AVANTSSAR

Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures

FP7-ICT-2007-1, Project No. 216471

www.avantssar.eu

ASLan++ specification and tutorial

Abstract

This document is about ASLan++, the AVANTSSAR specification language. It is kept up-to-date with the latest developments related to ASLan++. ASLan++ has been designed for formally specifying dynamically composed security-sensitive web services and service-oriented architectures, their associated security policies, as well as their trust and security properties, at both communication and application level.

We introduce ASLan++ by means of a tutorial and a formal definition. The semantics of ASLan++ is defined by translation to ASLan, the low-level input language for the back-ends of the AVANTSSAR Platform.

Project details

Start date: *January 01, 2008* Duration: *36 months*

Project Coordinator: *Luca Viganò*

Partners: UNIVR, ETH Zurich, INRIA, UPS-IRIT, UGDIST, IBM, OpenTrust, IEAT, SAP, SIEMENS

SEVENTH FRAMEWORK PROGRAMME

Contents

List of Figures

List of Tables

1 Introduction

This document introduces and describes ASLan++, the AVANTSSAR specification language. ASLan++ is a formal language for specifying security-sensitive service-oriented architectures, their associated security policies, and their trust and security properties. The semantics of ASLan++ is formally defined by translation to ASLan, the low-level specification language that is the input language for the back-ends of the AVANTSSAR Platform.

This is an update of Deliverable D2.3 [5]. It not only further consolidates ASLan++ and ASLan, but also comes with a tutorial for the benefit of modelers learning how to use ASLan++.

Background on ASLan++. The first version of ASLan++ (initially called ASLan v.2) was introduced in deliverable D2.2 [4]. In deliverable D2.3 [5], we built upon that previous version in particular by adding extensive support for using channels abstractions, and also automating a number of optimization techniques. The present document covers further minor extensions and clarifications.

We have developed ASLan++ to achieve the following design goals:

- The language should be expressive enough to model a wide range of service-oriented architectures while allowing for succinct specifications.
- The language should facilitate the specification of services at a high abstraction level in order to reduce model complexity as much as possible.
- The language should be close to specification languages for security protocols and web services, but also to procedural and object-oriented programming languages, so that it can be employed by users who are not experts in formal protocol/service specification.

In order to support formal specification of static and dynamic services and policy composition, ASLan++ introduces a number of features.

- ASLan++ has similar features to mainstream imperative programming languages: assignment statments, control flow constructs (e.g. `if` and `while`), etc., that enhance the readability and conciseness of the specifications, and make the modeling task easier for someone used to writing programs.
- Modularity is supported by the use of entities. Each entity is specified separately and can then be instantiated multiple times and composed with others. This allows the specifier, in particular, to localize policies in each entity by clarifying, for instance, who is responsible to grant or deny authorization as well as the various trust relationships between entities.
- There is an intuitive notation for channels, which may be used both as assumptions and as service goals and provides a simple but powerful way to specify communication and service compositionality.

The AVANTSSAR Platform [6] supports ASLan++ via a connector that works in two directions: it translates ASLan++ specifications to ASLan and translates attack traces from ASLan level to ASLan++ level.

Structure of the document.
This document is self-contained in terms of ASLan++ and ASLan.

In order to facilitate its use for ASLan++ modelers, we start by an ASLan++ tutorial in § 2 illustrating the common usage of ASLan++ and the related AVANTSSAR tools via examples that contain typical specification patterns. Next comes a detailed introduction to the ASLan++ concepts and syntax in § 3. The formal semantics of ASLan++ is defined in § 4 via a procedure for translating ASLan++ specifications into ASLan specifications. This procedure serves as a basis for the implementation of the ASLan++ connector/translator tool integrated into the AVANTSSAR Platform. We conclude the main document in § 5.

In Appendix A, we give a complete description of ASLan.

Appendix B gives a brief overview on the options of the ASLan++ connector and the various model checking back-ends.

2 ASLan++ tutorial

In this section, we aim to give a gentle introduction to ASLan++. We demonstrate using small examples, starting with the well-known Needham-Schroeder Public Key (NSPK) Protocol, how to write specifications that can be successfully translated and checked using the back-ends.

We give a number of practically important general hints and tricks and then provide and explain a number of useful typical specification patterns, used e.g. in the AVANTSSAR Library [7] of industrial use cases.

The examples include using sets, with operations like union and intersection, as well as checking properties like set inclusion and emptiness. We introduce various approaches to handle shared data and to communicate sets in ASLan++. Moreover, we demonstrate how to model a simple timeout mechanism, a basic hierarchical Role-Based Access Control (RBAC) system, and various aspects of TLS-style communication channels.

For each of these examples, we give a complete ASLan++ model that can be tried out. All models, plus a number of benchmark tests for exploring various features of ASLan++ and the AVANTSSAR Tools, are available for download on the AVANTSSAR home page at http://www.avantssar.eu/, via the menu items Test Library and Avantssar Platform. There you will also find directions to online versions of the AVANTSSAR Tools.

2.1 Hello, World!

This subsection describes the first steps with modeling in ASLan++, as well as using the translator and the AVANTSSAR Platform and its back-end model checkers. It is mainly targeted to those who are used to designing communication protocols and to application-level programming in a contemporary language like Java. Familiarity with formal modeling is not a prerequisite.

2.1.1 The Problem

As a running example, we use variants of the simple well-known Needham-Schroeder Public Key (NSPK) Protocol [21]. This protocol is intended to provide fresh mutual authentication between two parties communicating over a network, but in its proposed form is insecure. There is an attack [13] which the tools can detect and the reader can correct easily. Other protocol weaknesses are described in the literature but that are not of interest here.

We give a walk through the whole range of steps required from understanding the original protocol to correcting it based on the feedback provided by the model checkers.

The protocol runs between two parties, `A` (in role Initiator) and `B` (in role Responder). In the simple “Alice-and-Bob notation” [2], it reads as:

```
A -> B: {Na.A}_Kb
A <- B: {Na.Nb}_Ka
A -> B: {Nb}_Kb
```

Here, Na and Nb stand for nonces (i.e., non-guessable random values) produced freshly by the Initiator and Responder, respectively. Line 1 says that the Initiator sends her nonce and her name to the Responder, readable for B only. A invents Na and together with her name sends it via -> to B, encrypted with his public key Kb. The "." in Na.A means concatenation of two message parts.

Lines 2 and 3 specify similar message transmissions. B invents Nb, and sends it to A along with Na to prove to her his ability to decrypt with his private key inv(Kb). A confirms Nb to B, to prove to him her ability to decrypt with inv(Ka). Line 2 implies that A checks the value Na she receives; likewise line 3 implicitly requires that B checks the value Nb he receives.

At the end of the protocol, A and B are supposed to know each other's identities, and know both Na and Nb. These nonces may not become known to attackers. They must be kept secret between A and B.

2.1.2 Automated Security Analysis

Is the protocol described above secure? Can it be attacked? Experience shows that manual analysis is cumbersome and may overlook vulnerabilities that are more complex to exploit (this protocol is famous precisely for this reason and has become the standard example for the importance of the use of formal methods for protocol analysis). Our approach is to let an automated model checker do the hard work. How do we proceed from above informal protocol sketch? We now outline the procedure followed in the rest of this section.

Approach.

First, we formalize the protocol in cooperation with the protocol expert who is familiar with the protocol and its typical deployment scenarios. A useful modeling and discussion base are annotated sequence diagrams.

An automated model checker must be told explicitly what the security goals of the protocol are. Therefore, together with the protocol expert, we specify the security goals of the problem. Valuable discussion partners also are those parties that are to deploy the protocol. Their security goals should be covered by the protocol.

Next, we model the problem and the goals in ASLan++. Then we translate (compile) the problem into ASLan, which is the language understood by the model checkers. We feed the compiled model to the automated model checkers and interpret their feedback. If a model checker detects an attack trace, we correct the model and re-check it. Once no more vulnerabilities are found, we fix the original protocol.

This approach is followed within this introductory example.

The Attacker Model.

In the common ASLan++ attacker model, the attacker is *the network*, as sketched in Figure 1. The attacker is handed every message sent by the sender and he forwards it to the recipient (or not). The attacker may analyze, manipulate, generate, suppress, delay, reorder, replay, and divert arbitrary messages.

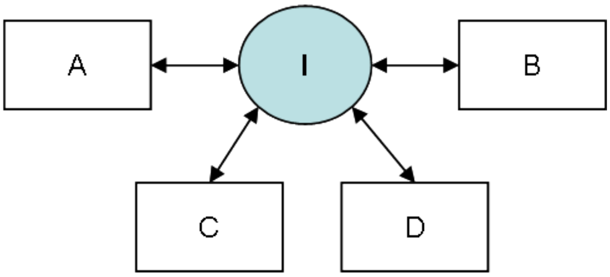


Figure 1: How the Model Checker simulates the Attacker.

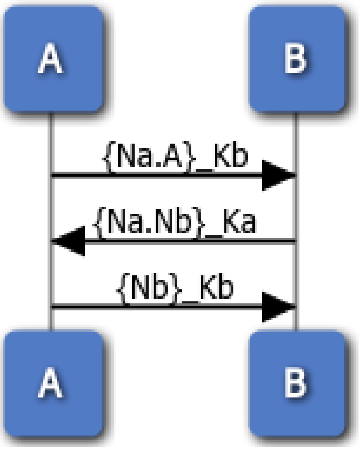


Figure 2: The Sequence Chart for the NSPK Protocol.

However, the attacker knows no secret keys unless they are disclosed to the attacker (he of course knows his own keys) and, in general, cannot break cryptography (e.g., he cannot decrypt or sign without knowing the respective key, nor un-hash a hashed message). An attacker may act using several different pseudonyms, who all share the same knowledge.

The model checkers automatically simulate an attacker. This attacker systematically tries to exchange messages with the lawful protocol parties, following and corrupting the intended protocol message flow at will. The attacker reads messages and impersonates honest parties. The attacker may even exploit several simultaneous protocol sessions, for instance by relaying messages from one session to the other session, and vice versa. The model checkers try automatically to evaluate every possible situation. However, complexity increases with every involved party and especially with every additional simultaneous session. This may cause the model checkers to require a long time for evaluation. Sometimes a model checker may even be overwhelmed by the problem complexity. Therefore, we must be careful about the number of involved parties and parallel sessions we request to be simulated.

2.1.3 Formalizing Behavior

Message Sequences.

Frequently, communication protocols as NSPK are developed and discussed using message

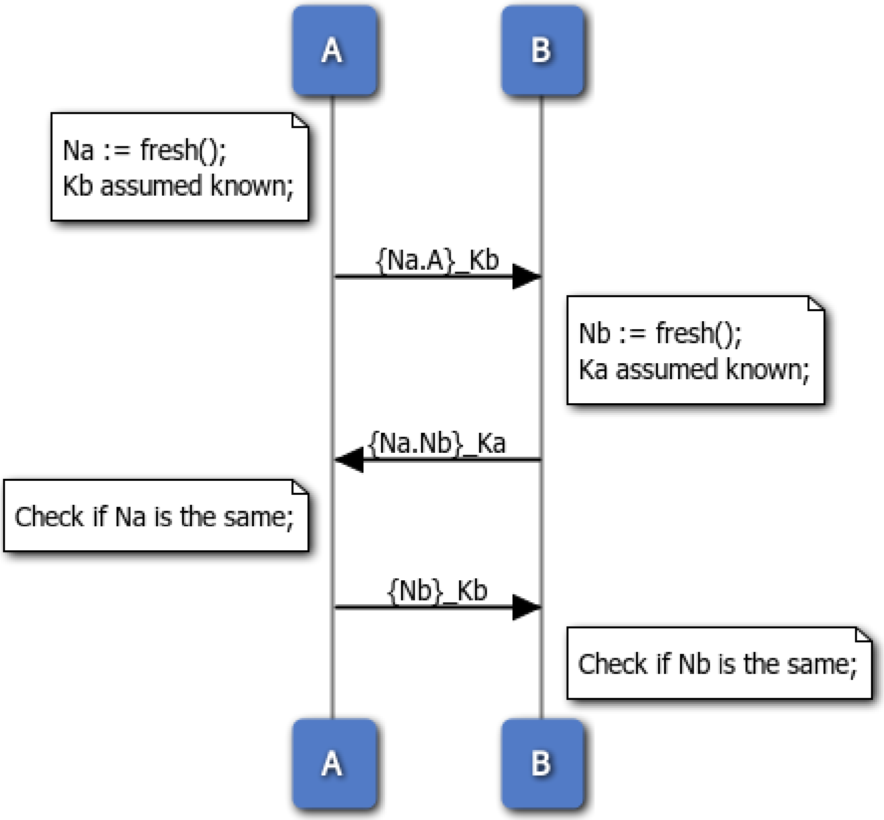


Figure 3: The annotated Sequence Chart for the NSPK Protocol.

sequence charts, as for instance depicted in Figure 2. For better efficiency it is advisable to use a text-based sequence-chart editor, like the “Quick Sequence Diagram Editor”[1] or “Web Sequence Diagrams”[2]. For instance, “Web Sequence Diagrams” takes as input

```
A -> B: {Na.A}_Kb
B -> A: {Na.Nb}_Ka
A -> B: {Nb}_Kb
```

and automatically produces as output the chart shown in Figure 2.

Role Behavior.

Usually, a communication protocol specification contains details not expressible in Alice-and-Bob notation or simple message sequence charts. In our NSPK example, such actions are for instance how and when values like nonces and agent names are computed and checked. That is, `Na := fresh()` and `Nb := fresh()` are newly created random values, public keys `Ka` and `Kb` are assumed as known, and `Na` and `Nb` are checked for equality by their respective issuers.

[1] http://sdedit.sourceforge.net

[2] http://www.websequencediagrams.com

One method to sketch such behavior in message sequence charts are annotations, as depicted in Figure 3.

The input of "Web Sequence Diagrams" corresponding to Figure 3 is:

```
note left of A
    Na := fresh();
    Kb assumed known;
end note
 A -> B: {Na.A}_Kb
note right of B
    Nb := fresh();
    Ka assumed known;
end note
 B -> A: {Na.Nb}_Ka
note left of A
    Check if Na is the same;
end note
 A -> B: {Nb}_Kb
note right of B
    Check if Nb is the same;
end note
```

2.1.4 Specifying Security Goals

We want to model our protocol in such a manner that an automated model checker can look for attacks. To be able to search systematically for ways an attacker may violate the desired protection requirements, the model checker needs to be told what these requirements are. To this end, we must explicitly specify security goals. We name our goals, so the model checkers can refer to them when printing which goals they have found violated.

In our NSPK example, the primary security goals are that the Initiator freshly authenticates the Responder and vice versa, agreeing on two nonces. There are secondary security goals expressing that the values of both nonces are secrets shared between the Initiator and Responder only. Thus, we face two main types of security goals, *secrecy goals* and *authentication goals* as representatives of a message *communication security goal.* Typical communication security properties are authenticity, integrity and confidentiality, as well as replay protection (freshness).

Secrecy Goals.

Secrecy goals specify who may gain knowledge of a certain piece of information. In our NSPK example, only one instance `A` of the Initiator and one instance `B` of the Responder may know `Na` and `Nb`. These nonces may not become known by the attacker (unless he legitimately plays the role of `A` or `B`). Thus, we have two secrecy `goals`, namely `secret_Na:(Na) {A,B}` and `secret_Nb:(Nb) {A,B}`. These secrecy goals are noted down in the annotated message sequence chart in Figure 4. The model checkers can refer to the goal names `secret_Na` and `secret_Nb` if they find a way to violate any of them.

To implement this secrecy, the Initiator and Responder, who are assumed to know each other's public keys, can send confidential messages to each other. Only the intended recipient can decrypt the message. Furthermore, the Initiator and Responder may not disclose these nonces to any other party, for instance by sending them to others.

Communication Security Notions.

ASLan++ offers an intuitive arrow notation used for both communication security assumptions[3] and goals. These properties[4] only refer to individual communication events, that is, of a single payload at a certain point in the system execution. Communication security properties are unidirectional. That is, transmission of a message from `A` to `B` can have different security properties than that from `B` to `A`.

