mathbf{S}\ consent(patient, anyone, key)) \vee (emergency(patient, anyone)))$
- $\forall anyone, patient, key\ (\Box(stores(anyone, ehr(patient, key))) \implies (\neg timeout(patient, anyone, key)\ \mathbf{S}\ consent(patient, anyone, key)) \vee (emergency(patient, anyone)))$
- $\forall anyone, patient\ (\Box(emergency(patient, anyone)) \implies isdoctor(anyone)))$
- $\forall anyone, patient, key\ (\Box(access(anyone, ehr(patient, key))) \implies \Diamond notification(patient, anyone)))$
- $\forall anyone, patient, key\ (\Box(access(anyone, ehr(patient, key))) \implies \Diamond^{-1}\ \exists y\ witness(y, key, ehr(patient, key))))$
  This is an authenticity check: if someone accesses an EHR, it is indeed authentic.
- $\forall anyone, patient, key\ (\Box(access(anyone, ehr(patient, key))) \implies logged(anyone, ehr(patient, key)))))$

Notice the difference between *access* and the usual "*knows*" predicate (that we do not explicitly use here, except for the attacker) and also the difference between *stores* and *witness*. Both "*access*" and "*stores*" are system events that happen in discrete moments of time. They are not predicates that are true in intervals of time, while “*knows*” is a predicate that is true in an interval of time. In fact, the predicate is monotone: If an agent accesses a data, then the agent *knows* the data thereafter. (Note that information may be inferred without being directly accessed. For instance, a piece of data may be “known” if it is the concatenation of several pieces that are known, or if it is the result of applying a public function to known data.) Contrary to “*stores*” which is a system event (one that happens, not only in the model, but also in the real system), “witness” is a *logical or virtual* event, used only for modelling purposes: whenever an authenticated and authorized entity, *anyone*, creates an EHR for *patient*, with a given *key*, as part of his normal workflow, the event $witness(anyone, key, ehr(patient, key))$ is modeled to recall (“witness”) the fact of this creation.

2.7 Infobase security goals

We have considered several security goals concerning the confidentiality and the integrity of sensible data, and authorization constraints that has to be met by the Infobase application scenario. In this section we will provide the reader with an informal description of every goals that we formalized in the specification(s) of this scenario, while details about their formalization are described in Section 3.

Confidentiality For the design time analysis we want to consider confidentiality at two different levels. The first one is the channel level where confidentiality of data is guaranteed by the use of secure (`*->*`) or confidential (`->*`) channel. But this is not enough to ensure that confidentiality of data holds, because channel confidentiality does not implies that the endpoints of the communication are trusted, and even if they were trusted, we do not know if all channels that a particular data will go through will be confidential. So our aim is to check that some particular data are confidential in the overall system. Indeed, even if those data could be considered confidential in the model, we also want to consider confidentiality at provision and consumption time analysis, i.e., at the implementation level. In other words, in order to check confidentiality of data while probing the real system we have to consider confidentiality goal at model level even if confidentiality is given by the usage of secure channels. This allows us to use the model to generate abstract test cases violating confidentiality and thus probe the real system, after a concretization step (see [11] for details on the correspondence between abstract and concrete test cases), and check if it satisfies confidentiality.

In details, we want to check confidentiality with respect to the following data:

- *Confidentiality of authentication data (passwords)*: the first phase of the model is the authentication of the Client entity. In this phase, the Client sends to the Repository entity the credential data (namely username and password). We checks if the password sent to the Repository is confidential, using in-line secrecy goals along the lines of [2].
- *Confidentiality of contents stored in the repository*: all data stored into the repository by a user must be confidential, unless the user has explicitly allowed others to access his data or in the case the content is stored as public to anyone who has access to the repository.
- *Confidentiality of cookies*: in the Infobase scenario, users log in the system and receive a cookie which allows them to access the repository and to issue requests to the system. The cookie must be confidential otherwise an intruder could be able to reuse it in order to obtain information he is not eligible to access. As for passwords, this goal has been modeled using in-line goals.
- *Confidentiality of responses from the* Repository: in order to be sure that the intruder is not able to get any information from the Repository entity, we have modeled an authorization goal, authorization_dishonest, which entails also a check on the confidentiality of answers that a user could receive from the Repository. With authorization_dishonest we indeed checks that whenever the intruder knows a particular response message he has to be a legitimate user of the system with the appropriate rights to see the response, i.e., the intruder is also a user of the system who can issue the request that caused the response message from the Repository. This goal partially encompass the goal about confidentiality of stored data, because some responses from the repository contains data stored in the system, but this is not always the case. Indeed, by accessing to a response from the repository he is not allowed to see, an intruder could learn some other sensible information which could be reused to attack the system, e.g., a list of content present in a repository.

Integrity Similar to confidentiality, the use of *->* and *-> guarantees the authentication and the integrity of messages at channel level [2], but we want to check it in the overall system as for confidentiality (cf. Section 2.7).

In particular, we considered the integrity of request messages sent by users to the Repository entity. This goal allows us to check and detect if an intruder is able to modify a request generated by a legitimate user in order to issue requests circumventing authentication and authorization mechanisms.

Authorization Only legitimate users with suitable permissions can issue requests to the `Repository` (e.g., deletion of files). More precisely, whenever a request is performed by the `Repository` entity, a user assigned to a role with sufficient permission to issue the request must have created and issued it. We considered also the case in which the attacker is inside the system (i.e., the intruder is also a user of the system) and has the rights to issue certain requests. The `authorization_dishonest` goal cover this particular scenario.

3 Formalizing and verifying the security goals

3.1 WebGoat

3.1.1 Authenticity

Log in is done over a confidential channel using a secret password shared between the user and the server:

```
Actor ->* S : login(Actor,secret_pwd:(password(Actor,S)));
goals
  secret_pwd:(_) {U,S};
```

3.1.2 Authorization

Several authorization goals are formalized in WebGoat.

Goal extracted from the "Bypass Data Layer" lesson. Roles are defined as Horn clauses. For example, the `isAuthorizedToView` role is defined as follows:

```
symbols
  % for implicit use
  isAuthorizedToView (agent, agent): fact;
  % for explicit use
  isAuthorizedToVisit (agent, agent): fact;
  hasProfile(agent, profile): fact;
clauses
  isAuthorizedToViewOwnProfile(A,P):
  A->isAuthorizedToView(A) :- A->hasProfile(P);
  isAuthorizedToViewBProfile (A,B):
  A->isAuthorizedToView(B) :- A->isAuthorizedToVisit(B);
```

Profiles can be viewed by authorized users only:

```
goals
  secret_profiles:
  forall A P.
    [](iknows(P) & A->hasProfile(P) =>
       i->isAuthorizedToView(A));
```

This goal is translated as an attack state, which is easier to check by a model-checker.

Goal extracted from the "Bypass Business Layer" lesson. Profiles can be deleted by authorized users only: To specify the security goal we make use of three different facts:

- `U->hasDeleted(P)` is true whenever the user `U` has deleted the profile `P`. This fact is issued whenever the server processes a message to delete a profile.
- `A->hasProfile(P)` expresses that `P` is the profile of user `U`.
- `U->isAuthorizedToDelete(A)` states that user `U` is authorized to delete `A`'s profile.

```
goals
  deleted_profiles:
  forall U A P. [](U->hasDeleted(P) & A->hasProfile(P)
                  => U->isAuthorizedToDelete(A));
```

Finally the authorization goal expresses that whenever a user `U` deletes user `A`'s profile, then user `U` must be authorized to do so. This goal is also translated as an attack state.

Goal extracted from the "stored XSS" lesson. For the "stored XSS" lesson we describe several approaches how to model the security goals. We start from a simple goal and discuss its issues and how they can be addressed. Then, we present a more complex formalization.

The viewing and editing actions that are non-correctly sanitizing their data are tagged (with `nonSanitizing` action facts) in the model. In addition we make use of Horn clauses to specify events where malicious code is injected into a user profile. Using these two artifacts we describe the simple "stored XSS" security goal as follows:

```
goals
  storedXSS_on_profiles:
  forall A B P.
    []((nonSanitizedReceivedBy(P,A) & (A != B))
      => [-](!codeInjectedBy(P,B)));
```

The goal expresses that whenever a user `A` has received a profile `P` with a non-correctly sanitizing action, any other user `B` must not have injected malicious code into the same profile `P` in the past. This formula introduces some problems. One issue is that the past operator `[-]` is not handled correctly by the back-ends. A second issue is that this security goal is not sound at the theory level and may output false positives. Before we move on to a more complex security goal to address the second issue, we first address the issue with the past LTL operator. We show how the given formula can be rewritten so that the back-ends can deal with it. To deal with the past operator, we experimented with two different approaches:

1. Using assert statements.

2. Introducing timestamps.

We insert `assert` statements after each actions that view a profile to handle the ordering of view and edit actions. The assert statement says that it must not happen that we view a malicious profile with a non-correctly sanitizing view action. A non-correctly sanitizing view action is tagged with the fact `nonSanitizingViewingAction`, and the Horn clause `codeInjectedBy` is true for a malicious profile. A profile is malicious if it was edited by a dishonest user with a non-correctly sanitizing edit action. These two concepts are expresses as follows:

```
clauses
 codeInjection(P,U,A):
   codeInjectedBy(P, U) :-
     editedByWith(P,U,A) &
     nonSanitizingEditingAction(A) &
     dishonest(U);

 codeAlreadyInjected(P,U):
   codeInjectedBy(P, U) :- maliciousProfile(P);

 codeReception(P,U,A):
   nonSanitizedViewedWith(P,A) :-
     nonSanitizingViewingAction(A) &
     codeInjectedBy(P, U);
```

At the same time a fact is issued after each action that edits a profile, expressing that the profile `Profile` was edited by user `U` with the action `editProfileAction`.

```
editedByWith(Profile,U,editProfileAction);
```

Finally the following `assert` statement is used after each view action in the model:

```
assert nonMaliciousProfile:
  !nonSanitizedViewedWith(Profile,viewProfileAction);
```

Doing so, we check for each view-profile request if some malicious code was injected into this profile in the past. Code was injected in the past if the corresponding facts are set. A drawback of this approach is that the modeler has to be aware where `assert` statements have to be inserted into the model, and the security goal has to be replicated for each view action instead of being expressed once in the dedicated goal section.