- `A -> B: M` indicates "default" communication without any protection.
- `A ->* B: M` denotes *confidential* communication. `A` sends some payload `M` that only `B` can read. However, `B` cannot be sure of the identity of `A` (as long as `A` has not been authenticated) so `B` cannot be sure this `M` has really been sent by `A`.
- `A *-> B: M` denotes *authentic* communication. `B` can be sure of the identity of `A`, as `A` has been authenticated. The `M` that `B` receives here indeed comes from `A`. It has not been altered in transmission, i.e., the integrity of `M` is granted. Moreover, the intended recipient `B` is included into the integrity protection, so `B` can rely on the message being meant for him. However, the communication is not confidential; thus any eavesdropper can access the contents in transit from `A` to `B`.
- `A *->* B: M` denotes *secure communication*, in the sense that `A` has been authenticated, the payload `M` is integrity and confidentiality protected, and it is intended by `A` for `B`. It does not protect against message replay, re-ordering, suppression, and delay, as well as traffic analysis.
- `A *->> B: M` denotes *replay-protected (fresh) communication*. The message `M` sent by `A` is accepted only once by `B`. This notation may only be used for communication including at least authenticity.

Goals for NSPK.

The main goal of the Initiator and Responder in our NSPK example is strong mutual

[3] With an assumption we declare that a communication already has special security properties without saying how they are achieved. They just are assumed present by us and the model checker. We an assumption if we do not want to bother how to achieve the property. We will not use communication security assumptions in our NSPK example, but rather implement the security ourselves.

[4] In the following, we assume communication security implemented via encrypted and signed messages, using the AVANTSSAR CCM (Cryptographic Channel Model). For more details and other channel models, see § 3.8.

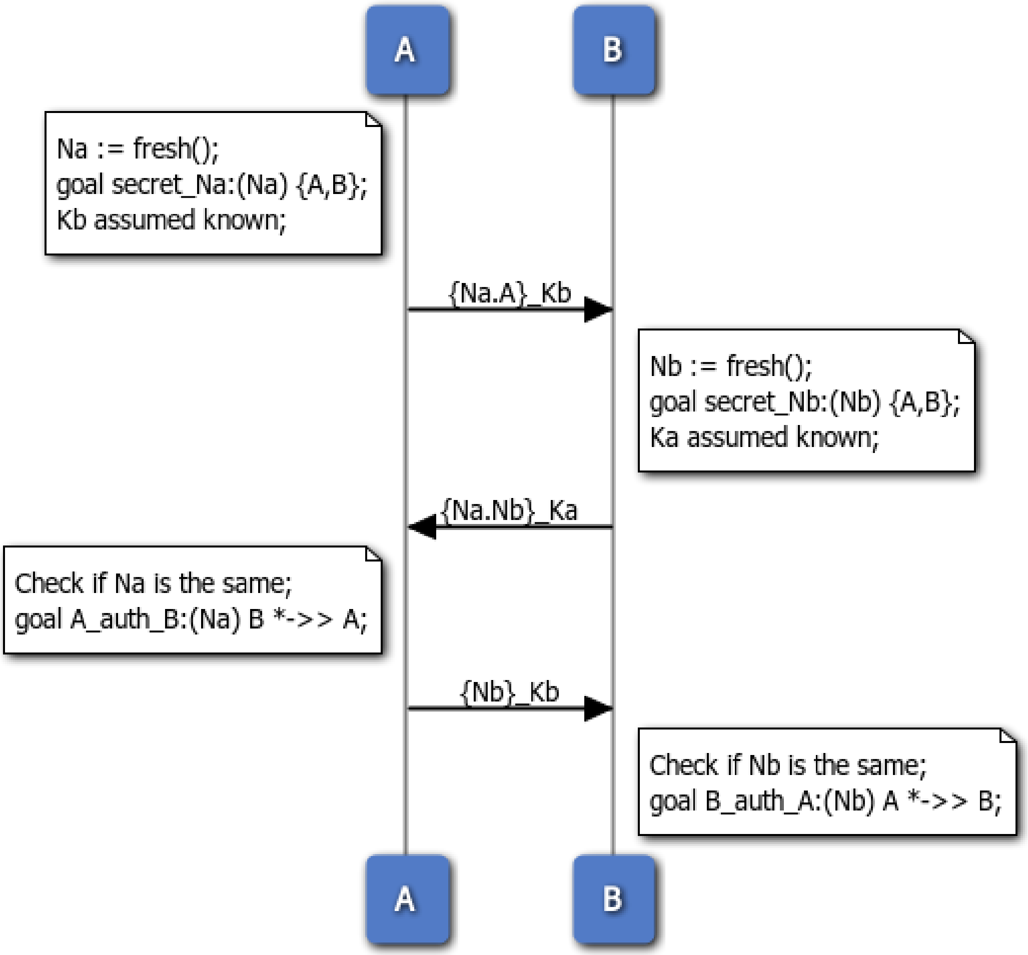


Figure 4: The Security Goals for the NSPK Protocol.

authentication: after checking of nonces, `A` and `B` expect that they have securely and freshly authenticated the other party.

Thus, they require as communication goals two authentic and fresh message transmissions, one from the Initiator to the Responder and one from the Responder to the Initiator. `A_auth_B:(Na) B*->>A` and `B_auth_A:(Nb) A *->> B` are the primary authentication goals. Their names are `A_auth_B` and `B_auth_A`.

The secondary secrecy goals for the transmitted nonces (see above) are `secret_Na:(Na) {A,B}` and `secret_Nb:(Nb) {A,B}`. Neither nonce may be disclosed to the attacker.

These goals are sketched in the annotated message sequence chart in Figure 4. The corresponding input to “Web Sequence Diagrams” is:

```
note left of A
    Na := fresh();
    goal secret_Na:(Na) {A,B};
    Kb assumed known;
end note
 A -> B: {Na.A}_Kb
note right of B
    Nb := fresh();
    goal secret_Nb:(Nb) {A,B};
    Ka assumed known;
end note
 B -> A: {Na.Nb}_Ka
note left of A
    Check if Na is the same;
    goal A_auth_B:(Na) B *->> A;
end note
 A -> B: {Nb}_Kb
note right of B
    Check if Nb is the same;
    goal B_auth_A:(Nb) A *->> B;
end note
```

2.1.5 Modeling the Overall Problem

ASLan++ Entities.

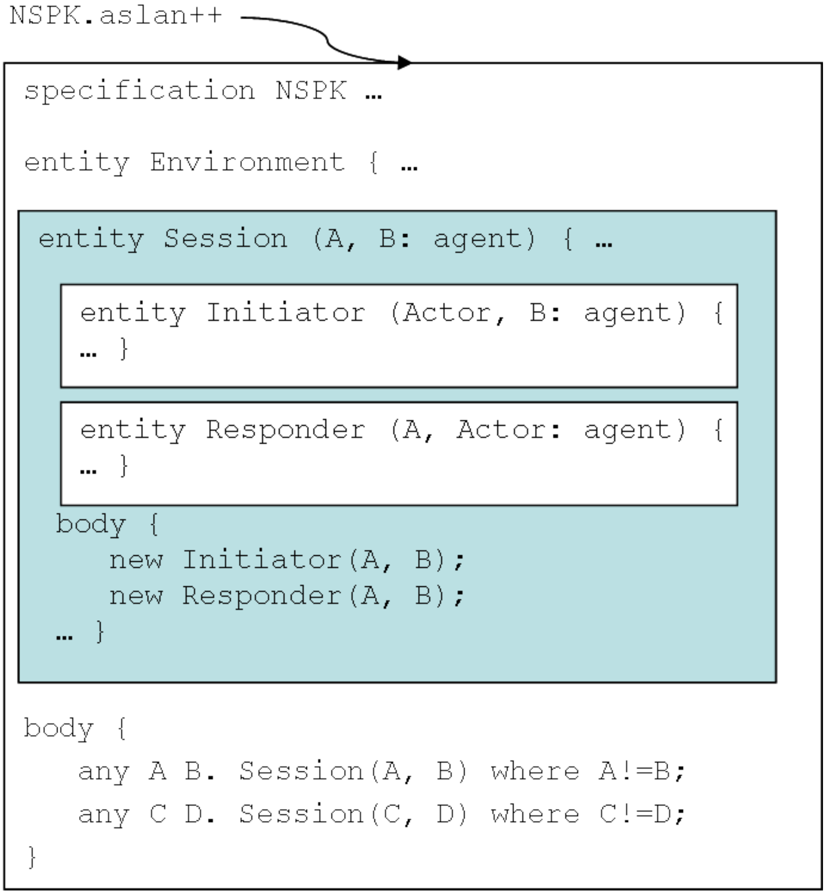
So far, everything we have discussed in this example is (more or less) independent of ASLan++. We have clarified and, to a certain degree, formalized our problem. Now, we make the transition to modeling the problem in ASLan++.

An ASLan++ specification of a system and its security goals has a name (which should be consistent with the name of its file) and contains the declaration of a hierarchy of *entities*, as sketched in Figure 5.[5]

The major ASLan++ building blocks are *entities*, similar to classes in Java or roles in security and communication protocols. Entities declare various symbols, macros, goals, and other items required by the entity and describe the entity behavior. Entities may have parameters and contain nested sub-entities, which inherit the items of their outer entities.

The top-level (i.e. outermost) entity usually is called `Environment`. It serves as the global root of the system being specified, similarly to the "main" routine of a conventional program. Inside the environment normally we define an entity called `Session`. A session contains further sub-entities. Each entity describes a role, that is, a behavioral pattern, that may be instantiated one or more times to a process, where each instance is played by some agent. Within an entity declaration, the `Actor` parameter is used to denote the agent playing the current instance of the entity, similarly to the `this` expression in an object-oriented language like Java. In our NSPK example, a session is conducted between

[5] A *channel model* characterizes the basic communication model of the specification. *CCM (Cryptographic Channel Model)* stands for communication security implemented via encrypted and signed messages. For more detail and other channel models, see § 3.8.



```
NSPK.aslan++

specification NSPK …

entity Environment { …

entity Session (A, B: agent) { …

  entity Initiator (Actor, B: agent) {
  … }

  entity Responder (A, Actor: agent) {
  … }
 body {
    new Initiator(A, B);
    new Responder(A, B);
 … }

body {
   any A B. Session(A, B) where A!=B;
   any C D. Session(C, D) where C!=D;
}
```

Figure 5: Nested Entities: Environment, its Sessions and their Actors.

two entities, `Initiator` and `Responder`. Any instance of `Initiator` (e.g, `a`), will initiate the NSPK protocol with some instance of `Responder`, therefore she must be told which Responder (e.g, `b17`), to contact.

The environment is instantiated once. Inside it, one or several sessions may be instantiated, which is specified using the `new` (or `any`) statements. The sessions instantiate their sub-entities, which communicate with each other following the specified protocol. In our example, as sketched in Figure 5, each session instantiates two processes, one playing the role Initiator and one the role Responder, based on the agent parameters handed to it by the environment. Note that an agent may execute one or more entity instances at the same time, and in general even the attacker may play such roles (whereas, being dishonest, he is not bound to follow the behavioral pattern defined in the respective entity bodies).

Each entity first declares its *symbols* and other items it requires. This declaration is followed by the entity's *body*, which specifies the entity's behavior. The body is executed when the entity is instantiated. An entity may also specify security *goals*.

```
entity Initiator(Actor, B: agent) {
```

$$\{\{(A,B),\ (C,D)\} \mid A \neq B\ \wedge\ C \neq D\ \wedge\ \{A,B,C,D\} \subseteq \{a1,a2,a3,a4,\mathtt{i}\}\} \tag{1}$$

Figure 6: Two parallel sessions – multiple interference possibilities.

```
    symbols Na, Nb: text;
    body {
       ...
    }
}
```

In our example, the entity `Initiator` has two parameters: next to the usual one referring to her current agent by the keyword `Actor`, she obtains a parameter `B` (of type `agent`), used to tell her which agent is her communication partner. She declares that she will need two variables (indicated by the keyword `symbols`), `Na` and `Nb` of type `text`[6]. In the `body` part, the Initiator's behavior is specified. Within that body, the agent running the current instance of Initiator can be referenced by using the keyword `Actor`.

ASLan++ Session Instantiation.

In the body of the environment, sessions are instantiated. To find an attack on NSPK, we need the Initiator to conduct at least two sessions in parallel.

```
body { % need two sessions for Lowe's attack
    any A B. Session(A,B) where A!=B;
    any C D. Session(C,D) where C!=D;
}
```

Therefore, we instantiate two so-called "symbolic sessions" between `A` and `B`, where `A` may be an honest `Initiator` instance or the dishonest attacker, also known as the intruder `i`), and `B` may be an honest `Responder` instance or the attacker. For details on symbolic sessions, cf. § 3.10. In this specification, two sessions of the protocol are launched, each involving one pair of agents with symbolic names `A` and `B`. However, we disallow entities talking to themselves by requiring the side condition `A!=B`. So, for instance, a session of `a1` and `a1`, which would lead to `a1` playing the roles of both Initiator and Responder, such that `a1` talks to herself, is not taken into account.

The model checker will evaluate the combinations specified in Figure 6, where the 'set' of two pairs is meant as a multiset (this is, a pair may occur twice, like this:$\{(a1,a2),\ (a1,a2)\}$). For two parallel sessions, the model checker, taking into account symmetries, will evaluate more than 6 different pairs of sessions. The model checker will interleave these parallel session pairs in all different protocol-compliant variants possible. It makes sense to be careful about the numbers of sessions one instantiates in parallel as the workload for the model checker

[6] The ASLan++ standard prelude defines some fundamental types like `fact` (anything that can be introduced and retracted as global meta-information) and `message` (anything that can be sent over a network). It also specifies that `text` and `agent` are subtypes of `message` and introduces a number of pre-defined constants. For more details on the standard prelude, see § 3.13.

explodes with the number of parallel sessions. This may preclude receiving an answer in reasonable time.

ASLan++ Implicit Operations.

In ASLan++, many operations are performed implicitly. Among these are the learning of new values received by message passing, the checking for equality of received values to already known values and the decryption of encrypted values.

We have seen that the Initiator and Responder, by message passing, need to learn the values of nonces and names and need to check them for equality. In ASLan++, this is done implicitly by pattern matching. For instance, let's look at the specification of entity `Responder`.

```
 1 entity Responder (A, Actor: agent) {
 2
 3    symbols
 4       Na, Nb: text;
 5
 6    body {
 7       ? -> Actor: {?Na.?A}_pk(Actor);
 8       secret_Nb:(Nb) := fresh();
 9       Actor  ->  A: {Initiator_authenticates_Responder:(Na).Nb}_pk(A);
10       A  ->  Actor: {Responder_authenticates_Initiator:(Nb)}_pk(Actor);
11       secret_Na:(Na) := Na;
12    }
13 }
```

In line 7, the Responder receives a message from some party, which at this point is still unknown to him. This is denoted by the `?` on the left-hand side of the message passing definition. He learns the name the sender claims to have and memorizes it in his variable `A`[7]. Learning a value for `A` is indicated by the `?` preceding `A`.

By `?Na`, the Responder also learns the value `Na`, which implicitly is decrypted using his private key corresponding to his public key `pk(Actor)`.[8]

In line 9, the Responder sends `A`'s nonce `Na` and his freshly generated own nonce `Nb` (from line 8) to that `A` that he has learned in line 2, encrypted with that `A`'s public key.

In line 10, the Responder receives his nonce back from `A`. This nonce must have the same value as the one he sent to `A`. Checking for equality is performed implicitly and indicated by the absence of the `?`. Again decryption is performed implicitly, as the Responder's private key is available within the entity `Responder`.

[7] One may wonder, why `Responder` is given `A` as parameter by `Session` and then plainly ignores its value. The reason for this is as follows. The translator/compiler from ASLan++ to ASLan needs to know with which agent variables inside the `Responder` entity the security goals given in the `Session` entity are connected. Therefore, we need not only declare the `Agent` parameter of `Responder` but also the `A` parameter standing for the Initiator, although the value passed during instantiation of `Responder` is overwritten later on receiving the first message.

[8] Every agent automatically is assigned a default public and private key pair. Definitions are part of the ASLan++ standard prelude. For details, see § 3.13.

The security goal names secret_Nb (line 8), Initiator_authenticates_Responder (line 9), Responder_authenticates_Initiator (line 10), and secret_Na (line 11) refer to security goals that must be met and may not be violated. We look at this in more detail in the following.

ASLan++ Goal Specification Conventions.
For reference by the model checker, in ASLan++ every goal needs to be given a name. Naming conventions propose to name a *secrecy goal* by the item that is to be kept secret, and a *channel goal* (a communication goal, e.g. authentication, integrity, confidentiality, replay protection, ...) by the sender, receiver, specific security characteristic such as authentication, and (within the respective sub-entities) the message value that is communicated.

Security goals are specified within entity definitions. If more than one entity wants to require a given security goal to be met, the security goal must be declared in an enclosing entity. If not all parameters required in the security goal specification are already available within the enveloping entity, we must use security goal templates. For this we first declare a security goal template within the outer entity and then fill in the template within the inner entities, after the missing parameters have been declared and the security goal is required to be met by the inner entity.

Let us look at our NSPK example. The Initiator and Responder both want to require the two nonce secrecy goals and the two authentication goals to be met. Thus, we specify the security goals in the enveloping entity, which is Session. However, Session cannot access the nonces that the Initiator and Responder instances will use, as they declare them locally, so we specify the security goals as templates within Session and fill in these templates within Initiator and Responder, when they refer to the respective goals.

After the specification of Session's body, we specify – still within the definition of the entity Session – our security goal templates as follows.

```
 1 entity Session (A, B: agent) { ...
 2
 3    body { ... }
 4
 5    goals
 6      secret_Na:(_) {A,B};
 7      secret_Nb:(_) {A,B};
 8      Initiator_authenticates_Responder:(_) B *->> A;
 9      Responder_authenticates_Initiator:(_) A *->> B;
10 }
```

In lines 6 and 7 of the above listing of the Session's goal section, the secrecy goals by the names of secret_Na and secret_Nb are specified as templates. They require some still undeclared item to be kept secret between A and B. A and B must fill in the placeholder for that item, (_), with Na and Nb respectively, when they are actually requiring the security goal to be met.

The same applies to the authentication goals named Initiator_authenticates_Responder and Responder_authenticates_Initiator in lines 8 and 9 of above listing. These goals require that some hitherto undeclared item is transmitted in an authenticated and fresh manner (`*->>`) from one party to the other, thus allowing the receiving party to authenticate the sending party and offering replay-protection at the same time.

In lines 8, 9, 10, and 11 of the listing of body of entity `Responder` in the previous paragraph, these four security templates are referenced and filled in. In line 8 the secrecy goal `secret_Nb` is concretized for `Nb`. Besides the Responder instance, only the Initiator instance (referred to by `A`) may learn of it. Neither party may disclose `Nb` to any other party and it must of course be secured in transmission. The equivalent applies to `A`'s nonce `Na`, as stated in secrecy goal `secret_Na` in line 11[9].

The authentication goal `Initiator_authenticates_Responder` in line 9 is concretized for the message transmission of `Na` from the Responder instance to `A`. The Responder's goal is to let `A` authenticate him via the reflection of `Na`. The equivalent applies to the Responder's nonce `Nb` that the Responder expects to let him authenticate `A` on, when she reflects `Nb` to him in line 10.

The goal labels of a channel goal are written on both communication endpoints immediately in-line with the corresponding communication event the goal refers to.

The goal labels of a secrecy goal are also specified at each entity that share the secret value, and as soon after generation or reception as possible. The resulting redundancy (i.e., duplication among the various "knowers") is intentional in order to fully capture also the cases where part of these parties are played by the intruder.

The complete ASLan++ Model.

In ASLan++, the complete protocol reads as follows, where comments start with a "%" symbol and extend until the end of the line.

```
% @satmc(--mutex=1)

specification NSPK_Unsafe
channel_model CCM

entity Environment {

  entity Session (A, B: agent) {

    entity Initiator (Actor, B: agent) {

      symbols
        Na, Nb: text;

      body {
        secret_Na:(Na) := fresh();
```

[9] Specified here, as only at this point the Responder instance finally feels confident about `A`'s identity. Before, the Responder could also have been talking to an attacker.

```
        Actor  ->  B: {Na.Actor}_pk(B);
        B  ->  Actor: {Initiator_authenticates_Responder:(Na).
                                              secret_Nb:(?Nb)}_pk(Actor);
        Actor  ->  B: {Responder_authenticates_Initiator:(Nb)}_pk(B);
      }
    }

    entity Responder (A, Actor: agent) {

      symbols
        Na, Nb: text;

      body {
        ? -> Actor: {?Na.?A}_pk(Actor);    % Responder learns A here!
        secret_Nb:(Nb) := fresh();
        Actor  ->  A: {Initiator_authenticates_Responder:(Na).Nb}_pk(A);
        A  ->  Actor: {Responder_authenticates_Initiator:(Nb)}_pk(Actor);
        secret_Na:(Na) := Na;    % Goal can only be given here,
                       % because A is not authenticated before!
      }
    }

    body { % of Session
      new Initiator(A,B);
      new Responder(A,B);
    }

    goals
     %secret_Na:(_) {A,B};                             % Okay.
     %secret_Nb:(_) {A,B};
% Attack found!
     %%% Commented out the above goals such that
     %%% the attack is printed on authentication.
      Initiator_authenticates_Responder:(_) B *->> A;  % Okay.
      Responder_authenticates_Initiator:(_) A *->> B;
% Attack found!
  }

  body { % of Environment
  % Two sessions needed for Lowe's attack.
    any A B. Session(A,B) where A!=B;
    any C D. Session(C,D) where C!=D;
  }
}
```

The entity Initiator, describing her behavior and security requirements (as long as she is played by an honest agent), has two parameters of type agent: Actor is used to refer to herself, and B is the name of the party she is supposed to communicate with. The entity Responder obtains the name of A via the first message it receives[10].

Please note that we have intentionally commented out the secrecy goal templates in the Session entity. The reason for this will be explained below, when analyzing the results of the model validation.

2.1.6 Validating the Model

Translating the ASLan++ Model to ASLan.

We have specified our NSPK model in the high-level language ASLan++. Now, we must translate it into the lower-level language ASLan, which the model checkers accept as input. Translation is done automatically by a compiler, also known as ASLan "connector".

The translator and its usage are described in detail in Deliverable 4.2 [6, §4.1]. For a brief overview of its options you can also refer to Appendix B. The translator is available in both an online and offline version. For directions please refer to the AVANTSSAR home page at http://www.avantssar.eu/ - Menu Item Avantssar Platform.

Assume that the above listed model is placed in a file named NSPK.aslan++. Then we can translate ("compile") our ASLan++ model with this command:

```
java -jar aslanpp-connector.jar -gas NSPK.aslan++ -o NSPK.aslan
```

The -gas option is recommended, as it makes the goals easier to handle for the automated model checkers[11].

The error messages and warnings of the compiler should be largely self-explanatory. Should you encounter a less telling message like "cannot proceed at ...", then it is best to look in close proximity of the location indicated in the message for a syntax error violating the grammar given in § 3.12.

Since the secrecy goal templates in the Session entity are commented out, on executing the above command you will receive a list of warnings from the translator that undefined secrecy goals are referred to in the Initiator and Responder entities. For now you can safely ignore these warnings and use the resulting NSPK.aslan file for analysis by the model checkers.

Checking the ASLan Model.

We can now submit the ASLan model to one or more of the back-ends of the AVANTSSAR Platform, namely CL-AtSe, OFMC, and SATMC. All three model checkers are available in

[10] Responder is given A as parameter by Session and then plainly ignores its value. This, as remarked already, is necessary, because the translator from ASLan++ to ASLan needs to know which agent variables inside the Responder entity the security goals given in the Session entity are connected to.

[11] The option -gas can also be given more verbosely as --goals-as-attack-states. It transforms, as far as possible, goal formulas as attack states, which can be handled by the model checkers more efficiently than goals in LTL (Linear Temporal Logic) form.

both an online and offline version. For directions please refer to the AVANTSSAR home page at http://www.avantssar.eu/ - Menu Item Avantssar Platform.

The back-ends are based on different technologies and thus provide complementary strengths, for instance: CL-AtSe efficiently deals with Horn clauses and exhibits good performance in general; OFMC supports algebraic reasoning for a few important operators such as the commutativity of exponentiation, a proved-correct pruning of the search space, compositionality, and an experimental combination with abstract-based interpretation; and SATMC supports LTL goals, all channel models, and Horn theories.

We can upload the ASLan file to the model checker service of our choice and let the model checker execute the model remotely. Alternatively, we can use the offline binary of a model checker locally, e.g. we use CL-AtSe on our ASLan file by executing

```
cl-atse NSPK.aslan
```

The search behavior and the attack output of the model checkers may be tuned by certain options. For instance, OFMC usually needs both the `--classic` and the `--untyped` option. CL-AtSe requires the `--nb` n option with n being least 2 in order to deal with loops and requires the `--not_hc` option to deal with negations in clauses. For detailed instructions on how to use and tune the various AVANTSSAR model checkers, please see Appendix B and refer to Deliverable 4.2 [6].

The options to fine-tune the model checkers are usually given manually as command line option each time the model checker is executed. ASLan++ models contain at the very beginning of their specifications hints which directives to use. Here as a special comment each model checker can be assigned options that one should use whenever the model is analyzed. For instance `% @clatse(--lvl 2)` refers to an option for CL-AtSe; likewise we use the keywords `satmc` and `ofmc` to hint at options for the model checkers SATMC and for OFMC. An example for such directives can be found below in § 2.1.7.

Interpreting the model checkers output.
The model checkers produce an abstract low-level output format, called *AVANTSSAR output format* (OF). Each model checker offers various options to tweak the output. For instance, one may launch CL-AtSe with the option "`--of if`", which results in a very detailed message-sequence-chart-like output of traces at ASLan level. In traces produced by OFMC, variable names like `x2602` may appear when an attack is possible for any (non-constrained) value at the given term position(s).

The ASLan++ connector can transform the low-level AVANTSSAR OF back to an output very close to the ASLan++ level. For example, the commands

```
cl-atse NSPK.aslan >NSPK.atk;
java -jar aslanpp-connector.jar NSPK.aslan -ar NSPK.atk
```

produce the following output:

```
INPUT:
           NSPK_Unsafe.aslan
```

```

SUMMARY:
        ATTACK_FOUND

VIOLATED:
   auth_Responder_authenticates_Initiator[AM=n21(Nb), AR=AR(2), AW=AW(2), IID=n23(IID_4)]  % on line 50

DETAILS:
        TYPED_MODEL

BACKEND:
        CL-ATSE 2.5-15_(2012-mai-10)

COMMENTS:
        CLAUSES coding internal intruder's actions are not shown
        EXPLICIT section shows only facts not removed by the simplification
        IMPLICIT section shows only facts needed for showing insecurity

STATISTICS:
        TIME 288 ms
        TESTED 2115 transitions
        REACHED 1110 states
        READING 0.02 seconds
        ANALYSE 0.27 seconds

STATES:
   Environment[Actor=root, IID=0, SL=1]

CLAUSES:

STATEMENTS:
g  A(5) != AR(2)  % on line 55
n  Session[Actor=dummy_agent_1, IID=n5(IID_1), SL=1, A=A(5), B=AR(2)]  % on line 55

STATES:
   Environment[Actor=root, IID=0, SL=2]
   Session[Actor=dummy_agent_1, IID=n5(IID_1), SL=1, A=A(5), B=AR(2)]

CLAUSES:

STATEMENTS:
g  AW(2) != i  % on line 56
n  Session[Actor=dummy_agent_2, IID=n6(IID_2), SL=1, A=AW(2), B=i]  % on line 56

STATES:
   Environment[Actor=root, IID=0, SL=3]
   Session[Actor=dummy_agent_1, IID=n5(IID_1), SL=1, A=A(5), B=AR(2)]
   Session[Actor=dummy_agent_2, IID=n6(IID_2), SL=1, A=AW(2), B=i]

CLAUSES:

STATEMENTS:
n  Initiator[Actor=AW(2), IID=n7(IID_3), SL=1, B=i, Na=dummy_text_1, Nb=dummy_text_2]  % on line 40
n  Responder[Actor=i, IID=n7(IID_4), SL=1, A=AW(2), Na=dummy_text_3, Nb=dummy_text_4]  % on line 41

STATES:
   Environment[Actor=root, IID=0, SL=3]
   Session[Actor=dummy_agent_1, IID=n5(IID_1), SL=1, A=A(5), B=AR(2)]
   Session[Actor=dummy_agent_2, IID=n6(IID_2), SL=3, A=AW(2), B=i]
   Initiator[Actor=AW(2), IID=n7(IID_3), SL=1, B=i, Na=dummy_text_1, Nb=dummy_text_2]
   Responder[Actor=i, IID=n7(IID_4), SL=1, A=AW(2), Na=dummy_text_3, Nb=dummy_text_4]

CLAUSES:

STATEMENTS:
```

```
f  Initiator[n7(IID_3)].Na := n8(Na)  % from fresh on line 16
s  AW(2) -> i : {n8(Na).AW(2)}_pk(i)  % on line 17

STATES:
   Environment[Actor=root, IID=0, SL=3]
   Session[Actor=dummy_agent_1, IID=n5(IID_1), SL=1, A=A(5), B=AR(2)]
   Session[Actor=dummy_agent_2, IID=n6(IID_2), SL=3, A=AW(2), B=i]
   Initiator[Actor=AW(2), IID=n7(IID_3), SL=3, B=i, Na=n8(Na), Nb=dummy_text_2]
   Responder[Actor=i, IID=n7(IID_4), SL=1, A=AW(2), Na=dummy_text_3, Nb=dummy_text_4]

CLAUSES:

STATEMENTS:
n  Initiator[Actor=A(5), IID=n23(IID_3), SL=1, B=AR(2), Na=dummy_text_1, Nb=dummy_text_2]  % on line 40
n  Responder[Actor=AR(2), IID=n23(IID_4), SL=1, A=A(5), Na=dummy_text_3, Nb=dummy_text_4]  % on line 41

STATES:
   Environment[Actor=root, IID=0, SL=3]
   Session[Actor=dummy_agent_1, IID=n5(IID_1), SL=3, A=A(5), B=AR(2)]
   Session[Actor=dummy_agent_2, IID=n6(IID_2), SL=3, A=AW(2), B=i]
   Initiator[Actor=AW(2), IID=n7(IID_3), SL=3, B=i, Na=n8(Na), Nb=dummy_text_2]
   Responder[Actor=i, IID=n7(IID_4), SL=1, A=AW(2), Na=dummy_text_3, Nb=dummy_text_4]
   Initiator[Actor=A(5), IID=n23(IID_3), SL=1, B=AR(2), Na=dummy_text_1, Nb=dummy_text_2]
   Responder[Actor=AR(2), IID=n23(IID_4), SL=1, A=A(5), Na=dummy_text_3, Nb=dummy_text_4]

CLAUSES:

STATEMENTS:
r  ? -> AR(2) : {n8(Na).AW(2)}_pk(AR(2))  % on line 30
m  Responder[n23(IID_4)].A := AW(2)  % from matching on line 30
m  Responder[n23(IID_4)].Na := n8(Na)  % from matching on line 30
f  Responder[n23(IID_4)].Nb := n21(Nb)  % from fresh on line 31
s  AR(2) -> AW(2) : {n8(Na).n21(Nb)}_pk(AW(2))  % on line 32
+  witness(AR(2),AW(2),auth_Initiator_authenticates_Responder,n8(Na))  % on line 32

STATES:
   Environment[Actor=root, IID=0, SL=3]
   Session[Actor=dummy_agent_1, IID=n5(IID_1), SL=3, A=A(5), B=AR(2)]
   Session[Actor=dummy_agent_2, IID=n6(IID_2), SL=3, A=AW(2), B=i]
   Initiator[Actor=AW(2), IID=n7(IID_3), SL=3, B=i, Na=n8(Na), Nb=dummy_text_2]
   Responder[Actor=i, IID=n7(IID_4), SL=1, A=AW(2), Na=dummy_text_3, Nb=dummy_text_4]
   Initiator[Actor=A(5), IID=n23(IID_3), SL=1, B=AR(2), Na=dummy_text_1, Nb=dummy_text_2]
   Responder[Actor=AR(2), IID=n23(IID_4), SL=4, A=AW(2), Na=n8(Na), Nb=n21(Nb)]

CLAUSES:

STATEMENTS:
r  i -> AW(2) : {n8(Na).n21(Nb)}_pk(AW(2))  % on line 18
m  Initiator[n7(IID_3)].Nb := n21(Nb)  % from matching on line 18
+  request(AW(2),i,auth_Initiator_authenticates_Responder,n8(Na),n7(IID_3))  % on line 18
+  request(AW(2),i,fresh_Initiator_authenticates_Responder,n8(Na),n7(IID_3))  % on line 18
s  AW(2) -> i : {n21(Nb)}_pk(i)  % on line 20
+  witness(AW(2),i,auth_Responder_authenticates_Initiator,n21(Nb))  % on line 20

STATES:
   Environment[Actor=root, IID=0, SL=3]
   Session[Actor=dummy_agent_1, IID=n5(IID_1), SL=3, A=A(5), B=AR(2)]
   Session[Actor=dummy_agent_2, IID=n6(IID_2), SL=3, A=AW(2), B=i]
   Initiator[Actor=AW(2), IID=n7(IID_3), SL=5, B=i, Na=n8(Na), Nb=n21(Nb)]
   Responder[Actor=i, IID=n7(IID_4), SL=1, A=AW(2), Na=dummy_text_3, Nb=dummy_text_4]
   Initiator[Actor=A(5), IID=n23(IID_3), SL=1, B=AR(2), Na=dummy_text_1, Nb=dummy_text_2]
   Responder[Actor=AR(2), IID=n23(IID_4), SL=4, A=AW(2), Na=n8(Na), Nb=n21(Nb)]

CLAUSES:

```

```
STATEMENTS:
r  AW(2) -> AR(2) : {n21(Nb)}_pk(AR(2))  % on line 33
+  request(AR(2),AW(2),auth_Responder_authenticates_Initiator,n21(Nb),n23(IID_4))  % on line 33
+  request(AR(2),AW(2),fresh_Responder_authenticates_Initiator,n21(Nb),n23(IID_4))  % on line 33
a  Responder[n23(IID_4)].Na := n8(Na)  % from assignment on line 34

STATES:
   Environment[Actor=root, IID=0, SL=3]
   Session[Actor=dummy_agent_1, IID=n5(IID_1), SL=3, A=A(5), B=AR(2)]
   Session[Actor=dummy_agent_2, IID=n6(IID_2), SL=3, A=AW(2), B=i]
   Initiator[Actor=AW(2), IID=n7(IID_3), SL=5, B=i, Na=n8(Na), Nb=n21(Nb)]
   Responder[Actor=i, IID=n7(IID_4), SL=1, A=AW(2), Na=dummy_text_3, Nb=dummy_text_4]
   Initiator[Actor=A(5), IID=n23(IID_3), SL=1, B=AR(2), Na=dummy_text_1, Nb=dummy_text_2]
   Responder[Actor=AR(2), IID=n23(IID_4), SL=6, A=AW(2), Na=n8(Na), Nb=n21(Nb)]

CLAUSES:

VIOLATED:
   auth_Responder_authenticates_Initiator[AM=n21(Nb), AR=AR(2), AW=AW(2), IID=n23(IID_4)]  % on line 50

EXPLICIT:
        dishonest(i)
        iknows(0)
        iknows(ak)
        iknows(atag)
        iknows(ck)
        iknows({n8(Na).n21(Nb)}_pk(AW(2)))
        iknows({n21(Nb)}_pk(i))
        iknows({n8(Na).AW(2)}_pk(i))
        iknows(ctag)
        iknows(hash)
        iknows(i)
        iknows(inv(ak(i)))
        iknows(inv(ck(i)))
        iknows(inv(pk(i)))
        iknows(inv(succ))
        iknows(n21(Nb))
        iknows(n8(Na))
        iknows(pk)
        iknows(root)
        iknows(stag)
        iknows(succ)
        iknows(C)
        iknows(D)
        iknows(A)
        iknows(B)
        request(AR(2),AW(2),auth_Responder_authenticates_Initiator,n21(Nb),n23(IID_4))
        request(AR(2),AW(2),fresh_Responder_authenticates_Initiator,n21(Nb),n23(IID_4))
        request(AW(2),i,auth_Initiator_authenticates_Responder,n8(Na),n7(IID_3))
        request(AW(2),i,fresh_Initiator_authenticates_Responder,n8(Na),n7(IID_3))
        state_Environment(root,0,1)
        state_Responder(i,n7(IID_4),1,AW(2),dummy_text_3,dummy_text_4)
        witness(AR(2),AW(2),auth_Initiator_authenticates_Responder,n8(Na))
        witness(AW(2),i,auth_Responder_authenticates_Initiator,n21(Nb))

MESSAGES:
 AW(2)        ->   <i>          : {n8(Na).AW(2)}_pk(i)
<?>           ->    AR(2)       : {n8(Na).AW(2)}_pk(AR(2))
 AR(2)        ->   <AW(2)>      : {n8(Na).n21(Nb)}_pk(AW(2))
<i>           ->    AW(2)       : {n8(Na).n21(Nb)}_pk(AW(2))
 AW(2)        ->   <i>          : {n21(Nb)}_pk(i)
<AW(2)>       ->    AR(2)       : {n21(Nb)}_pk(AR(2))
```

As you can see, CL-AtSe was able to find an attack path to violate some security goal.

This is indicated at the very beginning under `SUMMARY: ATTACK_FOUND`. After some statistics about itself and the present model checking run, the tool lists a protocol of the steps it performed while checking the model. The description of each step consists of a triple of `STATES`, `CLAUSES`, and `STATEMENTS`.

The `STATES` section lists all existing entity instances and their current parameter values. The `CLAUSES` section provides information about Horn clauses, which however are not used in our NSPK example, therefore this section is empty[12]. The `STATEMENTS` section lists the statements that were executed in that step together with the parameter values, referencing the line numbers in the ASLan++ file.

Once the model checker detects a security goal violation, it stops checking and under the heading `VIOLATED` prints out the violated security goal. This is followed by a kind of ”core dump”, which lists all facts that are known at the time of the goal violation. Here we have only `EXPLICIT` facts that have been introduced by declaring them explicitly, for instance by instantiation of an entity or generation of a nonce. As in our NSPK example we don't use Horn clauses, we have no `IMPLICIT` deduced facts.

The most interesting part of it however is the `MESSAGES` section with its compact message-sequence-chart-like attack representation at the bottom of this listing:

```
 AW(2)         ->   <i>          : {n8(Na).AW(2)}_pk(i)
<?>            ->    AR(2)       : {n8(Na).AW(2)}_pk(AR(2))
 AR(2)         ->   <AW(2)>      : {n8(Na).n21(Nb)}_pk(AW(2))
<i>            ->    AW(2)       : {n8(Na).n21(Nb)}_pk(AW(2))
 AW(2)         ->   <i>          : {n21(Nb)}_pk(i)
<AW(2)>        ->    AR(2)       : {n21(Nb)}_pk(AR(2))
```

This attack trace demonstrates that our NSPK protocol is vulnerable to a man-in-the-middle-attack, which was initially detected by Lowe [13]. If we input this attack trace to “Web Sequence Diagrams”, we obtain Figure 7. As described below, it would be logically more adequate to place the node `<AW(2)>` just right of `<?>`, but apparently the tool does not do this in order to avoid arrows overlapping with vertical lines.

`AW(2)` is an honest instance of `Initiator` and `AR(2)` is an honest instance of `Responder`. The angle brackets `<...>` around a party denote the intended or assumed communication partner, of whose identity, however, the honest communication peer cannot be sure. In other words, it may be actually the intruder acting in the name of the party given. Here `<i>` is the intruder in the name of himself, `<?>` is (for the Responder at least) an hitherto unknown party, and `<AW(2)>` is the party that the Responder assumes to be the Initiator, which however he cannot be sure of until proper authentication – and here in fact it is the intruder in the name of the Initiator.

Lowe's attack is sketched in a slightly more compact form in Figure 8. The different intruder forms `<i>`, `<?>`, and `<AW(2)>` from Figure 7 all are subsumed under a single `i`.

If `A` initiates a session with the agent `i` (i.e. the Initiator wants to communicate with the intruder, without knowing that the intruder is actually a dishonest party), `i` can relay

[12] For an example containing Horn clauses please refer to § 2.1.7 below.

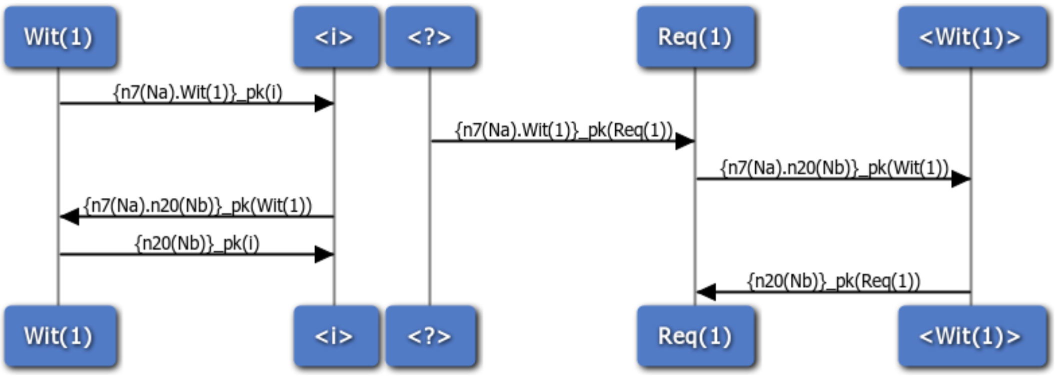


Figure 7: An AVANTSSAR Model Checker found a Vulnerability in the NSPK Protocol.

the data received from A to B and vice versa (with some re-encryption) and thus convince B that he is communicating with A. That is, the Responder is tricked into thinking that he is talking with the Initiator, while in fact he is communicating with the intruder.

The input of "Web Sequence Diagrams" corresponding to the compact attack trace in Figure 8 is:

```
A->i: {Na, A}_Ki
i->B: {Na, A}_Kb
B->i: {Na, Nb}_Ka
i->A: {Na, Nb}_Ka
A->i: {Nb}_Ki
i->B: {Nb}_Kb
```

This attack only succeeds if two NSPK sessions are conducted in parallel and interleaved, as sketched in Figure 8. The intruder i re-encrypts A's nonce and name received from A with the public key of B. He then (under the heading of <?> in Figure 7) relays this re-encrypted message to B.

B reads that A wants to communicate with him, generates his fresh nonce Nb and sends the response encrypted with A's public key to the party he assumes to be A (under the heading of <AW(2)> in Figure 7), which however (as we know) actually is i. Now, i cannot read Bs response, as B has encrypted his reply to A with A's public key.

However, i can relay B's response to A, who obligingly decrypts Nb for i. Then i can re-encrypt B's nonce and then (under the heading of <AW(2)> in Figure 7) forward it to B. B now erroneously believes that he has successfully authenticated A.

The attack can continue this far only because the secrecy goal templates in the Session entity have been intentionally commented out. If you re-activate the secrecy goals, the model checkers find a violation of a security goal already when A discloses Nb to i. Then the secrecy goal secret_Nb is violated, as only A and B may know of it. You are encouraged to try this out.

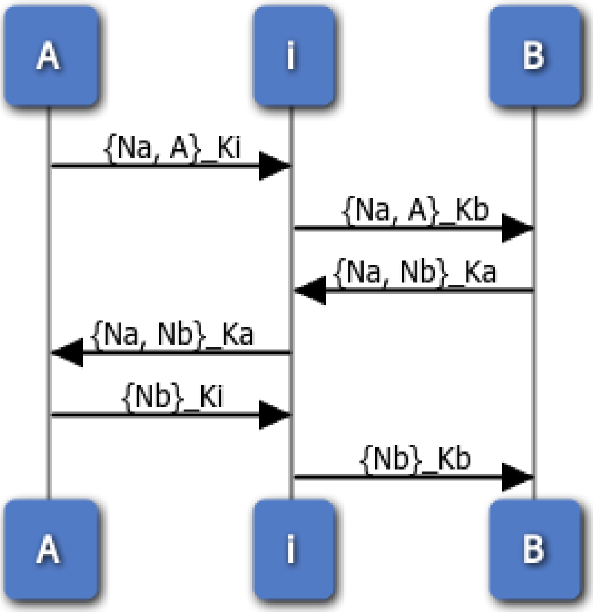


Figure 8: The Lowe Attack on the NSPK Protocol.

Correcting the ASLan++ Model.

What went wrong? When i relays B's response to A, A cannot discern that this response came from B originally. She thinks that she is communicating with i and i only. Thus, we have to extend the protocol with some indication for A that the response comes from B. B puts into his response not only A's nonce and his own, but also his name, before encrypting it for A, i.e. {Na.Nb.B}_Ka, as shown in Figure 9. Now i again relays this message, which he cannot decrypt, to A. However, on checking the name in the message, now A will notice that the nonce is from B and not from i, as she thought. Then she can react accordingly.

The input of "Web Sequence Diagrams" corresponding to Figure 9 is:

```
A -> B: {Na.A}_Kb
B -> A: {Na.Nb.B}_Ka
A -> B: {Nb}_Kb
```

We can make this simple correction in our ASLan++ model of the NSPK protocol. Within the definition of Responder, we add as additional parameter the Responder's name to the message he sends to the Initiator, and on the Initiator's side, we receive and check this name against the expected Responder identity.

```
entity Initiator(Actor, B: agent) {
   ...
   body {
      ...
      B  ->  Actor: {Initiator_authenticates_Responder:(Na).
                     secret_Nb:(?Nb).B}_pk(Actor);
      ...
```

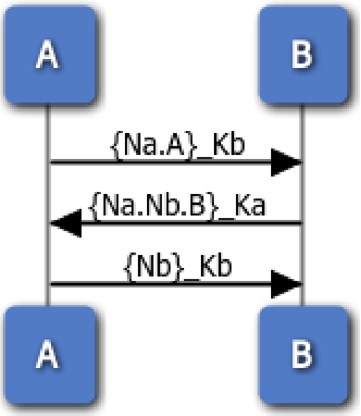


Figure 9: Lowe's Fix for the Vulnerability in the NSPK Protocol.

```
    }
...
}
entity Responder(A, Actor: agent) {
    ...
    body {
        ...
        Actor  ->  A: {Initiator_authenticates_Responder:(Na).
                       Nb.Actor}_pk(A);
        ...
    }
...
}
```

If we now re-translate and re-check the corrected model, no more attack is found.

2.1.7 Clauses, Deductions and Recursion

A frequently required concept is that of computation including recursion. Although in ASLan++ there are no function calls, *clauses*[13] may be used for such (and many other) purposes. This section introduces some basic use of clauses. We specify a simple Public Key Infrastructure (PKI) extension of our NSPK example, and show how to use Horn clauses to do deductions, including recursion, of additional facts from given facts.

On Public Key Infrastructures.

We assume that A and B (and i) have been issued Public Key certificates by a Certificate Authority (CA). A public key certificate is an electronic document that binds a name to a public key in a trustworthy manner. Public key certificates are signed by a trusted certificate

[13] These are extensions of classical Horn clauses; the syntax of the rules resembles Prolog.