Another approach to deal with the past LTL operator is to introduce timestamps. We introduce logical timestamps for every request and response. Whenever the server receives a request or sends a response, it sets a corresponding fact which includes the counter. The counter is increased for every processed request or response. The Horn clauses and facts only need to be chanced slightly to include variables `T1`, `T2` of type `nat`:

```
 clauses
  codeInjection(P,U,A,T1,T2):
    codeInjectedBy(P,U,T1) :-
      editedByWith(P,U,A,T2) &
      nonSanitizingEditingAction(A) &
      dishonest(U) &
      isStrictlyGreater(T1,T2);

  codeAlreadyInjected(P,U,T1):
    codeInjectedBy(P,U,T1) :- maliciousProfile(P);

  codeReception(P,U,A,T1):
    nonSanitizedReceivedBy(P,U,A,T1) :-
      receivedByWith(P,U,A,T1) &
      nonSanitizingViewingAction(A);

symbols
        editedByWith(Profile,U,editProfileAction,T1);
```

Finally we rewrite the first security goal (with the past LTL operator) as follows:

```
storedXSS:
  forall A B P.
    forall U1 U2 A P T1 T2.
      []((
        nonSanitizedReceivedBy(P,U1,A,T1) & (U1!=U2)) =>
        (!codeInjectedBy(P,U2,T2)));
```

What remains are false positives. To deal with this issue we have to provide further restrictions on the profile and edit-actions. Initially a profile may already contain malicious code. In addition we have to address the fact that several edit-actions may be performed. After identified these additional requirements, we introduce the following symbols and Horn clauses:

```
symbols
 receivedByWith(profile, agent, action, nat): fact;
 editedByWith(profile, agent, action, nat): fact;
 nonSanitizingEditingAction(action): fact;
 nonSanitizingViewingAction(action): fact;
 maliciousProfile(profile): fact;

clauses
 codeInjection(P,U,A,C):
  codeInjectedBy(P, U, C) :- editedByWith(P,U,A,C) &
    nonSanitizingEditingAction(A) & dishonest(U);

 codeReception(P,U,A,C):
  nonSanitizedReceivedBy(P, U,C) :-
    receivedByWith(P,U,A,C) &
    nonSanitizingViewingAction(A);

 codeAlreadyInjected(P,U, C):
  codeInjectedBy(P, U, C) :- maliciousProfile(P);
```

Using these definitions, and the intuitive semantic for `isStrictlyGreater`, we can formulate the security goal as follows: (A,B,C are agents, P is of type `Profile`, Act of type `action`, and C1, C2, C3 are timestamps of type `nat`)

```
goals
 forall A B C P Act C1 C2 C3. [](
  !(codeInjectedBy(P,B,C1) &
  nonSanitizedReceivedBy(P,A,C3) & (A != B) &
  (!(isStrictlyGreater(C3,C2) &
  isStrictlyGreater(C2,C1) & editedByWith(P,C,Act,C2) &
  (!nonSanitizingEditingAction(Act)))))
  );
```

This security goal states the following: Under the assumption that a profile contains malicious code, if this profile is viewed by a non-correctly sanitizing view-action, a correctly sanitizing edit-action must have happened in between.

This formalization of security goal does not make use of special LTL operators but uses timestamps. In addition it solves the issue with false positives. The drawback of this formalization is that we observe a state explosion so that the back-ends do not report a result.

Goal extracted from the "reflected XSS" lesson. Finally we consider a lesson about reflected XSS. Because several different forms of reflected XSS lessons exist and because they all have an influence on how they are modeled, we consider here the form-based reflected XSS. Form-based means that the user inputs the malicious code into a form that is then returned with the next response. The concrete lesson we model is a search engine for user profiles. The web application provides an input field where a search query is entered. After the server processed the query, it returns the result that includes the original query.

To formalize the security goal we use the following facts in the model to state that a response was received from the server:

```
receivedResponse(action): fact;
originsFromForm(action): fact;
```

where the parameter of the `receivedResponse` fact specifies the request (e.g., `searchProfileAction`, `loginAction`) this response belongs to. The second fact is used to tag actions that potentially contain user input data. These requests are tagged with the fact `originsFromForm`.

Together with previously introduced facts, we formalize the security goal for reflected XSS as follows:

```
goals
 reflectedXSS:
  forall Request. [](
   (receivedResponse(Request) & originsFromForm(Request))
   =>
   ((!nonSanitizingAction(Request)) |
   (!nonSanitizingResponse(Request))
   ));
```

The security goal expresses that whenever a response of a request is received, and this request contains user input data, then either the request itself or the corresponding response has to be a correctly-sanitizing action.

3.1.3 Confidentiality

See examples for secret password in Section Authenticity (`secret_pwd` goal) and for secret profiles in Section Authorization (`secret_profiles` goal).

3.1.4 Integrity

A profile must remain sanitized otherwise a user that accesses this profile could be victim of a XSS attack:

```
goals
  forall P.
  [](received(P) => SanitizedProfiles->contains(P));
```

		Model-Checker (Time in second)	
Lesson	Model	CL-AtSe	SATMC
RBAC 1	secure	✓ (0.04)	✓ (0.03)
	mutated	✘ (0.02)	✘ (0.06)
RBAC 3	secure	✓ (0.10)	✓ (3.00)
	mutated	✘ (0.02)	✘ (1.00)
reflected XSS	secure	✓ (0.35)	✓ (189)
	mutated	✘ (0.04)	✘ (0.44)
stored XSS	mutated goal v1	? (0.01)	- (600)
	mutated goal v2	✘ (0.02)	✘ (1.00)
	mutated goal v3	✘ (2.50)	- (600)
	mutated goal v4	✘ (52)	- (600)
- timeout	? inconclusive	✘ attack found	✓ no attack found

Table 1: Results of checking WebGoat security goals with each backend

3.1.5 Verification of Goals

Table 1 summarizes the results of checking the security goals with both CL-AtSe and SATMC model-checkers. CL-AtSe is run with the options `--lvl 0 --not_hc` for the models of RBAC 1 and RBAC 3 lessons and with the options `--lvl 0 --nb 2` for the reflected and stored XSS lessons. The option `--lvl 0` prevents CL-AtSe for rebuilding the protocol (which fails for our models translated by the current version of the ASLan++ translator); the option `--not_hc` activates the support for negative constraints in Horn clauses; and the option `--nb 2` increases by 1 the maximum number of role/loop iterations. SATMC is run without any additional option. For the stored XSS lesson, the original security goal (v1) makes use of past LTL operator and causes some trouble to the backends. Either the model-checker does not support LTL formula (CL-AtSe) or it does not terminate in less than 10 minutes (SATMC). This goal has been rewritten in three other different versions such that it can be translated as an attack state. SATMC (v3.4) is able to find an attack trace in the second version only, which uses an assertion. However, CL-AtSe reports an attack for all the three rewritten versions.

3.2 SAML-SSO

In this section, we describe the formalization of the security goals of SAML, starting from the informal description given in Section 2.2. All these goals have been translated into attack state instead of LTL goals because Satmc can handle more than one attack states goals but just one LTL. The validation of these goals has returned an INCONCLUSIVE response that means that with a fixed depth (tested up to 22) no attack has been found. We have also used OFMC in order to validate the specification and the final result has been NO_ATTACK_FOUND. The model is given in Appendix C

3.2.1 Client Authentication

In order to check *Client Authentication* we have modeled the channel goal SP_authn_C_on_uri. The Client entity sends the following message to the SP entity:

```
Actor -Ch_C2SP_1-> SP :
        httpRequest(get, URI,
        nil_http_element, nil_http_element);
```

Given that the Service Provider (SP) has to authenticates the client on a URI, in the Client entity we have used C_on_uri:

```
Actor -Ch_C2SP_1-> SP :
        httpRequest(get, SP_authn_C_on_uri:(URI),
        nil_http_element, nil_http_element);
```

Then at the end of the SP entity, SP_authn_C_on_uri refer to the security goals that must be met and may not be violated:

```
SP_authn_C_on_uri:(URI) := URI;
```

The channel goal SP_authn_C_on_uri checks that SP authenticates Client:

```
SP_authn_C_on_uri:(_) C *->  SP;
```

3.2.2 Service Provider Authentication

In order to check *Service Provider Authentication* we have modeled the channel goal SP_on_resource.

```
SP_on_resource:(_) SP *->  C;
```

The SP entity sends the following message to the SP entity:

```
Actor -Ch_SP2C_2-> C  :
        httpResponse(code_200, nil_agent,
        nil_http_element, SP_on_resource:(Resource));
```

And the Client entity receives it catching the Resource value.

```
AnySP -Ch_SP2C_2-> Actor :
        httpResponse(code_200, nil_agent,
        nil_http_element, SP_on_resource:(?Resource));
```

3.2.3 Confidentiality of Resource

We define the following goal:

```
SP_on_resource:(_) {SP,C};
```

to check that the `Resource` is known by Security Provider and Client only. Any time that `Resource` is used we label it with `SP_on_resource:()` in order to check confidentiality of that. The `SP` entity sends the following message to the `Client` entity:

```
Actor -Ch_SP2C_2-> C  :
        httpResponse(code_200, nil_agent,
        nil_http_element, SP_on_resource:(Resource));
```

And the `Client` entity receives it catching the Resource value.

```
AnySP -Ch_SP2C_2-> Actor :
        httpResponse(code_200, nil_agent,
        nil_http_element, SP_on_resource:(?Resource));
```

3.2.4 Confidentiality of Authentication data

We define the following goal:

```
secret_auth:(_){SP,C};
```

to check that the `authnRequest` and the `signedAuthnResponse` are known by Security Provider and Client only. Any time that one of these facts is used we label it with `secret_auth:()` in order to check confidentiality of that. The `Client` entity sends the following message to the `SP` entity:

```
Actor -Ch_C2SP_2-> AnySP : httpRequest (post, AnySP,
        nil_http_element, secret_auth:(ARsp));
```

Then the `SP` entity receives it:

```
C  -Ch_C2SP_2-> Actor : httpRequest (post, Actor,
        nil_http_element,secret_auth:(postBinding
        (signedAuthnResponse(inv(pk(IdP)),
        Actor, IdP, C, ID), URI)));
```

3.2.5 Integrity

The goal "Service Provider Authentication" implies integrity of Resource. This is due to the fact that in order to check this authentication we check if the channel is authenticated. As written in [1] if a message `M` is sent over an authenticated channel `*->`, then the integrity of `M` is guaranteed.

3.3 OpenID

In this section we describe the formalization of the security goals of OpenID, starting from the informal description given in Section 2.3. The LTL goal is in another file because the translator cannot translate it in attack state and SATMC can handle just one LTL formula. The validation of these goals has returned an `INCONCLUSIVE` response with SATMC because after a few hours, SATMC was still running. With Cl-Atse, we got a `NO_ATTACK_FOUND` result after a few seconds.