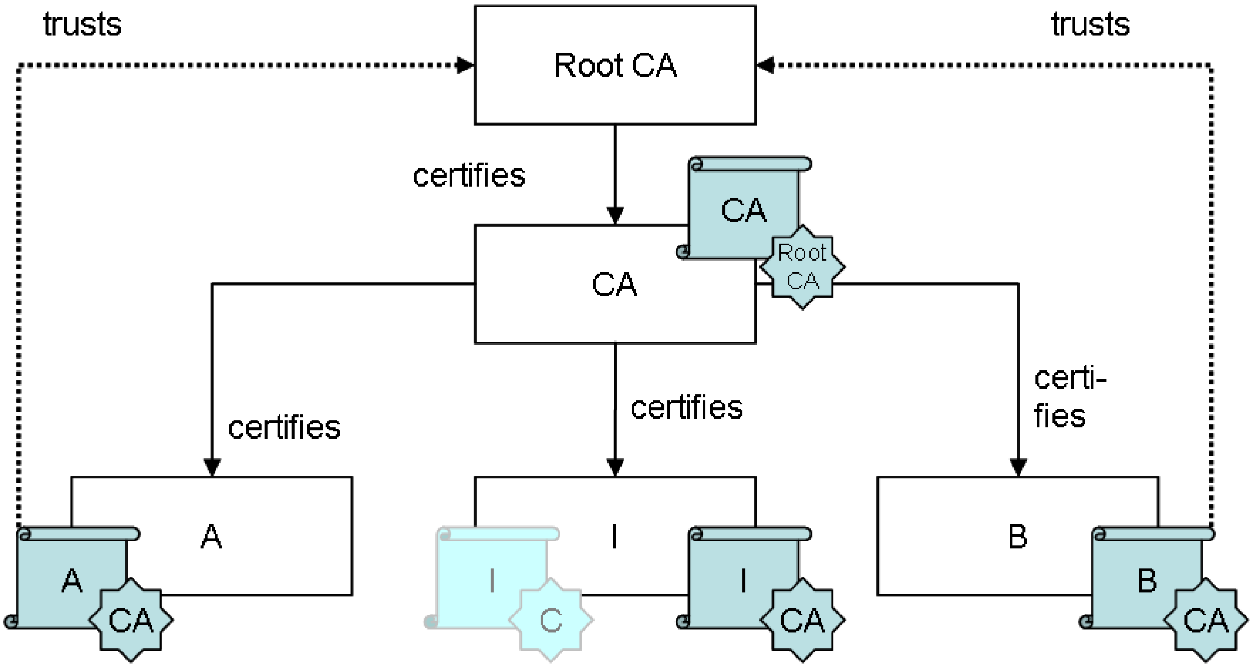


Figure 10: The CA Hierarchy Setup.

authority (CA), after the CA has verified the data of the applicant, e.g., by checking their passport. When validating a public key certificate, one must check:

Validity Range: Normally the "Valid From" and "Valid To" fields in the certificate can be compared to a reference time.

Revocation State: A certificate may be revoked before it expires, e.g., because an employee leaves the company. A check against a blacklist or directory should help here.

Chain of Trust: The certificate must be signed by a trusted party. Especially in large PKI structures trust may be indirect, for instance, a root CA signs the certificate o fa subordinate CA, which in turn signs end user certificates. Directories may help to discover chains of trusts.

Purpose: A certificate is issued for a certain certificate use, e.g., email signing, file encryption, client authentication, certificate signing, etc. The relying party must check the certificate's signed purpose against the intended certificate use.

Modeling certificates.

Certificates can be straightforwardly modeled as a pair of an agent name and a public key, signed by some agent. In general, some contents `M` signed by an agent `A` can be simply written as `{M}_inv(pk(A))` where `inv(pk(A)))` is the private key of `A`. More realistically, the contents `M` are concatenated with their hash value signed by `A`, which can be written as `M.{hash(M)}_inv(pk(A))` where `hash` is a generic hash function defined in the prelude.

In order to reduce redundancies and improve readability, we introduce a macro, `signed(A,M)`, that is expanded automatically to the term just mentioned. For further details about macros please refer to § 3.6.

For further enhancing readability, one can make use of infix expressions for function application, where the first function argument and the token `->` gets prepended to the function symbol, resulting in our case to `A->signed(M)`. In this way we end up with the following macro declaration.

```
  macros  % improving readability
    A->signed(M) = M.{hash(M)}_inv(pk(A)); % Agent A signed M
               % = M.sign(inv(pk(A)),hash(M))
```

In a similar fashion, for terms denoting certificates for agents `A` and their predefined public key, we define a slight abbreviation, also written in infix style:

```
    C->cert(A) = C->signed(C.A.pk(A));
       % Abstract certificate without details,
       % like validity period, cert. purpose, etc.
```

Validating Certificates.

To implement the validation of the chain of trust from a given certificate to a trusted root certificate authority, we introduce the following clauses within the `Environment` entity.

```
1  symbols
2  ...
3   trusted_agent(agent): fact;
4   trusted_pk(agent): fact;
5   issued(message): fact;
6  ...
7  clauses
8    trusted_pk_direct(C):
9      trusted_pk(C) :-
10       trusted_agent(C);
11
12   trusted_pk_cert_chain(A, B):
13     trusted_pk(A) :-
14       trusted_pk(B)
15     & issued(B->cert(A))
16     & B!=A;
```

With the clause named `trusted_pk_direct` (line 8) we express that we trust a public key of an entity `C` if `C` has been declared a trusted agent. In our model we will do this for the trusted root certificate authority only. This clause is a recursion termination point.

With the other clause named `trusted_pk_cert_chain` (line 12) we introduce the recursion step. We say that we trust the public key of an entity `A` if we trust the public key of another agent `B`, and if `B` has issued a certificate for `A`, accrediting `A`'s public key, and if it is not a self-signed certificate. A and B are placeholders for any agent, where A usually stands for an end-user, which in our example is an instance of the Initiator or Responder.

To simplify the presentation, we do not model the issuing process of certificates, but just declare them in additional clauses, as shown in the following listing.

```
1      root_ca_is_trusted:             % We accept root_ca as trusted.
2        trusted_agent(root_ca);
3
4      ca_has_certificate:             % Intermediate ca signs user certs.
5        issued(root_ca->cert(ca));    % ca's cert is signed by root_ca.
6
7      others_have_cert(A):            % Initiator's and Responder's
8        issued(ca->cert(A));          % certificate signed by CA.
9
10     i_has_invalid_cert:             % Invalid intruder certificate
11       issued(c->cert(i));           % signed by non-trusted party.
```

The CA signing the certificates of the Initiator and Responder in our example is an intermediate CA, with agent name ca, authorized by a superordinate trusted root CA. A and B both trust the root CA, as depicted in Figure 10.

- We declare the root CA as trusted by using a trivial clause stating trusted_agent(root_ca) (line 2) as fact.
- Further, we declare that the root CA has issued a certificate for the intermediate CA by saying issued(root_ca->cert(ca)) (line 5).
- We also declare (line 8) that the intermediate CA has issued certificates for any A, which includes the intruder i.
- Finally, we let some non-trusted party c also issue a certificate for i (line 11), which however no Initiator nor Responder would accept.

The Initiator (as sketched in below listing in line 4) and the Responder accept a signed message only if the signer has a trustworthy certificate, otherwise they refuse communication.

```
1 entity Initiator (Actor, B: agent) {
2 ...
3    body {
4       if(trusted_pk(B)) {
5           ...
6       }
7    }
8 }
```

A trustworthy certificate has been directly or indirectly signed by a trusted CA. To discover the chain of trust requires recursive verification of certificates until reaching the signature of a trusted CA or coming to the conclusion that no such CA can be found. This is done via deduction by evaluating above clauses against the known facts.

For simplicity, we do not model (and verify) the certificate's validity range nor its revocation state nor its intended purpose.

The complete ASLan++ Model.
The following listing contains the complete ASLan++ model for the adapted NSPK example[14].

```
% NSPK with Public Key Certificates / Horn Clause Use Case
% @clatse(--hc blr)
% @satmc(No)
% @ofmc(No)

specification NSPK_Cert_Cneq_Unsafe
channel_model CCM

entity Environment {

  symbols % global declarations
    root_ca, ca: agent;      % certificate authorities.
    c: agent;                % agent used as fake CA.
    trusted_agent(agent  ): fact;
    trusted_pk   (agent  ): fact;
    issued       (message): fact;

  macros  % improving readability
    A->signed(M) = M.{hash(M)}_inv(pk(A)); % Agent A signed M
               % = sign(inv(pk(A)),M).M

    C->cert(A) = C->signed(C.A.pk(A));
       % Abstract certificate without details,
       % like validity period, purpose, etc.

  clauses  % Horn clauses used to deduce new facts
    trusted_pk_direct(C):         % Direct trust C's pk
      trusted_pk(C) :-            % if C is a trusted
        trusted_agent(C);         % certificate authority.

    trusted_pk_cert_chain(A, B):  % Indirect trust C's pk
      trusted_pk(A) :-            % if we trust B's pk
        trusted_pk(B)             % and know a certificate
      & issued(B->cert(A))        %    where B accredits A.
```

[14] The model checkers can be fine-tuned by various options on the command line, as hinted on in the ASLan++ model. For instance, `@satmc(No)` indicates that SATMC cannot handle this model, and `% @clatse(--hc blr)` means that CL-AtSe should be used with the `--hc blr` option. This is because SATMC and OFMC cannot deal with the inequality given in the second clause and CL-AtSe goes astray on the recursive clause unless it is told to use a breadth-first evaluation strategy for clauses.

```
      & B!=A;    %%% CL-AtSe: no self-signed certs allowed.

    root_ca_is_trusted:           % We accept root_ca as trusted.
      trusted_agent(root_ca);

    ca_has_certificate:           % Intermediate ca signs user certs.
      issued(root_ca->cert(ca));  % ca's cert is signed by root_ca.

    others_have_cert(A):          % Initiator's and Responder's
      issued(ca->cert(A));        % certificate signed by CA.
        % :- A!=i & A!=c;

    i_has_invalid_cert:           % Invalid intruder certificate
      issued(c->cert(i));         % signed by non-trusted party.

  entity Session (A, B: agent) {

    entity Initiator (Actor, B: agent) {

      symbols
        Na, Nb: text;

      body {
        if(trusted_pk(B)) { % A talks to B if B's public key is trusted
           secret_Na:(Na) := fresh();
           Actor  ->  B: {Na.Actor}_pk(B);
           B  ->  Actor: {Initiator_authenticates_Responder:(Na).
                                                 secret_Nb:(?Nb)}_pk(Actor);
           Actor  ->  B: {Responder_authenticates_Initiator:(Nb)}_pk(B);
        }
      }
    }

    entity Responder (A, Actor: agent) {

      symbols
        Na, Nb: text;

      body {
        ? -> Actor: {?Na.?A}_pk(Actor);      % Responder learns A here!
        if (trusted_pk(A) & A!=Actor) { % B talks to A if A's pk is trusted
          secret_Nb:(Nb) := fresh();
          Actor  ->  A: {Initiator_authenticates_Responder:(Na).Nb}_pk(A);
          A  ->  Actor: {Responder_authenticates_Initiator:(Nb)}_pk(Actor);
          secret_Na:(Na) := Na;    % Goal can only be given here,
                         % because A is not authenticated before!
        }
      }
    }

    body { % of Session
      new Initiator(A,B);
      new Responder(A,B);
    }

    goals
      secret_Na:(_) {A,B};                             % Okay.
      secret_Nb:(_) {A,B};
% Attack found !
     %%% May comment out the above secrecy goals such
     %%% that the attack is printed on authentication.
      Initiator_authenticates_Responder:(_) B *->> A;  % Okay.
      Responder_authenticates_Initiator:(_) A *->> B;
% Attack found !

  }

  body { % of Environment
  % Two sessions needed for Lowe's attack.
    any A B. Session(A,B) where A!=B & A!=root_ca & B!=root_ca
                                     & A!=     ca & B!=     ca;
    any C D. Session(C,D) where C!=D & C!=root_ca & D!=root_ca
                                     & C!=     ca & D!=     ca;
  }
}
```

On translating the above listing you will receive a warning on Horn clauses being an experimental feature. You can safely ignore this warning and check the model with CL-AtSe. DvO: Yes. It's an euphemism – from experience I see no chance that SATMC and OFMC will ever support it.

Analysis results.
If all honest agents and the intruder `i` are granted a valid certificate, the well-known Lowe attack [13] succeeds. It is instructive to have a look at the `CLAUSES` sections of the attack output. For instance, when the Initiator checks the certificate of her responder `B`, CL-AtSe reports that it made use of the following `CLAUSES`:

```
others_have_cert[A=AR(2)]
trusted_pk_cert_chain[A=AR(2), B=ca]
ca_has_certificate
trusted_pk_cert_chain[A=ca, B=root_ca]
trusted_pk_direct[C=root_ca]
```

If the intruder i does not have a certificate that can be traced to a trusted root CA, then the Initiator refuses to communicate with i and no attack can take place. You may try this out by extending the rule

```
others_have_cert(A):               % Initiator's and Responder's
  issued(ca->cert(A));             % certificate signed by CA.
```

to

```
others_have_cert(A):               % Initiator's and Responder's
  issued(ca->cert(A)) :-           % certificate signed by CA,
    A!=i & A!=c;                   % with intruder and c excluded.
```

such that the intruder does not directly obtain a valid certificate, and neither c, which would otherwise lead to a valid chain of trust for the fake certificate c->cert(i).

For a further example, in § 2.7, we provide a simple RBAC model utilizing Horn Clauses.

2.2 Modeling hints

When specifying a protocol/system in ASLan++, you may wonder whether your model is correct. Of course, there may be conceptual mistakes. You for instance may have misunderstood the protocol/system properties and the resulting model therefore would not adequately represent reality. However here in this chapter we do not want to address conceptual correctness of individual models.

Rather we now discuss 'programming' errors and inefficiencies. These can render any conceptually flawless model useless. We point out a few basic strategies of how to avoid the most common mistakes.

2.2.1 Debugging wrt. executability problems

The following advice should be every model checker's intrinsic mantra. Prior to checking your model for attacks,

ALWAYS ensure executability of your model first!

If a model checker responds with NO_ATTACK_FOUND, this does not necessarily mean that the protocol/system described is secure. Before you may put trust in such a result, you must ensure as far as possible, that there are no problems in the model or in the tools. Until then, indeed you must be very suspicious if no attack has been found during a model check. Why?

Very often, no attack is reported by a model checker, because in the model not all statements can be executed. Non-executability of a model in this context means, that not even under optimal conditions (i.e. all participants are behaving as they should and no intruder is messing up things), the protocol can be executed until the intended end, if any, of the system run.

The most frequent reason for non-executability of a model is that a message sent by some party does not match with the message pattern expected by the intended receiver. Then the intended receiver, who is waiting for a different message, blocks execution and the rest of the protocol may never be executed.

Incompatibility may result because of missing sub-messages. Also wrongly set parentheses such as `(M1.M2).M2` vs. `M1.M2.M2`, which is implicitly parenthesized as `M1.(M2.M2)`, etc. can easily cause non-executability and are easily overlooked by human readers. Even more subtle mistakes are type mismatches, e.g. because the sender sends an agent name and the receiver expects a public key instead.

To help detecting such problems, the output format (OF) provides an `UNUSED:` section which shows any ASLan rules that were never used by the model checker during its analysis[15] So far only CL-AtSe and SATMC support this.

SATMC additionally offers the `--exec` (or equivalently, `-e`) option for executability checking, showing which rules can be executed and which not. OFMC offers the same options, which should – after some bug fixes – offer a similar functionality. In addition, OFMC offers the `--trace` (or equivalently, `-t`) option for interactively exploring all possible execution traces.

Independently of such a special support by the model checkers, a very helpful technique to spot and debug executability problems consists of manually adding checkpoints into the model like for instance

```
assert reached_breakpoint4711: false; % for exec testing!
```

at various places, in particular at the intended end of a protocol session. Then with the help of the model checkers and typically for a single session of your protocol/system you can observe if the artificial 'attack' provoked by the `false` formula actually is found. If so, the breakpoint can be commented out and other reachability properties or actual security goals can be checked.

This technique also helps you to detect for instance "overused" statement lines caused by CL-AtSe not being supplied with a large enough `--nb <n>` value. The `--nb` option instructs CL-AtSe, how often it should execute code multiple times, for instance in `while` loops. If you require a loop to be executed a lot of times, you might need to experiment a bit, before finding the optimal `--nb` value. If you choose an insufficient one, when exceeding the maximum number of iterations, CL-AtSe just quietly exits at that point and reports, that no attack was found until then. This however you will notice by an appropriately placed "`assert finished: false;`". The equivalent applies to situations, in which you try to remove an element from a set that this element is not contained in. Or if you attempt to retract a fact that has not been introduced before.