3.3.1 Mutual authentication

In OpenID, the authentication is a mutual authentication between the client C and a relying party RP.

The client sends the URI to the RP over an authentic and secure channel. The same type of channel is used by the RP to send the resource to C. Therefore C must be able to authenticate RP on the resource and the RP authenticates C on the URI sent in the first message.

We need to create a channel goal for each of these authentications:

```
goals
  RP_on_C :(_) RP *-> C ;
  C_on_RP:(_) C *->  RP;
```

When the client sends the URI to RP in the first exchanged message, we use the C_on_RP goal to check that the URI comes from the client:

```
  Actor *->* RP: httpRequest (post, C_on_RP:(URI),
    nil_http_element, userSupplID(Actor));
```

And when the client receives the resource at the end, we use the RP_on_C goal to check that the resource comes from the relying party:

```
  RP_on_C:(Resource) := Resource;
```

We use the C_on_RP goal to check that the URI received by the RP comes from the client:

```
  C_on_RP:(URI) := URI;
```

When the RP sends the resource to the client, we use the RP_on_C goal to check that the resource comes from the RP.

```
  Actor *->* AnyC: httpResponse(code_200, nil_agent,
     nil_http_element, RP_on_C:(Resource));
```

3.3.2 Confidentiality

Resource

In OpenID, the resource is known only by the relying party, because RP is the owner of the resource, and by an authenticated and authorized client.

Therefore, we can see the resource as a secret between the client and the relying party.

In order the check that the resource is only known by RP and C, we can define a secrecy goal “secret_data” between RP and C:

```
goals
  secret_Data :(_) { RP, C };
```

Then, when the resource is exchanged, we use this secrecy goal. The resource is sent from the RP:

```
  AnyRP *->*   Actor: httpResponse(code_200, nil_agent,
     nil_http_element, secret_Data:(?Resource));
```

and it is received by the client:

```
  secret_Data:(Resource) := fresh();
```

Requests and assertions

OpenID protocol is based on authentication requests and authentication responses (or assertions) to know whether the resource can be sent to the client or not. These messages contain critical data and they have to be known only by the client, because he is between the relying party and the provider, by the relying party because he will generate the authentication request and check the authentication response, and by the provider because he will receive the authentication request and generate the corresponding response. Therefore, these messages have to be known only by these three entities.

In order to check this, we create a secrecy goal “secret_ReqAndAssert” and we use it each time a request or response is sent or received by a participant.

In the models, only secure channels are used by the three participants, therefore this goal is “trivial”. But it should be defined as an explicit protocol goal.

Definition of the goal:

```
goals
  secret_ReqAndAssert :(_) { RP, C, OP };
```

For example, when C receives the authentication request from the RP, we check the secrecy goal:

```
RP *->*    Actor: httpResponse(code_30x, ?OP,
secret_ReqAndAssert:(?OIDAuthnRequest),
nil_http_element);
```

We check that the OpenID provider receives the authentication request from C:

```
on(C  *->* Actor: httpRequest(get, Actor,
secret_ReqAndAssert:(oidAuthnRequest(
C, Actor, Handle, RP)), nil_http_element)): {
...
}
```

And finally we check that the relying party generates the authentication request:

```
Actor *->*   AnyC: httpResponse(code_30x, OP,
secret_ReqAndAssert:(oidAuthnRequest(
C, OP, Handle, Actor)), nil_http_element);
```

3.3.3 Authorization

When a client accesses to the resource, the relying party generates and sends to the client a local ID (it can be viewed as a session ID).
Now the client can access the resource directly (without dealing with authentication requests and responses) using this local ID.
Therefore this local ID is a critical data and each time a client uses an ID, RP has to be sure that:

- RP has generated this ID;
- this ID is only for the client C with provider P;
- RP already has a session for C and his provider P with ID.

In order to check that, we defined some predicates and an LTL formula:

- *access* (*client*, *resource*, *rp*): A client accesses the resource on relying party *rp*.
- *haveSession* (*client*, *rp*, *p*, *resource*): A client with provider *p* has a session on relying party *rp* for the resource.

- *isProvider* (*p*, *client*): p is the provider of client
- *caused* (*assertion*, *localID*, *resource*): localID is caused by assertion for a specified resource.

$$\forall c, r, rp\ \Box(access(c, r, rp) => \Diamond^{-1}(\exists p, a, lid(isProvider(p, c) \wedge caused(a, lid, r) \wedge haveSession(c, rp, p, lid))))$$

But, in ASLan++, the existential quantifier ∃ cannot be used. We need to express it with the universal quantifier ∀ and change the formula. This is the adapted formula in ASLan++:

```
goals
  unauthorizedAccess:
    forall C R RP. [](C->access(R, RP) =>
      [-] (!forall P A LID.(!P->isProvider(C)  |
                            !caused(A, LID, R) |
                            !C->haveSession(RP, P, LID)))
    );
```

3.4 OAuth 2.0

We formalize the security goals of OAuth pertinent to the ACF flow, for which a formal model has be developed in SPaCIoS. From the security goals described in section 2.4 “resource owner’s authentication” and “client authentication in IF” are not applicable as they refer to the IF flow of the OAuth protocol. We therefore focus on the “client authentication in ACF” and “Authorization” goals here.

The verification results for these goals have been negative: the SATMC model checker did not terminate on these goals. We expect this is due to the complexity of the ACF flow model, not the complexity of the goals. We are currently working on finding a suitable level of abstraction for this model.

In the following, we consider the “facebook” model which represents the ACF flow of the OAuth protocol. The model is given in Appendix B.

3.4.1 Client Authentication

For “client authentication in ACF” we add the standard predicates “witness” and “running” to the specification of Facebook and Client respectively. That is, in the Facebook code, right before `Token:=fresh();`, we add `witness(Actor,App_ClientID, Code,App_Secr)`. Similarly, in the Client code we add `running(FB,APP_ClientID,Code,App_Secr)` right before the line %Access Token.

Then, the client authentication in ACF can be formalized as:

```
forall X Y Z W.[] ((witness(X, Y, Z, W)) =>
               <-> (running(X, Y, Z, W)));
```

This corresponds to non-injective agreement in Lowe's characterization of authentication. If injective agreement is needed, then either the server should check for uniqueness of "Code" during authentication, or nonce-based challenge response can be used.

3.4.2 Authorization

For the "Authorization" goal, we add three predicates to the model:

- In Client: add `has_token(Actor,APP_ClientID,Token)` right before the line %Accessing protected resources.
- in User: add `agreed(Actor,FB,APP_ClientID)` right after `Actor *->* FB: httpRequest(get,uri(FB,RED_URI),AzReq,`... Here `APP_ClientID` should be extracted from azReq.
- In Facebook: add `issued(Actor,APP_ClientID,Token)` right after `AccessTokens->add`....

Now, the Authorization property can be expressed as:

```
forall X Y Z.[]((has_token(X, Y, Z)) => exists A U.<->
(issued(A, Y, Z) \& <-> (agreed(U, A, Y))))
```

3.5 Pervasive Retail

In this section, we describe the formalization of the security goals of Pervasive Retail, starting from the informal description given in Section 2.5.

For offer.aslan++, the result with SATMC is not conclusive because it was still running after few minutes but with Cl-Atse, we got a `NO_ATTACK_FOUND` result after less than one second.

For logon.aslan++, Cl-Atse result is `INCONCLUSIVE` because properties and clauses have been ignored and there is a failure while reading aslan file (*Failure while reading : unknown signature for operator*). SATMC result is `INCONCLUSIVE`.

3.5.1 Mutual authentication

The mutual authentication goal refers to the authentication between Pervasive Retail Platform and various clients (Consumer Client, Retailer Client, and Product Provider Client). In the specification offer.aslan++, the entity "Client" is the Consumer Client, all the other entities are components of the Pervasive Retail Platform. The entity "Orchestrator" is the component interacting directly with the Consumer Client.

All messages between the Client and the Orchestrator are exchanged over secure channels.

In order to check that, we defined two channel goals, one to authenticate the Client by the Orchestrator and another to authenticate the Orchestrator by the Client:

```
goals
  Orch_on_C:(_) Orchestrator *-> Client;
  C_on_Orch:(_) Client *-> Orchestrator;
```

At the beginning, the client tries to create a session with the orchestrator sending an application_key. Orchestrator has to be sure that the client sent this session request:

```
Actor *->* Orchestrator :
create_session(C_on_Orch:(Application_Key));
```

Then, when the Orchestrator sends the offer to the client, the latter has to be sure that the Orchestrator sent this offer:

```
Orchestrator *->* Actor :
Orch_on_C:(secret_Offer:(?Deal.?Price)) ;
```

We do the same verification for the Orchestrator:

```
  Client *->* Actor : create_session(?Application_Key);
  C_on_Orch:(Application_Key) := Application_Key;
  ...
  Actor *->* Client :
  Orch_on_C:(secret_Offer:(Deal.Price)) ;
}
```

3.5.2 Secrecy

In this scenario, the offer sent by the Orchestrator is considered as critical data and it has to be kept secret by the Client and the Orchestrator.

In order to check that, we defined a secrecy goal "secret_Offer" between the Client and the Orchestrator:

```
goals
  secret_Offer:(_) {Orchestrator, Client};
```

Then, when the Client receives the offer, we check the secrecy:

```
Orchestrator *->* Actor :
Orch_on_C:(secret_Offer:(?Deal.?Price));
```

We do the same when the Orchestrator sends the offer to the Client:

```
Actor *->* Client :
Orch_on_C:(secret_Offer:(Deal.Price));
```

3.5.3 Non repudiation

In the offer scene, the retailer shall not deny the offer made for the consumer. It means the offer is signed by the Orchestrator and we only need to check that the offer comes from the Orchestrator and it has not been altered during the transmission. This goal is already defined using a channel goal in the mutual authentication.

In the coupon scene, if the ProviderClient provided a coupon, when the RetailerClient checks the coupon, the result has to be 1. This can be checked using a LTL formula and the following two predicates :

```
% ProviderClient provides a coupon
provide(agent, type_Of_Coupon)            : fact;

% RetailerClient check a coupon
getResult(agent, type_Of_Coupon, nat)     : fact;
```

The following formula checks whether the coupon has been provided before:

```
goals
  nonrepudCoupon:
    forall RC R PC C. [](
      (RC->getResult(C, R) & (R = 1)) =>
      <-> (PC->provide(C)));
```

3.5.4 Authorization

In Pervasive Retail, a role based access control mechanism is used to guarantee that the information about purchasing decision and coupon is accessible for product provider, but the price is only accessible for retailer.

In order to check this goal, we can use the following predicates and LTL formulae.

```
% An entity access to the decision
accessDecision(role) : fact;

% An entity access to the price
accessPrice(role) : fact;
```

Then, each time an entity accesses an information, we check its role:

```
goals
  authorizedAccessDecision:
    forall R. [](accessDecision(R) =>
    (R = retailer | R = provider));
  authorizedAccessPrice:
    forall R. [](accessPrice(R) => (R = retailer));
```

3.5.5 Privacy

In the coupon scene, private information of client should never be sent out. We can represent this information by a symbol *location* representing the location of the consumer. Then, we can add a secrecy goal with the client only.

```
secret_Privacy:(_) { Client } ;
```

And at the end of client entity, we check it:

```
secret_Privacy:(Location) := Location ;
```

3.5.6 Need to know

In the coupon scene, personal information which identify a person (name, gender, address) in the consumer profile should not be known by the Provider-Client. We can divide the consumer profile into two parts, Personally Identifiable Information (PII), and non Personally Identifiable Information (nonPII). We can express this goal as a secret between Consumer and Orchestrator on PII.

```
secret_NeedToKnow:(_) {Orchestrator, Consumer};
```

3.6 Infobase

In this section, we describe the formalization of the security goals of Infobase, starting from the informal description given in Section 2.7. All these goals have been translated into attack states instead of LTL goals because SATMC can handle more than one attack states goals but just one LTL. The validation of these goals has returned an INCONCLUSIVE response that means that with the default depth of 80 steps no attack has been found and even if the depth option is increased (tested up to 2000) SATMC still returns INCONCLUSIVE. We have also validated the specification with CL-Atse that, as expected, has returned NO_ATTACK_FOUND. The model is given in Appendix D.

3.6.1 Confidentiality of authentication data (passwords)

For the log in phase, the first message is sent from the Client entity to the Repository:

```
%user sends user and pass to the LoginService
Actor ->* Repository:
Actor.UserName.secret_Password:(Password);
```

This message is sent using a confidential channel and includes: the name of the User (Actor), the UserName and a secret Password. It is received by the Repository entity but the check is not done in the receiving line but right after:

```
 on(? ->* Actor: ?UserIP.?UserName.?Password &
 loginDB->contains((?UserName,?Password,?Role))): {
  Password := secret_Password:(Password);
```

The last statement is reached only in the case the UserName and the Password matches with an entry in the login database. This means that UserName is a legitimate user of the system. For this reason, we want to be sure that his password is secret. The UserIP variable in the received message should contain the value of type agent corresponding to the IP address of the user, indeed, the Repository is using it for the following message exchanges with the user. Even in the case the intruder sends correct UserIP.UserName.Password to the Repository, the secrecy goal is violated because of the activation of the secrecy_Password goal at the Client side. With secret_Password we thus define the secrecy goal that aims to check if the Password is known by Client (i.e. the user of the system) and Repository only.

```
secret_Password:(_) {UserIP, repository};
```

where repository is the agent name assigned to the Repository entity.

3.6.2 Confidentiality of content stored into the repository

Stored data is not part of the actual model yet. Fortunately, this goal is subsumed by the goal checking if the response that the Repository entity sends to the Client is confidential. The goal checks that an intruder who knows response “R”, created by the repository, must be a legitimate user of the system and must possess the permissions to issue the request that causes the `Repository` to create response “R”. In order to do this we have added the following goal:

```
 authorization_dishonest: forall Req.
 [](iknows(answerOn(Req)) =>
    checkPermissions_dishonest(Req));
```

that subsumes the in-line `secret_Answer(_)` goal we could have used. This goal is indeed more restrictive and allows us to check also for authorization in the case of dishonest users (i.e. an intruder which is also a legitimate user of the system that can issue certain requests to the system). The `checkPermissions_dishonest(Req)` predicate is asserted by means of a Horn Clauses:

```
permission(Us,Pwd,Ro,Req):
 checkPermissions(Us,Req) :-
  loginDB->contains((Us,Pwd,Ro)) &
  Ro->can_exec(Req);

permission_dishonest(Us,Req):
 checkPermissions_dishonest(Req) :-
  dishonest_usr(Us) &
  checkPermissions(Us,Req);
```

Thanks to those two Clauses, every time a dishonest user, `dishonest_usr(Us)`, (i.e. an intruder) has permissions to issue some requests `checkPermissions(Us,Req);`, the predicate `checkPermissions_dishonest(Req)` holds. This allowed us to avoid existential quantification in the goal above which would have required extra computation time from the model-checkers.

3.6.3 Confidentiality of cookies

This goal is modeled like the one for the passwords described in Section 3.6.1, using:

```
 secret_Cookie:(_) {UserIP, repository};
```

In the `Client` entity the secrecy goal is activated as soon as the cookie is received by the user:

```
select { on (repository *->* Actor:
 secret_Cookie:(?Cookie) &
 ?Cookie=cookie(UserName,?,?)): {} }
```

and in the `Repository` entity the goal is activated in conjunction with the instantiation of the cookie variable:

```
Nonce := fresh();
Cookie := secret_Cookie:(cookie(UserName,Role,Nonce));
```

3.6.4 Confidentiality of responses from the Repository

This is exactly the `authorization_dishonest` goal described in Section 3.6.2.

3.6.5 Authorization of requests

This goal is composed by two subgoals: `authorization_dishonest` and `authorization_honest`. The former has been already described in Section 3.6.2, the latter checks that only users having appropriate permissions can issue requests to the `Repository`. This goal is modeled as follow: a user who receives the response corresponding to a request must also have the necessary permission to issue that request:

```
authorization_honest: forall Us Req.
 [](Us->got(answerOn(Req)) => checkPermissions(Us,Req));
```

The `Us->got(answerOn(Req))` predicate, a syntactic sugar for `got(Us,answerOn(Req))`, is asserted in the `Client` entity just after the user receives the message `answerOn(Req)`, and `checkPermissions(Us,Req)` is introduced by the Horn Clause `permission(Us,Pwd,Ro,Req)` (cf. Section 3.6.2).

3.6.6 Integrity of requests

This goal is modeled with the `request_integrity` in-line goal:

```
request_integrity:(_) UserIP *-> repository;
```

The goal is activated in the `Client` entity in conjunction with the shipping of the cookie and the request to the `Repository`:

```
Actor *->* repository: Cookie.request_integrity:(Request);
```

In the `Repository` entity the goal is activated in the sending statement where the response is sent to the `Client`:

```
Actor *->* UserIP: request_integrity:(Request).Answer;
```

4 Discussions

In this section, we first discuss how we transform the specification of a security goal when the goal refers to the events that are not observable in the system under test. Then, we briefly explain the issue of testability in asynchronous settings and also explain why this is not a concern in the context of SPaCIoS.

4.1 Unobservable events

Let us start with a simple example that demonstrates the problem of unobservable events in security goals. Take the following definition of the authentication property which is based on Lowe's notion of *non-injective agreement* [7].

$$\mathbf{G}\,\forall(\mathtt{state}_{\mathtt{sp}}(7,\mathtt{SP},[\mathtt{C},\ldots,\mathtt{URI},\ldots]) \Rightarrow \exists\,\mathbf{O}\,\mathtt{state}_{\mathtt{c}}(2,\mathtt{C},[\mathtt{SP},\ldots,\mathtt{URI},\ldots])) \quad (1)$$

stating that, if SP reaches the last step 7 believing to talk with C, who requested URI, then sometime in the past C must have been in the state 2, in which he requested URI to SP. Here, SP and C refer to the participants of the SAML SSO protocol, examined in Sections 2.2 and 3.2 while $\mathtt{state}_r$ is an event of the form $\mathtt{state}_r(j,a,[e_1,\ldots,e_p])$ meaning that a, playing role r, is ready to execute the protocol step j, and $[e_1,\ldots,e_p]$, for $p \geq 0$ is a list of expressions representing the internal state of a. (See Table 2).

Since we aim at testing implementations using attack traces as test cases with the purpose of detecting a violation of the authentication property, we need to be sure that at the end of the execution of the attack trace, the property has been really violated. Thus, we need to take into account the testing scenario in terms of the observability of channels and of the internal states of each principal. This can be done by defining a set of observable facts. For instance, in case the tester can observe the messages passing through a channel c then, for all rs, b, a, and m, the $\mathtt{sent}(rs,b,a,m,\mathtt{c})$ facts are observable. Similarly, in case the tester can observe the internal state of an agent a, then for all r, j, e_1, ..., e_n the $\mathtt{state}_r(j,\mathtt{a},[e_1,\ldots,e_n])$ facts are observable.

Once defined the set of observable events according to the testing scenario, we rewrite the formula using them. For instance, in the example above let us suppose that the internal state of sp is not observable, while the channel $\mathtt{c}_{\mathtt{SP2C}}$ that connects SP to C is observable. Then, we rewrite the property (1) as follows:

Fact	Meaning
$\mathtt{state}_r(j, a, [e_1, \ldots, e_p])$	The agent a, playing role r, is ready to execute the protocol step j, and $[e_1, \ldots, e_p]$, for $p \geq 0$ is a list of expressions representing the internal state of a.
$\mathtt{sent}(rs, b, a, m, c)$	The agent rs, pretending to be b, sent the message m to a on channel c.

Table 2: Facts and their informal meaning

$$\mathbf{G}\,\forall(\mathtt{sent}(\mathtt{SP}, \mathtt{SP}, \mathtt{C}, \mathtt{res}(\mathtt{URI}), \mathtt{c}_{\mathtt{SP2C}}) \Rightarrow \exists\,\mathbf{O}\,\mathtt{state}_{\mathtt{c}}(2, \mathtt{C}, [\mathtt{SP}, \ldots, \mathtt{URI}, \ldots])) \quad (2)$$

where $\mathtt{sent}(\mathtt{SP}, \mathtt{SP}, \mathtt{C}, \mathtt{res}(\mathtt{URI}), \mathtt{c}_{\mathtt{SP2C}})$ stands for $\mathtt{SP}$ pretending to be $\mathtt{SP}$ sends to $\mathtt{C}$ a message $\mathtt{res}(\mathtt{URI})$ over the channel $\mathtt{c}_{\mathtt{SP2C}}$. Here, $\mathtt{res}(\mathtt{URI})$ represents the "resource" returned by $\mathtt{SP}$ when $\mathtt{SP}$ reaches the step 7 (see Table 2). When the model does not satisfy the expected security property, a counterexample (i.e. an *attack trace*) is generated and returned by the model checker. The attack trace can be checked on the implementation; its executability, which shows a violation of the authentication property (see Equation 2) is observable by the tester. To summarize, in case a security goal refers to unobservable events, the modeler must identify events in the specification that refer to observable events, and then rewrite the property using observable events. This transformation cannot in general be automated. However, we expect that for a number of popular security goals one can automatically rewrite the goals using observable events.