So, again, before actually starting to check your model for real attacks, we strongly advise you first to make sure and double-check that all parts of the modeled protocol/system can in fact be reached and executed.

[15] Please note that individual unreachable statements not necessarily are a reason to worry about. For instance, with parameterized models, if you instantiate your session with one parameter value, sections of your model only applicable to other parameter values of course may not be executed. Further, if you use an `if`-branch without any `else`-branch in ASLan++, in ASLan there will implicitly be added an empty `else`-branch, which may show up as a single non-executable statement pointing to the `if`-branch of your ASLan++ file.

2.2.2 Detecting uninitialized variables

A problem easily overlooked is that variables might be used without being initialized first. This typically leads to attacks not found or spurious (wrong) attacks being reported.

For example, the following entity declaration contains four such mistakes.

```
entity Test {  % implicit Actor parameter not initialized!
  symbols
    U: message;
    S: message set;
  body {
    send(i,U);       % sends an uninitialized msg variable!
    S->add(i);       % uses  an uninitialized set variable!
    S := {};         % initializes the set variable
    S->add(U);       % adds  an uninitialized msg variable!
  }
}
```

Uninitialized variables after translation in the resulting ASLan model have pseudo values such as `dummy_message` and `dummy_nat`, depending on the type of the variable. When an attack trace is printed by the model checkers, one may have a close look at it to spot such values in term positions where variables are used (e.g. in a term that is sent or assigned to some other variable).

In the given example, an attack trace would include

```
STATEMENTS: of Test[n3(IID)]
s  dummy_agent -> i : dummy_message
+  dummy_set_message->contains(i)
a  S := set_1(n3(IID))
+  set_1(n3(IID))->contains(dummy_message)
```

clearly showing the four `dummy_`... values resulting from using `Actor`, `U` (two times), and `S` while not yet being initialized.

So far there is no direct tool support for finding such mistakes. Yet one may extend an ASLan++ specification with a couple of auxiliary declarations and goals at the level of the outermost entity, as in the listing given next.

```
%% This idea does not work as expected, because for each uninitialized variab
%% the translation produces a different dummy constant. So the output is SAFE

specification Uninitialized_Unsafe
channel_model CCM

entity Environment
{
  symbols
    uninitialized_msg(message    ): fact;
    uninitialized_set(message set): fact;

  symbols
    UA: agent;
    UM: message;
    UK: symmetric_key;
    UN: nat;
    UP: public_key;
    US: private_key;
    UT: text;

    SA: agent set;
    SM: message set;
    SK: symmetric_key set;
    SN: nat set;
    SP: public_key set;
    SS: private_key set;
    ST: text set;

  entity Test {  % implicit Actor parameter not initialized!
    symbols
      U: message;
      S: message set;
    body {
      send(i,U);      % sends an uninitialized msg variable!
      S->add(i);      % uses  an uninitialized set variable!
      S := {};        % initializes the set variable
      S->add(U);      % adds  an uninitialized msg variable!
    }
  }

  body
  {
    uninitialized_msg(UA); uninitialized_set(SA);
    uninitialized_msg(UM); uninitialized_set(SM);
    uninitialized_msg(UK); uninitialized_set(SK);
    uninitialized_msg(UN); uninitialized_set(SN);
    uninitialized_msg(UP); uninitialized_set(SP);
    uninitialized_msg(US); uninitialized_set(SS);
    uninitialized_msg(UT); uninitialized_set(ST);

    new Test;
  }
  goals
    no_uninitialized_msg: [](forall U  .
                  !(     iknows(U) & uninitialized_msg(U)));
    no_uninitialized_set: [](forall U S.
                 !(S->contains(U) & uninitialized_set(S)));
    no_uninitialized_mem: [](forall U S.
                 !(S->contains(U) & uninitialized_msg(U)));
}
```

The goals `no_uninitialized_msg`, `no_uninitialized_mem` and `no_uninitialized_set` will raise an attack when an uninitialized message is sent, when an uninitialized set reference is used, or when there is an uninitialized element in a set.

Indeed, when running the example, the above mistakes are found. For instance, SATMC, which can report multiple goal violations at once, prints

```
DETAILS
  ATTACK_FOUND

⋮

GOAL
  no_uninitialized_msg(dummy_message)
  no_uninitialized_mem(set_1(fnat(iid_1,0,0)),dummy_message)
  no_uninitialized_set(dummy_set_message,i)
```

This technique could be extended further to check e.g. for uninitialized sender and receiver variables in transmission statements.

2.2.3 Improving efficiency

Models may take quite a long time to get checked by a model checker. Sometimes the size and complexity of a model may even prevent receiving an answer within reasonable time.

If the protocol and system to be modeled is so very complex and cannot be simplified, sometimes this is unavoidable. Maybe it can in some cases be solved by using a better computer. Or it would require improving the translator and/or the model checkers. Or according to current facilities it cannot be fixed at all.

However modelers may follow some guidelines to improve the efficiency of their models, making it easier for the translator to generate more efficient[16] ASLan code and for the model checkers to perform faster checks.

- Try to make the 'code' in the bodies of entities as 'linear' as possible. In particular, try to avoid `if` and `while` constructs as far as possible. If in a conditional statement you only need a positive branch like

  ```
  if(condition) {
    statements
  }
  ```

[16] This will allow for more for better 'lumping' of ASLan transitions, that is, more efficient ASLan code generation resulting in reduced search space.

then this can be coded more efficiently as

```
select {
  on(condition): {
    statements
  }
}
```

- Use types as narrow as possible. For example, prefer type `text` over `message` whenever there is no need for the modeling to decompose into sub-messages, and use compound types indicating the internal structure of messages as far as possible. This will reduce search time in particular for SATMC.
- Avoid complex/recursive Horn clauses, complex LTL goals, etc. For instance, in many cases the `<->` ("once") LTL operator can be avoided by using stable predicates, i.e. facts that are never retracted.

2.3 Set operations

In ASLan++ we can use sets of elements of a certain type. These types may be basic, like `text` or `nat`. They also may be composite, like we will see in § 2.9 on TLS, where we use sets to model a session database. Elements of sets must be of a uniform type and may not be of type `fact`. Please note that sets are not multi-sets, thus elements may not be present in duplicates within a set. The following examples define sets.

```
NatResultSet   : nat set;
TextSet        : text set;
ServerSessions : agent*text*nat set;
```

Set variables like `NatResultSet`, `TextSet`, and `ServerSessions` are all references, like pointers in C. You can imagine, they point to a location where the elements in the set are stored. For instance, if you copy with ”`TextSetCopy := TextSet;`”, then both pointers point to the same location and thus they are aliases of the same set of elements. You do NOT get a cloned set of elements that way. Naturally you also cannot concatenate sets simply with a ”`.`”, like ”`Set3 := Set1.Set2;`”. Just imagine, what it would mean to concatenate two pointers in C.

2.3.1 Basic Commands

For querying and manipulating sets in ASLan++, we can use the following basic facts and statements (commands). For all these operations, the type of variable `E` must be the type of the elements of `MySet`.

Contains: `contains(MySet,E)`, or equivalently `MySet->contains(E)`.
This fact states that the element `E` belongs to `MySet`. If you use a ”?” like a wildcard with the element name, then `MySet->contains(?E)` returns the value

of a single random element, if there is at least one element in the set. If you use composite elements, then you also may partially qualify the query, like in `ServerSessions->contains((C,SessID,?N))`, where client `C` and `SessID` are known and `?N` may have any value. If multiple elements would qualify, this query also returns only a single random element.

IsEmpty: `if(!MySet->contains(?)) {% it is empty ...}`.
There is no dedicated `isEmpty()`-operator. Yet, the mentioned fact allows us to check whether the set is empty. We have learned already that the statement `MySet->contains(?E)` returns a single random value, as long as there is at least one element in the set. Here we use just a "?", since we are not interested in any specific element value.

Add: `add(MySet,E)`, or equivalently `MySet->add(E)`.
This statement adds the element `E` to `MySet`. If the element is in the set already then the addition has no effect.

Remove: `remove(MySet,E)`, or equivalently `MySet->remove(E)`.
This statement removes the element `E` from `MySet`. Here also you can use a "?" like a wildcard with the element name. If multiple elements would qualify, this statement still removes just one single random element. Please be aware, if you try to remove a specific element from a set, ensure first that this element really is contained in the set. Otherwise execution simply block at this point. The same applies for trying to remove a wildcard element from an empty set.

These basic commands allow us to query and manipulate sets.

2.3.2 ForAll and Copy

If we want to copy sets and/or perform some actions on all elements of a set, things get more complex. As we have seen, a `set` is merely a pointer to a location where the elements in the set are stored, rather than the set itself. Moreover, we cannot access the elements in the set sequentially, since set elements are not ordered, so there is no `next()`-operator for sets. Consequently, we must proceed somewhat more circumstantial to provide a set copy functionality and a ForAll operator.

ForAll. If we want to perform some action for all elements in a set and do not need to retain the set as such, we can proceed as follows.

```
while (MySet->contains(?E)) {
   % perform the action for E
   MySet->remove(E);
}
```

Here we iterate though all elements. Each element processed is simply removed from the set, until the set is empty and all elements have been processed.

Copy. If, however, we need to retain the original set, we must copy the set, either before or while iterating through it and performing the desired actions on the elements. To copy all elements of a set into another set and still retain the original set of elements requires some organization, as shown next.

```
MySetCopy   := {}; % point to a fresh location
MySetBackup := {}; % point to a fresh location

% transfer elements to both MySetCopy and MySetBackup
while (MySet->contains(?E)) {
   MySetBackup->add(E);
   MySetCopy->add(E);
   MySet->remove(E);
}
% move back the elements from MySetBackup to MySet
while (MySetBackup->contains(?E)) {
   MySet->add(E);
   MySetBackup->remove(E);
}
```

We first prepare two new locations for the elements, one for the required copy, and one for a backup of the elements of the original set. Then we iterate through the original set, copying all elements in both the copy and the backup location. Note that for improved efficiency, we combine performing actions on the elements and making the necessary backup for set preservation.

Finally, in the second loop, we restore the elements from the backup location to the original location. For the given example, we could also do this more efficiently by:

```
   MySet  := MySetBackup;
   MySetBackup := {} % in case keeping a copy is undesirable
```

To make this work with SATMC, you need to set no special options. For OFMC, as usual, please use the options `--classic --untyped`. When using CL-AtSe, you must state the option `--nb <n>`. Select `<n>` large enough to cover the maximal number of iterations of `while` loops in the model. For instance, `<n>` resolves to at least 5 if you plan to use an example set with 5 elements. This allows CL-AtSe to execute the verb|while| loop 5 times.

2.3.3 Union and Intersection

Union. If we want to compute the union of two sets, *not* preserving the original sets and ending up with the union of both sets, we may simply move the contents to one set into the other set.

```
MySetUnionResult := MySet1;    % Result points to MySet1

% MySetUnionResult := MySet1 + MySet2;
while (MySet2->contains(?E)) {
   MySetUnionResult->add(E);
   MySet2->remove(E);
}
```

After this, MySetUnionResult points to the same location as MySet1 where also the elements of MySet2 have been added. MySet2 is now empty.

If we need to preserve the original sets, we must use the copy strategy introduced in the previous paragraph. This is left to the reader as simple exercise.

Intersection. Analogously, if we want to compute the intersection of two sets, and do *not* need to preserve the original sets, we can proceed as follows.

```
MySetIntersectionResult := {};  % Provide a fresh location

% MySetIntersectionResult := MySet1 ^ MySet2;
while (MySet1->contains(?E)) {
   if (MySet2->contains(E)) {
      MySetIntersectionResult->add(E);
   }
   MySet1->remove(E);
}
```

After this, MySetIntersectionResult contains the intersection of MySet1 and MySet2. The latter set has been left unchanged, whereas MySet1 is now empty.

Again, if we need to preserve the original MySet1, we must use the copy strategy.

Minus. If we want to subtract all elements of one set from another set, *not* needing to preserve the original set, we might do this as follows.

```
MySetMinusResult := {};  % Provide a fresh location

% MySetMinusResult := MySet1 - MySet2;
while (MySet1->contains(?E)) {
   if (!MySet2->contains(E)) {
      MySetMinusResult->add(E);
   }
   MySet1->remove(E);
}
```

After this, MySetMinusResult contains those elements of MySet1 not part of MySet2. The former set is now empty, whereas the latter set is still in its original state.

If we need to preserve the original MySet1, we must take measures to copy it, either before or on the fly, as mentioned in § 2.3.2.

2.3.4 IsSubset and IsEqual

IsSubset. If we want to check whether one set is a subset of another set, we can subtract the other set from that set and check whether the result is the empty set. How to subtract one set from another we have just described in § 2.3.3. How to test a set for emptiness is detailed above in § 2.3.1.

However, we should take care to preserve the original sets carefully, as nobody will expect a query operation like this to modify the sets in question. Including optimizations like making a backup while checking for the subset-property, this might look as follows.

```
...
nonpublic isSubset: fact;  % locally declared auxiliary fact
...
MySetBackup := {};  % provide a fresh location
isSubset;           % initially assume the subset property

% MySet1->isSubsetOf(MySet2);
while (MySet1->contains(?E) & isSubset) {
   if (!MySet2->contains(E)) {
         retract isSubset;
   }
   if (isSubset) {
         MySetBackup->add(E);
         MySet1->remove(E);
   }
}
% move back the elements from NatSet1Backup to NatSet1
while (MySetBackup->contains(?E)) {
   MySet1->add(E);
   MySetBackup->remove(E);
}
```

After this, if the fact isSubset still holds, we have checked that MySet1 is a subset of MySet2. By restoring all elements from MySetBackup to the original location in MySet1, we have preserved the original set and thus the validity of the tested set property.

As said already, this works. However it is pretty inefficient and does in no way utilize the full capabilities of the model checkers.

How could we proceed more efficiently? By properly challenging the model checkers and letting them do the hard work they have been desigend to do. We can query for the sub-set property more efficiently utilizing the fact that the model checkers will evaluate any possible combination on their own anyway, when properly challenged. For this we state the following.

```
...
nonpublic nonSubset: fact;   % initially this fact is not set
...
% MySet1->isSubsetOf(MySet2);
if(MySet1->contains(?E) & !MySet2->contains(?E)) {
      nonSubset;
}
assert subset_test: !nonSubset;
```

This also works. And it is much less work for us, and less work for the model checker as well. You will notice, that this evaluates much faster. Black magic? No. How does it work?

If you challenge a model checker by stating a security goal, the model checker will look for ways to violate that goal. This is similar to what it does here. We say, activate the nonSubset fact if (and only if) there is some E that indicates that the subset property does not hold. The model checker will now try to find a way to violate the subset property by trying all possible values of elements from the two sets. It will take each element from MySet1 and look at all elements from MySet2 to find whether that element from MySet1 is in MySet2 as well. Just as we wanted to. Smart? Yes.

IsEqual. Two sets S1 and S2 are equal if and only if S1 is a subset of S2 and S2 is a subset of S1. Checking equality can therefore be reduced to computing — in both directions — the subset relation, which can be done as described in the previous paragraph. We first do it the efficient way.

```
...
nonpublic nonMutualSubset: fact;   % initially fact is not set
...
% MySet1->isEqualTo(MySet2);
if ((MySet1->contains(?E) & !MySet2->contains(?E)) |
   ((MySet2->contains(?E) & !MySet1->contains(?E)))) {
      nonMutualSubset;
}
assert equal_test: !nonMutualSubset;
```

Again, the model checker, being challenged to violate the property of the mutual subset property, will search for ways to find elements in MySet1 or MySet2 to violate this property. That was easy. And sufficient.