It is worth mentioning that the aforementioned solution is not general: it is possible that the model checker generates (attack) traces containing unobservable events, even when the goal is expressed using observable events only. The aforementioned solution is a first-approximation: in our problem cases we have successfully used this solution.

4.2 Testability in asynchronous settings

In synchronous settings, the tester observes the events in the same order as they occur in the system under test (SUT). This need not be the case in asynchronous settings; e.g., when the tester interacts with the SUT via asynchronous channels. Asynchronous observations in general limit the "knowledge" of the tester about the behaviors of the SUT. In the extreme cases, properties that can be tested in synchronous settings may become untestable in asynchronous ones. There are two situations where the tester and the SUT are connected asynchronously.

- The SUT has multiple remote input/output ports, and the tester consists of an observation point on each port. The observation points must communicate in order to reach a consensus on the global behaviors of the SUT; the consensus can be used to issue a pass/fail verdict for the test case at hand. This problem has been studied in the literature; e.g., see [6, 5].
- The tester observes the SUT via an asynchronous "lens", e.g. an asynchronous channel. Then, if the tester observes event a followed by event b, it cannot be sure whether in fact a has happened in the SUT before b, or perhaps b has happened before a and the channel has reordered the events. This problem has also been studied in the literature; e.g., see [4, 8].

In none of our problem cases we have identified SUTs with multiple remote input/output ports, and also the communication between the SUT and the tester can in our problem cases be assumed to be synchronous. Therefore, the problem of testability in asynchronous settings is not a concern in the context of SPaCIoS.

5 Conclusion

In this deliverable, we have reported on our experiences with using ASLan++ to formalize the security goals of the SPaCIoS problem cases, and using the AVANTSSAR platform for checking the goals.

We observe that ASLan++ is sufficiently expressive for the goals of the SPaCIoS problem cases; however, in a number of cases SATMC and the other AVANTSSAR verification back-ends failed to verify the goals. This can be due to either the complexity of the goal or the complexity of the model (or both). It turns out that in a number of cases we can rewrite the goals into an "equivalent" simpler goal that is amenable to automated verification. We are currently looking at different levels of abstraction for simplifying the models. Obviously, the verification back-ends also need to be strengthened and extended.

In the future, we intend to work on the following three issues: (1) finding patterns for simplifying security goals, (2) finding suitable levels of abstraction for models, so that the models would be amenable to automated verification and would also be sufficiently detailed for the purpose of test case generation, and (3) extending the verification back-ends of AVANTSSAR, in particular SATMC, in order to handle the problem cases of SPaCIoS.

References

[1] Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures. http://www.avantssar.eu/.

[2] AVANTSSAR. Deliverable 2.3 (update): ASLan++ specification and tutorial, 2011. Available at http://www.avantssar.eu.

[3] AVISPA. AVISPA v1.1 User Manual. http://www.avispa-project.org, 2006.

[4] Karthikeyan Bhargavan, Satish Chandra, Peter J. McCann, and Carl A. Gunter. What packets may come: automata for network monitoring. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 206–219, New York, NY, USA, 2001. ACM.

[5] Robert M. Hierons. Controllable testing from nondeterministic finite state machines with multiple ports. *IEEE Trans. Computers*, 60(12):1818–1822, 2011.

[6] Robert M. Hierons and Hasan Ural. Overcoming controllability problems with fewest channels between testers. *Computer Networks*, 53(5):680–690, 2009.

[7] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW'97)*, pages 31–43. IEEE Computer Society Press, 1997.

[8] Neda Noroozi, Ramtin Khosravi, Mohammad Reza Mousavi, and Tim A. C. Willemse. Synchronizing asynchronous conformance testing. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *SEFM*, volume 7041 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2011.

[9] SPaCIoS. Deliverable 2.1.1: Analysis of the relevant concepts used in the case studies: applicable security concepts, security goals and attack behaviours, 2011.

[10] SPaCIoS. Deliverable 5.1: Proof of Concept and Tool Assessment v.1, 2011.

[11] SPaCIoS. Deliverable 2.1.2: Modeling security-relevant aspects in the IoS, 2012.

A LTL Definitions (recap)

Operator	ASLan++ connective	Explanation
$\neg$	`!f`	negation
$=$	`f_1 = f_2`	equality
$\neq$	`f_1 != f_2`	inequality
$\wedge$	`f_1 & f_2`	conjunction
$\vee$	`f_1 | f_2`	disjunction
$\Rightarrow$	`f_1 => f_2`	implication
$\forall$	`forall V_1 ... V_n.f`	universal quantification
$\exists$	`exists V_1 ... V_n.f`	existential quantification
neXt	`X(f)`	in the next state[1]
Yesterday	`Y(f)`	in the previous state[1]
Finally	`<>(f)`	at some time in the future[2]
Once	`<->(f)`	at some time in the past
Globally	`[](f)`	always
Historically	`[-](f)`	at all times in the past
Until	`U(f_1,f_2)`	f_1 holds until f_2 holds and f_2 will eventually hold
Release	`R(f_1,f_2)`	f_2 holds until and including the point where f_1 first becomes true; if f_1 never becomes true f_2 must remain true forever.
Since	`S(f_1,f_2)`	f_2 was true at least once in the past and since then f_1 holds

Table 3: LTL operators for specifying goals

Operator	Syntax	π satisfies formula φ at time i ($\pi, i \models \varphi$)
Globally	$\Box a$	$(\forall j \geq i)\pi, j \models a$
Eventually	$\Diamond a$	$(\exists j \geq i)\pi, j \models a$
Until	$a\mathbf{U}b$	$(\exists j \geq i)\pi, j \models b$ and $(\forall k \in \{i, \ldots, j-1\})\pi, k \models a$
Next time	$\bigcirc a$	$\pi, i+1 \models a$
Globally in the past	$\Box^{-1} a$	$(\forall j \in \{0, \ldots, i\})\pi, j \models a$
Eventually in the past	$\Diamond^{-1} a$	$(\exists j \in \{0, \ldots, i\})\pi, j \models a$
Since	$a\mathbf{S}b$	$(\exists j \in \{0, \ldots, i\})\pi, j \models b$ and $(\forall k \in \{j+1, \ldots, i\})\pi, k \models a$
Last time	$\bigcirc^{-1} a$	$i > 0$ and $\pi, i-1 \models a$