If we want to do it the hard way, querying for equality manually, but nevertheless including some optimzations, we can do as follows.

```
...
isSubset, isEqual: fact;
...

MySetIntersection := {};     % Provide a fresh location
isSubset;                    % initially assume the subset property
% isEqual; retract isEqual;  % initially assume sets are not equal

% MySet1->isSubsetOf(MySet2);
while (MySet1->contains(?E) & isSubset) {
   if (!MySet2->contains(E)) {
      retract isSubset;
   }
   if (isSubset) {
      MySetIntersection->add(E);
      MySet1->remove(E);
      MySet2->remove(E);
   }
}
if (isSubset & !MySet2->contains(?)) { % MySet2 is Empty
   isEqual;
}
% move back the elements from MySetIntersection to MySet1
% and MySet2
while (MySetIntersection->contains(?E)) {
   MySet1->add(E);
   MySet2->add(E);
   MySetIntersection->remove(E);
}
```

At the end of executing this, if the fact `isEqual` holds, we have checked that `MySet1` and `MySet2` are equal. `MySet1` has been shown a subset of `MySet2` and, after removing every element contained in `MySet1` from `MySet2`, `MySet2` is empty. This confirms that also `MySet2` is a subset of `MySet1`. Which is what we needed to show.

By restoring all elements from `MySetIntersection` to the original locations in `MySet1` and `MySet2`, we have preserved the original sets.

2.3.5 Exercises

For the advanced reader we propose the following exercises.

- Count how many elements there are in a set (using the `succ()`-operator).
- Implement indexed sets (using composite elements and the `succ()`-operator).

Have fun!

2.4 Shared Data

Sometimes it is useful to model that more than one entity has access to the same data.

As long as the data is constant, this can be easily achieved by simply declaring the values as constants (characterized by lower-case symbols) in an entity surrounding the others, for instance in the root entity, which, as we know, is usually called `Environment`. Then the constant data can be used without further ado in this and all enclosed (sub-)entities.

Things become a bit more involved when data may change over time and thus needs to be stored in a variable. Here we show three methods of dealing with this: global variables, facts, and shared sets used e.g. as global databases.

2.4.1 Global variables

A standard method is similar to the style using global constants mentioned before in the introduction to this section. The shared variable is declared in an outer entity and may be used in any inner one. See the following example.

```
% @verbatim(Modifiable shared variable by inheritance)
% @clatse(--nb 2)
% @ofmc(NO)

specification SharedVariable1_Unsafe
channel_model CCM

entity Environment {

  entity Session (A, B: agent) {

    symbols
      SharedVar: nat;

    entity Alice (Actor, B: agent) {

      body {
        SharedVar := succ(SharedVar);
            % this refers two times to the inherited
            % shared variable - does not work for OFMC!
        Actor *->  B: i; % just an authentic signal to B
      }
    }

    entity Bob (A,Actor: agent) {

      body {
        A *-> Actor: i;
        assert test_SharedVar_still_1: SharedVar = 1;
             % should report violation because now
             % "shared(SharedVarId, succ(1))" holds
      }
    }

    body {
      SharedVar := 1;
      new Alice(A,B);
      new Bob  (A,B);
    }
  }
  body
    any A B. Session(A,B);
}
```

Here both Alice and Bob have access to the variable named `SharedVar`. When Alice changes it, Bob can observe this, such that the given assertion

```
  assert test_SharedVar_still_1: SharedVar = 1;
```

does not hold anymore.

When translating this example, with

```
java -jar aslanpp-connector.jar
                SharedVariable1_Unsafe.aslan++
                -o SharedVariable1_Unsafe.aslan
```

the ASLan++ connector warns:

```
WARNING at line 20, column 26: Variable "SharedVar" defined in the scope of "Session"
   is used in the scope of "Alice". This may cause problems to some of the backends.
WARNING at line 20, column 18: Variable "SharedVar" defined in the scope of "Session"
   is used in the scope of "Alice". This may cause problems to some of the backends.
WARNING: Variable "SharedVar" defined in the scope of "Session"
   is used in the scope of "Bob". This may cause problems to some of the backends.
WARNING at line 31, column 39: Variable "SharedVar" defined in the scope of "Session"
   is used in the scope of "Bob". This may cause problems to some of the backends.
```

And indeed, OFMC terminates abruptly stating

```
ofmc-core: multiple waiting terms in lhs of a rule:
[child(x201.x202,i),state_Alice(x203,1,x204,x202),
          state_Session(x205,x206,x207,x208,x209,x201)]
```

SATMC (without the need for special options) confirms the expected violation of the assertion. CL-AtSe also does this, but needs to be called with the `--nb 2` option to cope with variable inheritance without executability problems.

Consequently, this method of global shared variables may only be used if its side effects regarding the selection of a model checker are acceptable.

2.4.2 Implementation via facts

An alternative method to inherited variables takes advantage of facts being global. Therefore facts may be (ab-)used for indirect communication. To this end, we can globally declare a symbol

```
  shared(protocol_id, message): fact;
```

that may be used to hold the value of a shared variable (of type `message` in this case). Here we refer to the variable by some value of type `protocol_id` representing the identity of the variable, for instance

```
  SharedVarId: protocol_id;
```

Initializing this variable to some value `x` amounts to introducing the fact

```
  shared(SharedVarId, x)
```

The variable may be read by referring to the fact as in

```
  shared(SharedVarId, <pattern>)
```

which has the same effect as testing the variable for equality (where the `<pattern>` may be e.g. `?X` such that the local variable verb|X| is bound to the current value of the shared variable.

The effect of assigning to the variable is achieved by retracting the previous fact for this variable and then introducing a new one, e.g.

```
  retract shared(SharedVarId, ?);
  shared(SharedVarId, <new value>);
```

See the following example:

```
% @verbatim(Modifiable shared variable implemented via facts)

specification SharedFact_Unsafe
channel_model CCM

entity Environment {

  entity Session (A, B: agent) {

    symbols
      shared(protocol_id, nat): fact;
      SharedVarId: protocol_id;

    entity Alice (Actor, B: agent, SharedVarId: protocol_id) {
    % Alice and Bob does not inherit SharedVarId, but
    % get a copy of the reference to SharedVar as parameter

      symbols
        ValueOfSharedVar: nat;
      body {
      % read and remove the old value and set a new value:
        select {
          on(shared(SharedVarId, ?ValueOfSharedVar)): {
             retract (shared(SharedVarId, ValueOfSharedVar));
             shared(SharedVarId, succ(ValueOfSharedVar));
          }
        }
        Actor *->  B: i;                       % handover to B
      }
    }

    entity Bob (A,Actor: agent, SharedVarId: protocol_id) {

      body {
        A *-> Actor: i;                      % handover from A
        assert test_SharedVar_still_1: shared(SharedVarId, 1);
                                             % reads the current value
                                 % should report violation because now
                                % "shared(SharedVarId, succ(1))" holds
      }
    }

    body {
      SharedVarId := fresh();
      shared(SharedVarId, 1); % sets the initial value
      new Alice(A,B,SharedVarId);
      new Bob  (A,B,SharedVarId);
    }
  }

  body
    any A B. Session(A,B) where A!=B & !dishonest(A);
}
```

This model uses all the mechanisms just described to achieve essentially the same effect as the previous example.[17]

This variant is more involved than the previous one, but has the advantage that all three model checkers, SATMC, OFMC and CL-AtSe can deal with it without problems.

2.4.3 Shared databases

A very typical way to implement shared databases is via sets, which by the semantics of ASLan++ are implemented using references to (global) facts. This technique will be used, among others, in § 2.5.1 and § 2.9.

To implement a database, we first have to declare it. With the definition `SharedDB: agent*text*nat set` we specify that each entry in our shared database will hold an agent, a nonce and a counter.

To work with our database, we need some basic commands. With `SharedDB->add ((Actor,Nonce,0))` we can add an entry to the database.

[17] Regarding semantics, there is one caveat: when changing the value of the shared variable, there might be race conditions if between the retraction of the old value and the introduction of the new value there is interleaving with any other entity instances that use the same shared variable. Yet due to default optimization options of the translator, the retraction and the introduction are typically executed in a single atomic transition.

With SharedDB->remove((B,?,1)) we can remove an entry. With SharedDB->contains((B,?,1)) we can query the database if an entry of a certain form is present (or not: !SharedDB->contains((B,?,1))). For more details on sets and how to use them, see the ASLan++ syntax definition in § 3 and the set examples in § 2.3.

To make our database shared between the entities Alice and Bob, we declare it in the entity immediately enclosing these two entities, which is the Session entity. Session passes the reference to the shared database to Alice and Bob as parameter. For the complete example see the following listing.

```
% @verbatim(Modifiable contents of shared database)
% @ofmc(NO)

specification SharedDB_Unsafe
channel_model CCM

entity Environment {

  entity Session(A,B: agent) {

    symbols
      SharedDB: agent*text*nat set;   % Contents global for
                                            % Alice and Bob

    entity Alice(Actor,B: agent,
                            SharedDB: agent*text*nat set) {

      symbols
        Nonce: text;
      body {
        Nonce:=fresh();
        SharedDB->add((Actor,Nonce,0));   % add entry to DB
        Actor *->  B: Nonce;                % handover to B
        B *->  Actor: ?;                  % handover from B
        if (SharedDB->contains((B,?,1))) {
           SharedDB->remove((B,?,1));    % remove B from DB
        }
        Nonce:=fresh();
        SharedDB->add((Actor,Nonce,2));   % add entry to DB
        Actor *->  B: Nonce;                % handover to B
      }
    }

    entity Bob(A,Actor: agent,
                            SharedDB: agent*text*nat set) {

      symbols
```

```
        Nonce: text;
      body {
        A *-> Actor: ?;                   % handover from A
        Nonce:=fresh();
        SharedDB->add((Actor,Nonce,1));   % add entry to DB
        Actor *->  A: Nonce;                % handover to A
        A *-> Actor: ?;                   % handover from A

        assert test_sharedDB_B:
                       SharedDB->contains((Actor,Nonce,1));
                      % should report violation because now
               % A has removed B's entry from the shared DB
      }
    }

    body {
      SharedDB := {};
      new Alice(A,B,SharedDB);
      new Bob  (A,B,SharedDB);
    }
  }

  body
    any A B. Session(A,B) where A!=B &
                             !dishonest(A) & !dishonest(B);
}
```

In this example, the `Session` provides an empty shared database to Alice and Bob, a reference to which they obtain as parameter. The contents of the shared database are common to both Alice and Bob (call-by-reference semantics). Alice puts an entry with her name into the database and hands over to Bob. Bob puts an entry with his name into the database and hands back to Alice. Alice checks,whether Bob has entered an entry with his name into the database, and, in case he did, removes that entry. Then she hands over to Bob. Bob checks whether his entry is still in the database and complains if this is not the case any longer.

In fact, once you translate and check[18] this model with

```
java -jar aslanpp-connector.jar
        SharedDB_Unsafe.aslan++ -o SharedDB_Unsafe.aslan
cl-atse SharedDB_Unsafe.aslan >SharedDB_Unsafe.atk
java -jar aslanpp-connector.jar SharedDB_Unsafe.aslan -ar
                                    SharedDB_Unsafe.atk
```

the model checker reports the expected ”attack”. Please note, that here the model checker uses `A(1)` to refer to the instance `A` of Alice, and `Actor(0)` to refer to the instance `B` of Bob.

[18] For more information on connector and model checker options please see Appendix B.

```
INPUT   : SharedDB_Unsafe.aslan
SUMMARY : ATTACK_FOUND
...
STATES:
   ...
   Alice[Actor=A(1), IID=n2(IID), SL=5, B=Actor(0),
         Nonce=n3(Nonce),
         SharedDB={Actor(0).n7(Nonce).1, A(1).n3(Nonce).0}]
   Bob[Actor=Actor(0), IID=n2(IID), SL=7, A=A(1),
         Nonce=n7(Nonce),
         SharedDB={Actor(0).n7(Nonce).1, A(1).n3(Nonce).0}]
CLAUSES:
STATEMENTS:
g  set_1(n1(IID))->contains(Actor(0).n7(Nonce).1) % line 26
-  set_1(n1(IID))->contains(Actor(0).n7(Nonce).1) % line 27
f  Alice[n2(IID)].Nonce := n5(Nonce)     % fresh on line 29
+  set_1(n1(IID))->contains(A(1).n5(Nonce).2)  % on line 30
s  A(1) *-> Actor(0) : n5(Nonce)               % on line 31

STATES:
   ...
   Alice[Actor=A(1), IID=n2(IID), SL=10, B=Actor(0),
             Nonce=n5(Nonce),
             SharedDB={A(1).n3(Nonce).0, A(1).n5(Nonce).2}]
   Bob[Actor=Actor(0), IID=n2(IID), SL=7, A=A(1),
             Nonce=n7(Nonce),
             SharedDB={A(1).n3(Nonce).0, A(1).n5(Nonce).2}]
CLAUSES:

VIOLATED:
   test_sharedDB_B[Actor=Actor(0), IID=n2(IID),
   Nonce=n7(Nonce), SL=7, SharedDB=
        {A(1).n3(Nonce).0, A(1).n5(Nonce).2}]  % on line 47
...
MESSAGES:
A(1)       *->   <Actor(0)> : n3(Nonce)
Actor(0)   *->   <A(1)>     : n7(Nonce)
A(1)       *->   <Actor(0)> : n5(Nonce)
```

You can see nicely that both Alice and Bob see the same contents of SharedDB, and that manipulations of either of them are immediately visible to the other party.

Synchronization of database access is the responsibility of the modeler.

2.5 Set communication

Since in ASLan++ one cannot send or receive sets directly, we focus in this section on the different approaches to communicate sets between entities.

2.5.1 Shared set variables and synchronization

A simple way of communicating sets among entities in ASLan++ is to declare the sets as shared/global variables. This resembles the shared memory communication model in distributed computing. The burden of synchronizing access to the shared sets is however left to the modeler. We explain this method using a small example. A more detailed example can be found in the ASLan++ specification of the Public Bidding scene [7, §4.1].

Suppose Alice and Bob wish to communicate a set. The set is created by Bob, and then Bob asks Alice to modify the set. Alice then notifies Bob that she is done with her changes, before Bob accesses the global set again.

```
specification SetsGlobal
channel_model CCM

entity Environment
{
  symbols
    sync1, sync2 : message;
    a, b, e1, e2 : agent;
    SetGlobal : agent set;

  entity Alice(Actor, B: agent) {
    body {
      B *-> Actor: sync1;

      if(SetGlobal->contains(e1))
         SetGlobal->remove(e1);

      Actor *-> B: sync2;
    }
  }

  entity Bob(A, Actor: agent) {
    body {
      SetGlobal := {e1, e2};    % Possible assignment method
      % SetGlobal->add(e1); % Alternative assignment method
      % SetGlobal->add(e2); % Alternative assignment method

      Actor *-> A : sync1;
      A *-> Actor: sync2;
      assert no_alarm: !(SetGlobal->contains(e1));
    }
  }

  body {
    SetGlobal := {};                % Initialize global set
    new Alice(a,b);
    new Bob  (a,b);
  }
}
```

Initially, e1 belongs to SetGlobal. Bob asks Alice, by sending the synchronization message sync1, to apply her changes to the set. Alice assigns the empty set to SetGlobal. Then, Alice notifies Bob, by sending message sync2, that he can now access the shared set. Indeed, at this moment, SetGlobal does not contain e1 anymore. Hence, the assertion no_alarm is never violated and therefore no attack has to be reported.

You do not need any special options for the model checkers. Please note that when translating this model, there will be warnings regarding the use of global variables. You may ignore them at this point. Currently only CL-AtSe and SATMC support global sets; OFMC cannot be used for verifying specifications which use global sets.

The synchronization mechanism, as it is always the case with accessing shared objects, is crucial for the correct working of the entities. The AVANTSSAR Platform can indeed help the modelers and designers to debug their synchronization mechanism. In the example given above, for instance, using authenticated channels (*->) to communicate synchronization messages is essential for the correctness of the specification. In particular, if any of sync1 or sync2 is sent over insecure channels, the attacker can fake them, hence subverting the goal no_alarm.

- If sync1 is sent over an insecure channel, then the attacker can fake this message, hence enabling Alice to use SetGlobal while it still holds the empty set, that is to before Bob inserts {e1} in this set. Recall that actions of Alice and Bob are interleaved in the semantics of ASLan++; see § A.3. Then, Bob and Alice would synchronize on sync2, while SetGlobal meanwhile contains e1. Consequently, the no_alarm assertion would be violated in the state of Bob. You are encouraged to try this out. The attack has been reproduced using CL-AtSe.