Table 4: Non-strict LTL syntax and semantics

B OAuth 2.0 Formal Model

```
specification OAuth
channel_model ACM


entity Environment {
 types
 int < message;

 %% Common
 uri < message;
 uri_path < message;
 method < message;
 code < message;

 %% HTTP protocol
 http_element < message;
 data < message;

 %% OAuth2.0
 client_id < message;
 client_secret < message;
 scope < message;
 state < message;
 authz_code < message;
 access_token < message;
 oauth_authz_message < http_element;
 oauth_tok_message < http_element;
 oauth_token < http_element;

 symbols
 %% NIL values
 nil : message;
 nil_agent : agent;
 nil_uri : uri;
 nil_http_element : http_element;
 %% HTTP protocol
 get, post : method;
 code_30x, code_200 : code;


 %% Agents
 app, u1, fb : agent;

 app_clientid, badapp_clientid : client_id;
 nonpublic app_secr : client_secret;
 badapp_secr : client_secret;
 app_uri_path, badapp_uri_path : uri_path;
 app_red_uri_path, badapp_red_uri_path :
uri_path;
 app_scope, badapp_scope : scope;

 u1_userdata_path : uri_path;
 nonpublic u1_userdata : data;

 fb_az_ep, fb_tok_ep : uri_path;

 running(agent, client_id, authz_code, client_secret):
fact;
 witness(agent, client_id, authz_code, client_secret):
fact;

 %% Channels
 CChannels : agent.channel.channel set;
 ch_u2app_1s1, ch_app2u_1s1,
 %ch_u2app_2s1, ch_app2u_2s1,
 %ch_u2fb_s1, ch_fb2u_s1,
 ch_app2fb_s1, ch_fb2app_s1,
 ch_u2i_1s2, ch_i2u_1s2,
 %ch_u2i_2s2, ch_i2u_2s2,
 %ch_u2fb_s2, ch_fb2u_s2,
 ch_i2fb_s2, ch_fb2i_s2,
 ch_u2fb, ch_fb2u : channel;

 %% Intruder
 fake_ac : authz_code;
 fake_at : oauth_token;

 %%%% Symbols for messages

 %% Common
 uri(agent, uri_path) : uri;
 resource(data) : http_element;

 %% HTTP protocol
 httpRequest(method, uri, http_element, http_element)
: message;
 httpResponse(code, uri, http_element, http_element)
: message;

 %% OAuth 2.0 protocol

 %authzRequest(client_id, redirect_uri, scope,
state)
 %authzResponse(authz_code, state)
 %accessTokenRequest(client_id, client_secret,
authz_code, redirect_uri)
```

```
 %accessTokenResponse(access_token, expires_in,
refresh_token, scope)
 %accessToken(int)
 authzRequest(client_id, uri, scope, state)
: oauth_authz_message;
 authzResponse(authz_code, state) : oauth_authz_message;
 accessTokenRequest(client_id, client_secret,
authz_code, uri) : oauth_tok_message;
 accessTokenResponse(access_token) : oauth_tok_message;
 accessToken(access_token) : oauth_token;


 %%%% Variables
 RegisteredApp: client_id.uri.client_secret
set;
 SocialGraph : agent.uri.data set;
 AuthzCodes : client_id.agent.authz_code
set;
 AccessTokens : client_id.agent.access_token
set;

 link(channel, channel) : fact;

 %%%%%%%%%%%%%%%% SESSION %%%%%%%%%%%%%%%%
 entity Session (
 App, U, FB : agent,
 App_URI_Path : uri_path,
 App_Red_URI_Path : uri_path,
 App_ClientID : client_id,
 App_Secr : client_secret,
 UserData_URI : uri_path,
 Scope : scope,
 RegisteredApp : client_id.uri.client_secret
set,
 SocialGraph : agent.uri.data set,
 AuthzCodes : client_id.agent.authz_code
set,
 AccessTokens : client_id.agent.access_token
set,
 Ch_U2App_1, Ch_App2U_1,
 Ch_U2App_2, Ch_App2U_2,
 Ch_U2FB, Ch_FB2U,
 Ch_App2FB, Ch_FB2App: channel,
 Channels : agent.channel.channel set
 ) {

 %%%%%%%%%%%%%%%% OAUTH CLIENT %%%%%%%%%%%%%%%%
 entity Application(
 Actor, U, FB : agent,
 App_URI_Path : uri_path,
 App_Red_URI_Path : uri_path,
 App_ClientID : client_id,
 App_Secr : client_secret,
 UserData_URI : uri_path,
 Scope : scope,
 Ch_U2App_1, Ch_App2U_1,
 Ch_U2App_2, Ch_App2U_2,
 Ch_App2FB, Ch_FB2App: channel
 ) {
 symbols
 State : state;
 Code : authz_code;
 Token : access_token;
 UserData : data;
 RenderUserDataInPage : data;

 body {
 ?U -Ch_U2App_1-> Actor : httpRequest(get,
uri(Actor, App_URI_Path), nil_http_element,
nil_http_element);

 %% --- BEGIN OF OAUTH2.0 --- %%
 State := fresh();
 %% Authorization request
 Actor -Ch_App2U_1-> U : httpResponse(code_30x,
uri(FB, fb_az_ep), authzRequest(App_ClientID,
uri(Actor, App_Red_URI_Path), Scope, State), nil_http_element);

 %% Authorization response
 ?U -Ch_U2App_2-> Actor : httpRequest(get,
uri(Actor, App_Red_URI_Path), authzResponse(?Code,
State), nil_http_element);
 running(FB,App_ClientID,Code,App_Secr);
 %% Access Token Request
 % Access Token:
 % facebook doesn't clarify whether this
%request is done in POST or GET. Current OAuth2
%draft says POST.
 Actor -Ch_App2FB-> FB : httpRequest(post,
uri(FB, fb_tok_ep), nil_http_element, accessTokenRequest(App_ClientID,
App_Secr, Code, uri(Actor, App_Red_URI_Path)));
 %% Access Token Response
 FB -Ch_FB2App-> Actor : httpResponse(code_200,
nil_uri, accessTokenResponse(?Token), nil_http_element);


 % Accessing protected resource
 Actor -Ch_App2FB-> FB : httpRequest(get,
uri(FB, UserData_URI), accessToken(Token),
nil_http_element);
 FB -Ch_FB2App-> Actor : httpResponse(code_200,
nil_uri, nil_http_element, resource(?UserData));


 %% --- END OF OAUTH2.0 --- %%
 RenderUserDataInPage := fresh();
 Actor -Ch_App2U_2-> U : httpResponse(code_200,
nil_uri, nil_http_element, resource(RenderUserDataInPage));

 %% Hack to have the request in the last
step
 App_URI_Path := App_authn_User_on_uri:(App_URI_Path);

 }
 }

 %%%%%%%%%%%%%%%% OAUTH USER-AGENT %%%%%%%%%%%%%%%%
 entity User(
 Actor, App, FB : agent,
 App_URI_Path : uri_path,
 Ch_U2App_1, Ch_App2U_1,
 Ch_U2FB, Ch_FB2U : channel,
 Channels : agent.channel.channel set
 ) {
```

```
 symbols
 Red_URI : uri_path;
 AzReq : authzRequest(message, uri(agent,
uri_path), scope, state);
 AzResp : authzResponse(authz_code, state);
 RenderUserDataInPage : data;
 Code : authz_code;
 State : state;
 Ch_U2App_2, Ch_App2U_2 : channel;

 body {
 %% App_authn_User_on_uri:(uri(App, App_URI_Path))
 Actor -Ch_U2App_1-> App : httpRequest(get,
uri(App, App_authn_User_on_uri:(App_URI_Path)),
nil_http_element, nil_http_element);

 %% --- BEGIN OF OAUTH2.0 --- %%
 %% Authorization Request (1): redirection
 %% to Facebook
 App -Ch_App2U_1-> Actor : httpResponse(code_30x,
uri(?FB, ?Red_URI), ?AzReq, nil_http_element);


 %% User authentication and granting access
 % - how to deal with that?!?!?!

 %% Authorization Request (2): redirected
 %% to FB
 Actor -Ch_U2FB-> FB : httpRequest(get, uri(FB,
Red_URI), AzReq, nil_http_element);
 %% Authorization Response(1): redirection
 %% to the client
 select {
 on(FB -Ch_FB2U-> Actor : httpResponse(code_30x,
uri(?App, ?Red_URI), ?AzResp, nil_http_element)
&
 Channels->contains((?App, ?Ch_U2App_2, ?Ch_App2U_2))):
{

 %% Authorization Response(2): redirected
 %% to the client
 Actor -Ch_U2App_2-> App : httpRequest(get,
uri(App, Red_URI), AzResp, nil_http_element);
 %% --- END OF OAUTH2.0 --- %%
 App -Ch_App2U_2-> Actor : httpResponse(code_200,
nil_uri, nil_http_element, resource(?RenderUserDataInPage));
 %assert finished: false;
 }
 }
 }
 }


 %%%%%%%%%%%%%%%% Facebook %%%%%%%%%%%%%%%%
 entity Facebook(
 Actor, U, App : agent,
 RegisteredApp : client_id.uri.client_secret
set,
 SocialGraph : agent.uri.data set,
 AuthzCodes : client_id.agent.authz_code
set,
 AccessTokens : client_id.agent.access_token
```

```
set,
 Ch_U2FB, Ch_FB2U,
 Ch_App2FB, Ch_FB2App: channel

 ) {

 symbols
 UserData_URI : uri_path;
 UserData : data;
 App_ClientID : client_id;
 App_Secr : client_secret;
 AnyApp : agent;
 App_Red_URI_Path : uri_path;
 Code : authz_code;
 Token : access_token;
 Scope : scope;
 State : state;

 body {
 select {
 %% Authorization Request: U is redirected
 %% with an Authorization Request
 on(U -Ch_U2FB-> Actor : httpRequest (get,
uri(Actor, fb_az_ep), authzRequest(?App_ClientID,
uri(?App, ?App_Red_URI_Path), ?Scope, ?State), nil_http_element) &
 % Note: user is authenticated (no guard)
 % 2) User is asked to grant/deny App to
 % access his data
 SocialGraph->contains((U, uri(Actor, ?UserData_URI),
?UserData)) &
 RegisteredApp->contains((?App_ClientID,
uri(?App, ?App_Red_URI_Path), ?App_Secr))):
{ %App_Secr not checked here because there's no App authentication
 %(see next step)
 %secret_AC:(Code) := fresh();
 Code := fresh();
 AuthzCodes->add((App_ClientID, U, Code));
 Actor -Ch_FB2U-> U : httpResponse(code_30x,
uri(App, App_Red_URI_Path), authzResponse(Code,
State), nil_http_element);

 select {
 %% Access Token Request: App presents the
 %% Code asking for a Token
 on(?AnyApp -Ch_App2FB-> Actor : httpRequest(post,
uri(Actor, fb_tok_ep), nil_http_element,
accessTokenRequest(?App_ClientID, ?App_Secr, ?Code,
uri(?App, ?App_Red_URI_Path))) &
 % App is authenticated && his redirection
 % URI is verified)
 RegisteredApp->contains((?App_ClientID,
uri(?App, ?App_Red_URI_Path), ?App_Secr))
&
 AuthzCodes->contains((?App_ClientID, ?U,
?Code))): {
 witness(Actor, App_ClientID, Code, App_Secr);
 Token := fresh();
 AccessTokens->add((App_ClientID, U, Token));
 Actor -Ch_FB2App-> AnyApp : httpResponse(code_200,
nil_uri, accessTokenResponse(Token), nil_http_element);
 select {
```

```
 %% The application is accessing to a resource
 %% showing a token
 on (?AnyApp -Ch_App2FB-> Actor : httpRequest(get,
uri(Actor, ?UserData_URI), accessToken(?Token),
nil_http_element) &
 %% Token is checked, UserData_URI has to
 %% be a resource that belongs to the user
 AccessTokens->contains((?App_ClientID, ?U,
?Token)) &
 SocialGraph->contains((?U, uri(Actor, ?UserData_URI),
?UserData))): {
 Actor -Ch_FB2App-> AnyApp : httpResponse(code_200,
nil_uri, nil_http_element, resource(UserData));

 }
 }
 }
 }
 }
 }
 }
 }

 body {


 new Application(App, U, FB, App_URI_Path,
 App_Red_URI_Path, App_ClientID, App_Secr,
 UserData_URI, Scope, Ch_U2App_1, Ch_App2U_1,
 Ch_U2App_2, Ch_App2U_2, Ch_App2FB, Ch_FB2App);
 new User(U, App, FB, App_URI_Path,
 Ch_U2App_1, Ch_App2U_1, Ch_U2FB, Ch_FB2U, Channels);
 new Facebook(FB, U, App, RegisteredApp,
 SocialGraph, AuthzCodes, AccessTokens,
 Ch_U2FB, Ch_FB2U, Ch_App2FB, Ch_FB2App);
 }
 goals
 Client_Authentication_ACF: forall X Y Z
W.[]((witness(X, Y, Z, W)) => <->(running(X,
Y, Z, W)));
 App_authn_User_on_uri:(_) U *-> App;
 %C_on_res:(_) C *-> App;

 }
 body {
 %% channel properties
 %unilateral_conf_auth(ch_u2app_s1, ch_app2u_s1,
 % U, App);
 confidential_to(ch_u2app_1s1, app);
 authentic_on(ch_app2u_1s1, app);
 weakly_confidential(ch_app2u_1s1);
 weakly_authentic(ch_u2app_1s1);
 link(ch_u2app_1s1, ch_app2u_1s1);

 confidential_to(ch_u2i_1s2, i);
 authentic_on(ch_i2u_1s2, i);
 weakly_confidential(ch_i2u_1s2);
 weakly_authentic(ch_u2i_1s2);
 link(ch_u2i_1s2, ch_i2u_1s2);

 %bilateral_conf_auth(ch_u2fb, ch_fb2u, U,
 %FB) that gets authentic to U after her authentication
 %using username/password
 confidential_to(ch_u2fb, fb);
 authentic_on(ch_fb2u, fb);
 confidential_to(ch_fb2u, u1);
 weakly_authentic(ch_u2fb);
 link(ch_u2fb, ch_fb2u);

 %unilateral_auth_conf(Ch_U2App_2, Ch_App2U_2,
 %U, App);
 %confidential_to(ch_u2app_2s1, app);
 %authentic_on(ch_app2u_2s1, app);
 %weakly_confidential(ch_app2u_2s1);
 %weakly_authentic(ch_u2app_2s1);
 %link(ch_u2app_2s1, ch_app2u_2s1);

 %confidential_to(ch_u2i_2s2, i);
 %authentic_on(ch_i2u_2s2, i);
 %weakly_confidential(ch_i2u_2s2);
 %weakly_authentic(ch_u2i_2s2);
 %link(ch_u2i_2s2, ch_i2u_2s2);

 %% FB <-> APPs
 confidential_to(ch_app2fb_s1, fb);
 authentic_on(ch_fb2app_s1, fb);
 weakly_confidential(ch_fb2app_s1);
 weakly_authentic(ch_app2fb_s1);
 link(ch_app2fb_s1, ch_fb2app_s1);

 confidential_to(ch_i2fb_s2, fb);
 authentic_on(ch_fb2i_s2, fb);
 weakly_confidential(ch_fb2i_s2);
 weakly_authentic(ch_i2fb_s2);
 link(ch_i2fb_s2, ch_fb2i_s2);

 RegisteredApp := { (app_clientid, uri(app,
app_red_uri_path), app_secr),(badapp_clientid,
uri(i, badapp_red_uri_path), badapp_secr) };
% application app is registered
%at FB, it provides an OAuth2 end-point at app_red_uri_path
%and it is sharing app_secr with FB
 SocialGraph := { (u1, uri(fb, u1_userdata_path),
u1_userdata) }; % user u, owns userdata available at userdata_uri
 AuthzCodes := {}; % shared memory
 AccessTokens := {}; % shared memory
 CChannels := {(app, ch_u2app_1s1, ch_app2u_1s1),
 (i, ch_u2i_1s2, ch_i2u_1s2)};
 new Session(app, u1, fb, app_uri_path, app_red_uri_path, app_clientid,
 app_secr, u1_userdata_path, app_scope,
 RegisteredApp,  SocialGraph, AuthzCodes,
 AccessTokens, ch_u2app_1s1, ch_app2u_1s1, ch_u2app_1s1, ch_app2u_1s1,
 ch_u2fb, ch_fb2u, ch_app2fb_s1, ch_fb2app_s1, CChannels);
 new Session(i, u1, fb,  badapp_uri_path, badapp_red_uri_path,
 badapp_clientid, badapp_secr, u1_userdata_path, badapp_scope,
 RegisteredApp, SocialGraph, AuthzCodes, AccessTokens, ch_u2i_1s2,
 ch_i2u_1s2, ch_u2i_1s2, ch_i2u_1s2, ch_u2fb, ch_fb2u, ch_i2fb_s2,
 ch_fb2i_s2, CChannels);
 }

}
```

C SAML model

```
specification SAML_SSO_SP_init
channel_model ACM

entity Environment {
  types
    int          < message;
    method       < message;
    code         < message;

        http_element < message;
        resource     < http_element;

        saml_message < message;
    saml_binding < http_element;

  symbols
    %% NIL values
        nil
: message;
        nil_agent
: agent;
        nil_http_element
: http_element;

        %% HTTP protocol values
    get, post
: method;
    code_30x, code_200
: code;
    uri_sp, uri_i
: agent;

        %% Agents
        TrustedSPs
: agent set;
    c, sp, idp
: agent;

        %% Channels
        CChannels
: agent.channel.channel set;
        % Session 1
        ch_c2sp_1s1, ch_sp2c_1s1,
        ch_c2idp_s1, ch_idp2c_s1
: channel;
        % Session 2
        ch_c2i_1s2, ch_i2c_1s2,
        ch_c2idp_s2, ch_idp2c_s2
: channel;

        %% Values for the intruder
        fake_id
: int;

        %% Labels
        httpRequest(method, agent, http_element, http_element)
: message;
    httpResponse(code, agent, http_element, http_element)
: message;
```

```
        htmlForm(agent, saml_binding)
: http_element;
    authnRequest(agent, agent, int)
: saml_message;
        noninvertible
        signedAuthnResponse(private_key, agent, agent, agent, int):
                                                    saml_message;
        httpBinding(saml_message, agent)
: saml_binding;
        postBinding(saml_message, agent)
: saml_binding;

        link(channel, channel)
: fact;


  clauses
    analysis_signedAuthnResponse(K, A1, A2, A3, N):
      iknows(A1.A2.A3.N) :- iknows(signedAuthnResponse(K, A1, A2, A3, N))
                               & iknows(K);

  %%%%%%%%%%%%%%%% SESSION %%%%%%%%%%%%%%%%
  entity Session (C, IdP, SP: agent, TrustedSPs : agent set, URI :
        agent, Ch_C2SP_1, Ch_SP2C_1, Ch_C2SP_2, Ch_SP2C_2,
        Ch_C2IdP, Ch_IdP2C: channel, Channels: agent.channel.channel set) {

    %%%%%%%%%%%%%%%% CLIENT %%%%%%%%%%%%%%%%
    entity Client (Actor, SP, IdP: agent, URI : agent, Ch_C2SP_1,
        Ch_SP2C_1, Ch_C2IdP, Ch_IdP2C: channel,Channels:
        agent.channel.channel set) {
      symbols
                AReq: httpBinding(authnRequest (agent, agent, int), agent);
                ARsp: postBinding(signedAuthnResponse
                      (inv(public_key), agent, agent, agent, int), agent);
        AnySP: agent;
                Resource    : http_element;
                Ch_C2SP_2   : channel;
                Ch_SP2C_2   : channel;

      body {
        %% C-SP (1)
                %URI := C_on_uri:(URI);
                Actor -Ch_C2SP_1-> SP: httpRequest(get,
                                SP_authn_C_on_uri:(URI),
                                nil_http_element, nil_http_element);
        SP    -Ch_SP2C_1-> Actor : httpResponse(code_30x, IdP, ?AReq,
                                nil_http_element);


                %% C-IDP
        Actor -Ch_C2IdP->  IdP: httpRequest (get, IdP, AReq,
                                nil_http_element);
        select {
                  on(IdP   -Ch_IdP2C->  Actor : httpResponse(code_200 ,
                        nil_agent,nil_http_element,
                        htmlForm(?AnySP, ?ARsp)) &
                  Channels->contains((?AnySP, ?Ch_C2SP_2, ?Ch_SP2C_2))): {
        %% C-SP (2)
        Actor -Ch_C2SP_2-> AnySP : httpRequest (post, AnySP,
                nil_http_element, secret_auth:(ARsp));
        AnySP -Ch_SP2C_2-> Actor : httpResponse(code_200, nil_agent,
                nil_http_element, SP_on_resource:(?Resource));
                  }
                }
      }
    }

    %%%%%%%%%%%%%%%% IDENTITY PROVIDER %%%%%%%%%%%%%%%%
    entity IdentityProvider (Actor, C, SP: agent, TrustedSPs:
        agent set, Ch_C2IdP, Ch_IdP2C: channel) {
      symbols
        ID    : int;
        URI   : agent;

      body {
                %% cambiare questo con On-Select
                select {
          on(C -Ch_C2IdP-> Actor   : httpRequest (get, Actor,
                httpBinding(authnRequest(?SP, Actor, ?ID),
                ?URI), nil_http_element) &
                     TrustedSPs->contains(?SP)): {
                       Actor -Ch_IdP2C-> C : httpResponse(code_200,
                                nil_agent, nil_http_element, htmlForm(
                                SP, postBinding(
                                signedAuthnResponse(inv(pk(Actor)),
                                SP, Actor, C, ID), URI)));
                  }
                }
      }
    }

    %%%%%%%%%%%%%%%% SERVICE PROVIDER %%%%%%%%%%%%%%%%
    entity ServiceProvider (Actor, IdP, C: agent, URI : agent,
        Ch_C2SP_1, Ch_SP2C_1, Ch_C2SP_2, Ch_SP2C_2: channel) {
      symbols
                %AnyC         : agent;
        ID            : int;
        Resource      : http_element;

      body {
                C  -Ch_C2SP_1-> Actor : httpRequest(get,
                URI, nil_http_element, nil_http_element);

        ID := fresh();
        Actor -Ch_SP2C_1-> C   : httpResponse(code_30x, IdP, httpBinding(
                authnRequest(Actor, IdP, ID), URI), nil_http_element);

        C  -Ch_C2SP_2-> Actor : httpRequest (post, Actor,
                nil_http_element,secret_auth:(postBinding(
                signedAuthnResponse(inv(pk(IdP)), Actor, IdP, C, ID),
                URI)));
                %assert finished: false;
        Resource := fresh();
        Actor -Ch_SP2C_2-> C   : httpResponse(code_200, nil_agent,
                nil_http_element, SP_on_resource:(Resource));

        %assert finished: false;
                SP_authn_C_on_uri:(URI) := URI;
      }
    }
    body {
          %% channel properties
          %unilateral_conf_auth(Ch_C2SP_1, Ch_SP2C_1, C, SP);
          confidential_to(Ch_C2SP_1, SP);
```

```
          authentic_on(Ch_SP2C_1, SP);
          weakly_confidential(Ch_SP2C_1);
          weakly_authentic(Ch_C2SP_1);
          link(Ch_C2SP_1, Ch_SP2C_1);

          %bilateral_conf_auth(Ch_C2IdP, Ch_IdP2C, C, SP)
          %that gets authentic to C after
          %her authentication using username/password
          confidential_to(Ch_C2IdP, IdP);
          authentic_on(Ch_IdP2C, IdP);
          confidential_to(Ch_IdP2C, C);
          weakly_authentic(Ch_C2IdP);
          link(Ch_C2IdP, Ch_IdP2C);


          %unilateral_auth_conf(Ch_C2SP_2, Ch_SP2C_2, C, SP);
          confidential_to(Ch_C2SP_2, SP);
          authentic_on(Ch_SP2C_2, SP);
          weakly_confidential(Ch_SP2C_2);
          weakly_authentic(Ch_C2SP_2);
          link(Ch_C2SP_2, Ch_SP2C_2);

      %% a new session is built
      new Client(C, SP, IdP, URI, Ch_C2SP_1, Ch_SP2C_1,
        Ch_C2IdP, Ch_IdP2C, Channels);
      new IdentityProvider(IdP, C, SP, TrustedSPs,
        Ch_C2IdP, Ch_IdP2C);
      new ServiceProvider(SP, IdP, C, URI, Ch_C2SP_1, Ch_SP2C_1,
        Ch_C2SP_2, Ch_SP2C_2);
    }
        goals
                SP_authn_C_on_uri:(_) C *->  SP;
                SP_on_resource:(_) SP *->  C;
                SP_on_resource:(_) {SP,C};
                secret_auth:(_){SP,C};

  }
  body {
    TrustedSPs := {sp, i};
        CChannels := {(sp, ch_c2sp_1s1, ch_sp2c_1s1),
        (i, ch_c2i_1s2, ch_i2c_1s2)};
    %% Sessions
    new Session(c, idp, sp, TrustedSPs, uri_sp, ch_c2sp_1s1, ch_sp2c_1s1,
        ch_c2sp_1s1, ch_sp2c_1s1, ch_c2idp_s1, ch_idp2c_s1, CChannels);
        % honest IdP and SP

        new Session(c, idp, i , TrustedSPs, uri_i , ch_c2i_1s2,
        ch_i2c_1s2, ch_c2i_1s2, ch_i2c_1s2, ch_c2idp_s2,
        ch_idp2c_s2, CChannels);
        % honest C talking to i(SP)
  }

}
```

D Infobase model

```
% @verbatim(Formalization of the Infobase case study)
% @modeler(Alberto Calvi, UniVr, 2011)
% @modeler(Marco Rocchetto, UniVr, 2011)
% @modeler(David von Oheimb, Siemens AG, 2012)
% @clatse(--lvl 1)

% Assumptions made while modeling this scenario:
% A_1) a User has already registered in the system
%      and has a defined UserName and Password.
% A_2) the User of A_1 has been also assigned to
%      a specific "role"

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

specification Infobase_Scene1
channel_model CCM

entity Environment {

  types
    usr    < text;
    pwd    < text;
    role   < text;
    cookie < text;
    request  < message;
    response < message;

  symbols
    usr1,
    usr2: usr;
    role1,
    role2: role;
    request1,
    request2: request;

    repository: agent;

    % the databases need to be shared by all sessions.
    nonpublic loginDB  : (usr,pwd,role) set;
    nonpublic cookiesDB: cookie set;
              cookie(usr,role,text): cookie;
    nonpublic noninvertible can_exec(role,request): fact;

  entity Session(UserIP:agent, UserName:usr, Role:role, Request:request) {

    symbols
      Password: pwd;
      nonpublic noninvertible checkPermissions(usr,request): fact;
      nonpublic noninvertible checkPermissions_dishonest(request): fact;
      nonpublic               answerOn(request): response;
      nonpublic noninvertible got(usr,response): fact;
      nonpublic noninvertible dishonest_usr(usr): fact;
    % needed to avoid spurious secrecy attacks

    clauses
      permission(Us,Pwd,Ro,Req):
        checkPermissions(Us,Req) :-
        loginDB->contains((Us,Pwd,Ro)) & Ro->can_exec(Req);
      % if a User has a given Password and Role
      % (checked via "contains")
      % and this role is given the right to for
      % the given Request (via "can_exec")
      % then the User has the permission for the
      % Request to the repository (checkPermissions)
      permission_dishonest(Us,Req):
        checkPermissions_dishonest(Req) :-
```

```
          dishonest_usr(Us) & checkPermissions(Us,Req);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %% the Client entity

    entity Client(UserName: usr, Password: pwd, Request: request,
                  Actor, Repository: agent) {

      symbols
        Cookie: cookie;
        Answer: response;

      body {
        % sends his/her name and password to the Repository
        Actor ->* Repository: Actor.UserName.secret_Password:(Password);
        % Note that the current agent name (Actor) is used as the UserIP.
        % The model should allow the intruder to spoof UserIPs,
        % and indeed it does because the agent names are public.

        % the repository login service responds to
        % the login request with a cookie
        % (after the Repository forwards the user name and password)
        select { on (repository *->* Actor: secret_Cookie:(?Cookie) &
                     ?Cookie=cookie(UserName,?,?)): {} }
        % A check of the UserName in the cookie is needed
        % to avoid silly attack

%%% for attack scenario cookie of honest user leaked/disclosed,
%%% uncomment the following line:
        %iknows(Cookie);
%%% it would be great to have automatic means for doing such muations

        % sends to the repository frontend the request and a cookie
        % that gives him access to the repository
        Actor *->* repository: Cookie.request_integrity:(Request);
        % for the Auth. goal (SP2)

        % the Repository sends back to the user the answer
        % of the repository
        % To avoid replay of answer that does not fit to request,
        % we added "Request":
        repository *->* Actor: Request.?Answer;
        UserName->got(Answer);

        %%% for executability test, uncomment the following line:
        %assert reached_breakpoint1: false;

      }
    } %% end of Client entity


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %% the online Repository entity

    entity Repository(UserIP, Actor: agent) {

      symbols
        UserName: usr;
        Password: pwd;
        Role    : role;
        Request : request;
        Answer  : response;
        Nonce    : text;
        Cookie   : cookie;
        Count    : nat;

      body {
        % it would be more realistic to have a main loop with "while(true)"
        % but we do use this variant because of
        % modelchecking performance problems.
        % The loop below allows for two rounds per Repository instance,
        % which should be sufficient.

        % translation optimization: not using "Count := 0"
        % here and using "?" in the next line saves use one ASLan rule
        while(Count != succ(succ(?))) {
        select {

          %% Case 1: login service receives the user request
          %% and generation of a new cookie for the session

          % receives authentication data from the user
          on(? ->* Actor: ?UserIP.?UserName.?Password &
          % cannot require secret_Password in Repository
          % as this would lead to spurious attack

          % anyone might try to authenticate as user

          % checks if the data are available in the database
          %% "select..on" is more efficient to model-check than "if"
            loginDB->contains((?UserName,?Password,?Role))): {
            % At this point, we have checked, using the password,
            % that the user is legitimate.
            % With the query, we extract the role of the legitimate user.
            % Yet since the user may still be dishonest,
            % the UserIP may be forged, and therefore
            % we do not state secret_Password:(Password) here,
            % as this implicitly relies on UserIP.
            % In this model, it is sufficient to state
            % secret_Password:(Password) for the Client.

              % creates the cookie and sends it back to the user
              Nonce := fresh();
              Cookie := secret_Cookie:(cookie(UserName,Role,Nonce));

              % adds the Cookie into the DB associated
              % with the name of the user
              cookiesDB->add(Cookie);

              % uses the IP address sent by the client
              % to communicate the cookie to the correct user
              Actor *->* UserIP: Cookie;
          }

          %% Case 2: having a cookie, a user makes a request to the frontend

          on(?UserIP *->* Actor: cookie(?UserName,?Role,?Nonce).?Request &

            % checks if the user is allowed to do this request
            %    and if the user is linked to the cookie
            checkPermissions(?UserName,?Request) &
            cookiesDB->contains(cookie(?UserName,?Role,?Nonce))): {

            % if the user has the right credential, then the Repository
            % sends the request to the repository, which will return the answer
            Answer := answerOn(Request);
            %% shortcut for simplicity: no real Repository

            Actor *->* UserIP: request_integrity:(Request).Answer;
            % for the Auth. goal (SP2)
          }

          % Case 3: otherwise the user is either
          % a cheater who hasn't achieved his
          % goal or a user that has an invalid
          % cookie to issue the request
          on(?UserIP *->* Actor: cookie(?UserName,?Role,?Nonce).?Request &

            % checks if the user is allowed to do this request
            %    and if the user is linked to the cookie
            checkPermissions(?UserName,?Request) &
            checkPermissions(?UserName,?Request) &
            cookiesDB->contains(cookie(?UserName,?Role,?Nonce))): {

            % if the user has the right credential, then the Repository
            % sends the request to the repository, which will return the answer
            Answer := answerOn(Request);
            %% shortcut for simplicity: no real Repository

            Actor *->* UserIP: request_integrity:(Request).Answer;
            % for the Auth. goal (SP2)
          }


        }
        Count := succ(Count);
        }
      }
    } %% end of Repository entity

    body { %% of Session

      % translation optimization: not using "Password := fresh()"
      % here, but distributing it into both "if" branches
      % saves use one ASLan rule
      % A_1) UserName has already registered in the system
      % and dishonest user(s) know their password(s)
      if(dishonest(UserIP)) {
        Password := fresh();
        dishonest_usr(UserName);
        iknows(Password);
      }
      else
        Password := fresh();

      % A_2) UserName has been assigned to a specific "role"
      loginDB->add((UserName,Password,Role));

      %% User and Login Service instantiation
      new Client      (UserName, Password, Request,
                       UserIP, repository);
      new Repository  (UserIP, repository);
    }

    goals
    % SP1.1 - confidentiality of passwords
```

```
    % the Password must be a secret between
    % any honest user and the repository login service
      secret_Password:(_) {UserIP, repository};

    % SP1.2??? - confidentiality of cookies
%%% the following goal should be commented it out
%%% for cookie leak/disclosed attack scenario:
      secret_Cookie:(_) {UserIP, repository};

    % SP2 - authorization constraints
    % only users having appropriate permissions
    % will receive answers on their requests

      authorization_honest: forall Us Req.
      [](Us->got(answerOn(Req)) => checkPermissions(Us,Req));

    % This goal is similar to authorization_dishonest, but too strong:
    % secret_Answer: forall Req. [](iknows(answerOn(Req)) =>
    %   dishonest_usr(UserName) & checkPermissions(UserName,Req));

      authorization_dishonest: forall Req.
      [](iknows(answerOn(Req)) => checkPermissions_dishonest(Req));

    % If the repository accepts a request from an honest user,
    % it has been issued by that user in the same session
      request_integrity:(_) UserIP *-> repository;

  } %% end of Session entity


  body { %% of the Environment entity

    % users having the role "role1" can execute "request1" etc.
    role1->can_exec(request1);
    role2->can_exec(request2);
    % translation problem: the above two statements
    % are not lumped with the following ones

    any UserIP. Session(UserIP, usr1, role1, request1)
    where !dishonest(UserIP);
    % second session (with potentially dishonest user)
    % needed for attack scenario
  % any UserIP. Session(UserIP, usr2, role2, request2);
    % takes too long
  % new         Session(i     , usr2, role2, ?       );
    % does not work for SATMC
    new         Session(i     , usr2, role2, request1);
  }
}

%%% For XEmacs:
%%% Local Variables:
%%% tab-width: 2
%%% End:

%  LocalWords: UniVr forall modelchecking frontend nat succ
%  LocalWords: UserName loginDB InfobaseScene UserIP UserIPs
%  LocalWords: checkPermissions createCookie cookiesDB
%  LocalWords: answerOn userAuth Auth Rcv Pwd Ro Req usr pwd auth
```