- If sync2 is sent over an insecure channel, then the attacker can fake this message, hence causing Bob to check the content of SetGlobal before Alice assigns the empty set to this set. The fact that Bob and Alice would synchronize on sync1 obviously enables the no_alarm assertion to be violated in Bob's state. You may want to try this. The attack has been reproduced using CL-AtSe.

If you dislike translator warnings and want to avoid problems with the model checkers, you can also use the following version of our example.

```
specification SetsGlobal2
```

```
channel_model CCM

entity Environment
{
  symbols
    sync1, sync2 : message;
    a, b, e1, e2 : agent;
    SetGlobal : agent set;

  entity Alice(Actor, B: agent, SetGlobalA: agent set) {
    body {
      B *-> Actor: sync1;

      if(SetGlobalA->contains(e1))
         SetGlobalA->remove(e1);

      Actor *-> B: sync2;
    }
  }

  entity Bob(A, Actor: agent, SetGlobalB: agent set) {
    body {
      % SetGlobalB := {e1, e2};      % May NOT be used here !
      SetGlobalB->add(e1); % Only possible assignment method
      SetGlobalB->add(e2); % Only possible assignment method

      Actor *-> A : sync1;
      A *-> Actor: sync2;

      assert no_alarm: !(SetGlobalB->contains(e1));
    }
  }

  body {
    SetGlobal := {};                 % Initialize global set
    new Alice(a,b, SetGlobal);
    new Bob  (a,b, SetGlobal);
  }
}
```

Here we use the same method as we already have used for the shared database in § 2.4.3. In fact, it is an equivalent example. What are the differences here compared to the previous example?

In the "shared DB"-style example, we pass the reference to the global set as parameter to both Alice and Bob. They may both use this reference to query, add and remove elements to and from the global set.

Please be aware of what here of course you may not do. You may not re-assign a different location to the local parameter. This for instance Bob would do by stating `SetGlobalB := {e1, e2}`. Just imagine, what would happen, if in a C function you get passed a parameter containing a pointer, and then plainly ignore the pointer by assigning a new pointer to the variable. In the previous example, however, assigning a new set location to a global variable does not cause a problem, as the changes are visible to all partners.

2.5.2 References

We can communicate sets among agents in ASLan++ by passing references to the sets. This method should however be used with caution because of aliasing. For instance, if a reference to a set `S` which is stored locally at A is passed to B, then B can directly read and manipulate `S` from the moment he receives the reference. In particular, if A adds elements to `S` after sending the reference to `S` to B, the added elements will be visible to B. Similarly, B can directly add (or remove) elements to `S`, even though `S` is locally stored at A. The modeler can think of the set passed by reference as if it resides at a global level, hence being accessible to both A and B. Obviously, if the receiver B is dishonest, the content of the shared set `S` will be readily available to the attacker. We remark that A can always make a local copy of `S`, and pass the reference to the copy to B, so that B would not be able to directly manipulate or read the content of `S`.

A simple example of how the reference method of set communication works, may look as follows.

```
specification Sets_Sent_As_Reference
channel_model CCM

entity Environment
{
  symbols
    a, b : agent;
    e1, e2, signal: message;
    setref (message set): message;

  entity Alice(Actor, B : agent) {
    symbols
      Set1: message set;
    body {
      Set1 := {e1,e2};
      Actor *->* B: setref(Set1);
      B *-> Actor: signal;
      assert no_alarm: !(Set1->contains(e2));
    }
  }

  entity Bob(A, Actor : agent) {
    symbols
      Set2: message set;
    body {
      A *->* Actor : setref(?Set2);
      if(Set2->contains(e2))
         Set2->remove(e2);
      Actor *-> A: signal;
    }
  }

  body {
    new Alice(a, b);
    new Bob  (a, b);
  }
}
```

Note that in this specification, Alice sends the set `Set1` to Bob by passing the reference to the set, namely `setref(Set1)`. After receiving the reference, Bob removes the element `e2` from the set. This operation is visible to Alice. That is, the assertion `no_alarm` within Alice is not violated: `e2` does not belong to `Set1` anymore when Alice checks it.

It is worth mentioning that in order to prevent the attacker from manipulating the set, Alice sends the reference to the set to Bob both confidentially and authentic: `Actor *->* B: setref(Set1)`. It must be confidential because, if the attacker can read the reference, the attacker may read and manipulate the set. However it also must be authentically from Alice, as otherwise an attacker can substitute Alice's reference by his own and trick Bob into using the attacker's set instead. This attack you may provoke, if you send the reference without ensuring authenticity (removing the `*` left to the `->` in both Alice and Bob). Please try it!

As usual with global variables, the modeler needs to explicitly deal with synchronizing the access to sets shared using references. If the modeler intends that Alice creates a set and that Bob should access after the (reference to the) set has been passed to Bob, then the modeler should devise a proper synchronization mechanism. Here, before manipulating the set any further, Alice should wait for an acknowledgement from Bob, stating that he has finished accessing the set. To this end, in the above example we use of the constant message `signal`. Note that the goal `no_alarm` could be violated if the `signal` was not sent over an authentic channel.

2.5.3 Concatenated elements

An further approach for communicating a set between agents is to send a copy of the set by concatenating all its elements. This is comparable to marshalling and un-marshalling parameters for remote procedure calls. This approach is less efficient than the ones introduced above, but allows for loose coupling of the agents without sharing memory of any kind, thus not incurring any synchronization problems.

The sender first transforms the set to be sent into a complex message containing a concatenation of the elements of the set. That is, in order to send a set `S = {A1,..,An}`, we send the message `SMsg = A1.A2...An`, where `"."` denotes the concatenation operator.

In the following example, Alice communicates the set `NatSet := {1,2,3,4}` to Bob.

```
specification Sets_Concatenation
channel_model CCM

entity Environment {

  symbols
    a,b: agent;
    nil: message;
    box(message): message;

entity Alice(Actor, B: agent) {

  symbols
    NatSet, NatSetBackup : nat set;
    E : nat;
    NatSetMessage : message;

  body {
    NatSet        := {1,2,3,4};
    NatSetBackup  := {};          % point to a fresh location
    NatSetMessage := nil;         % empty

    % preserve elements in NatSetBackup
    while (NatSet->contains(?E)) {
      NatSetMessage := E.box(NatSetMessage);
      NatSetBackup->add(E);
      NatSet->remove(E);
    }
    % move back the elements from NatSetBackup to NatSet
    while (NatSetBackup->contains(?E)) {
      NatSet->add(E);
      NatSetBackup->remove(E);
    }
    Actor *->*B: NatSetMessage;
  }
}

entity Bob(Actor, A: agent) {

  symbols
    BNatSet : nat set;
    BE : nat;
    BMessage, Rest : message;

  body {
    A *->* Actor: ?BMessage;
    BNatSet := {}; % point to a fresh location

    while ((BMessage != nil)) {
      select {
        on(BMessage = ?BE.box(?Rest)): {
          BNatSet->add(BE);
          BMessage := Rest;
        }
      }
    }
    assert finished: false;
  }
}

  body { % of Environment
    new Alice(a,b);
    new Bob(b,a);
  }
}
```

Alice forms a concatenation of all elements of `NatSet` into `NatSetMessage`. We use a `box()`-bracket to aid Bob in precise un-marshalling. We preserve a backup of `NatSet` in `NatSetBackup`. In the end, `NatSetMessage` consists of a concatenation of all the elements in `NatSet`, which might look like `3.box(2.box(4.box(1.box(nil))))` (which reminds us that `contains()` non-deterministically returns any element). We restore the original `NatSet` from `NatSetBackup`. Now Alice ships `NatSetMessage` to Bob.

Bob has to revert the procedure to retrieve the set from the message containing the concatenation of the set's elements. In Bob's specification, `BMessage` denotes the message received by Bob. It contains a `box()`ed concatenation of elements of type `nat`. This message Bob processes iteratively: we retrieve its elements `BE` one by one, and add them to `BNatSet`. Every added element `BE` is split off `BMessage`.

The `box()`-bracket aids Bob in precise un-marshalling. Why is this needed? Let's suppose we would not used the `box()` operator and instead transmit the message `SetMessage=M1.M2....Mn` directly, where each `Mi` is an element of type `message` from `MySet`, which is a `set` of type `message`. Then, on the receiving side, a variable `M` of type message can match `Mi`, but also `(Mi....Mj)`. The receiver cannot determine where to split.

We can translate and check the above example with the given four-element set, for instance using Cl-AtSe, supplying the (`--nb 4`) option:

```
java -jar aslanpp-connector.jar sets_concatenation.aslan++
                             -o sets_concatenation.aslan
cl-atse --nb 4 -f sets_concatenation.aslan
```

```
                                    >sets_concatenation.atk
java -jar aslanpp-connector.jar sets_concatenation.aslan -ar
                                     sets_concatenation.atk
```

2.6 Time-out

In this section, we give an example of specifying a simple timeout mechanism in ASLan++.

A common use of timeout is to enable an alternative action for an entity. For instance, Alice sends a message to Bob, and expects to receive an acknowledgement from Bob saying that he has received the message. If the acknowledgement message does not arrive “in time”, Alice may timeout and do something else, for instance raise an alarm. We abstract away the actual amount of time the entity Alice would wait for the acknowledgement in this case, and model the timeout simply as a non-deterministic event.

After sending her message to Bob, Alice starts an imaginary “timer” (which we do not model explicitly). The timer may go off at any moment after it has been initiated, on which Alice introduces the fact `timeout`. Meanwhile Alice may receive the acknowledgement message from Bob, or she may timeout if the timer goes off before such an acknowledgement arrives.

How do we model non-deterministic events in ASLan++? The language reference defines in § 3.10:

> ”The select statement ... checks the guards given, blocking as long as no guard is fulfilled, then nondeterministically chooses any one of the fulfilled guards and executes the corresponding statement block.”

Thus using `select`-statements allows us to model non-deterministic events. This is reflected in the following ASLan++ specification.

```
specification TimeOut
channel_model CCM

entity Environment {

  symbols
    alice,bob : agent;
    ack (message): message;
    ackrcvd: fact;
    alarm: fact;

  entity Alice(Actor, B: agent) {

    symbols
      M : message;
      timeout: fact;

    body {
      M := fresh();
      Actor *->* B: M;
      timeout;
      select {
        on(B *->* Actor : ack(M)): ackrcvd;
        on(timeout): alarm;
      }
    }
  }

  entity Bob(Actor, A: agent) {

    symbols
      Datum : message;

    body {
      A *->* Actor: ?Datum;
      Actor *->* A: ack(Datum);
    }
  }

  body {
    new Alice (alice,bob);
    new Bob   (bob,alice);
  }

  goals
    % no_ackrcvd: [](!ackrcvd);
    no_alarm: [](!alarm);
}
```

In this model, `timeout` is set, and therefore we expect that the fact `alarm` is (at least in some of the executions of the model by a model checker) introduced to the state. For translation and model checking with Cl-AtSe we simply use the following commands and options.

```
java -jar aslanpp-connector.jar timeout.aslan++ -o timeout.aslan
cl-atse -f timeout.aslan > timeout.atk
java -jar aslanpp-connector.jar timeout.aslan -ar timeout.atk
```

The goal `no_alarm` does not hold for this model, as confirmed by CL-AtSe.

You may want to try the following two experiments. First we are curious, if really the acknowledgement from Bob can arrive before the timeout occurs. For this we can check the model against the following goal `no_ackrcvd`.

```
  no_ackrcvd: [](!ackrcvd);
```

To try this, in above listing you comment out the goal no_alarm (in line 47) and comment in the goal no_ackrcvd (in line 46). Then you re-check the model as described. Indeed, the acknowledgement message in this specification may be received by Alice before the timeout. Therefore, the goal no_ackrcvd too is unsatisfied. CL-AtSe confirms that no_ackrcvd can be violated, and provides the corresponding attack trace.

Now we at last want to check, what happens, if Bob never sends any acknowledgement at all. For this, in above listing, we prevent Bob from returning a receipt by commenting out his reply (in line 36). Then the goal no_ackrcvd holds safe in the model. If you re-check this modified model, now Cl-AtSe confirms to you that it found no attack anymore.

2.7 Policies

In § 2.1.7 we have already seen how we can utilize Horn Clauses to model recursion. In this section, we give an example of specifying a simple Role-Based Access Control (RBAC) system, with role hierarchy, by using Horn Clauses. Consider a file server which allows any user to read any file, and any admin inherits the rights of a user, and additionally may write files. That is, the RBAC system consists of two roles, namely user and admin, with admin≥user. This scenario can be modeled using Horn clauses in ASLan++:

```
   can_write(X):        forall Y. can_write(X,Y):- is_admin(X);
   can_read(X):         forall Y. can_read(X,Y):- is_user(X);

   role_hierarchy(X): is_user(X) :- is_admin(X);
```

We specify a file server with the RBAC system described above. Note the use of “explicit” facts (add_user and add_admin) serve to introduce these rights, as opposed to “implicit” facts (is_user and is_admin, see also § 3.5).

```
specification RBAC
channel_model CCM

  entity Server {

   symbols

      Datum : message;
      alice,bob,eve : agent;

      is_user  (agent) : fact;
      is_admin (agent) : fact;
      add_user  (agent) : fact;
      add_admin (agent) : fact;
      can_read  (agent,message) : fact;
      can_write (agent,message) : fact;
      discloses_to (agent, message) : fact;
```

```
    clauses

      write_right(X):
        forall Y. can_write(X,Y):- is_admin(X);

      read_right(X):
        forall Y. can_read(X,Y):- is_user(X);

      role_hierarchy(X):
        is_user(X) :- is_admin(X);

      introduction_user(X):
        is_user(X) :- add_user(X);

      introduction_admin(X):
        is_admin(X) :- add_admin(X);

    body {
      Datum := fresh();
      add_admin(alice);

      if (can_read(alice, Datum) & can_write(alice, Datum)) {
         Datum := fresh();
         if (is_admin(alice)) {
            % add_user(alice);
            add_user(bob);
         }
      }

      if (can_read(bob, Datum)) {
         retract add_admin(alice);
         if (can_write(alice, Datum)) {
            Datum := fresh();
         }
         if (can_read(alice, Datum)) {
            discloses_to(eve, Datum);
         }
      }
    }

    goals
       readers_goal_eve:    [](!discloses_to(eve, Datum));
}
```

As goal we check readers_goal_eve, which states that Eve may never be told Datum. First, Datum is initialized and Alice is granted admin status. If she can both read and write

Datum, she initializes Datum. If she has admin status, she grants Bob user status.

Potentially, Alice could disclose Datum to Eve, because administrators also are users. Therefore Alice also has reading rights. And Alice has declared Bob user. So Bob has reading rights, too, and could tell Eve about Datum.

Now we verify that Bob indeed has reading rights on Datum. Then we retract Alice's admin role. After this she can neither write nor read Datum anymore. Thus she can no longer disclose it to Eve. Cl-AtSe confirms this.

Assume however that Alice, while still being an admin, also creates herself a plain user account (remove the comment in line 43). Then, on retracting the admin role from Alice, she still retains that user account and therefore can still read and disclose Datum to Eve. This is also confirmed by Cl-AtSe.

2.8 Communication in different channel models

We currently support three different channel models.

Cryptographic Channel Model (CCM): communication is realized by passing individual messages via the intruder; security is achieved via applying cryptography to the messages.

Ideal Channel Model (ICM): This is similar to CCM, with the difference being that message protection is achieved by more abstract means.

Abstract Channel Model (ACM): named channels are realized using explicit send and receive events that may be constrained by LTL formulas over the traces.

Thanks to the encoding of channels by cryptographic means that all current backends support, CCM can be used with all existing backends. ICM is currently more of theoretical interest.

While in CCM and ICM the focus is on individual message transfers, in ACM there is a built-in notion of channel identities. ACM is currently supported only by SATMC. It has a slightly different syntax and semantics of channels, which allows for transmitting messages with respect to a channel name, and for specifying channel properties more flexibly via LTL formulas.

For more details about the syntax and semantics of channels, please refer to § 3.8 and § 4.7.6.

To highlight the differences between the CCM/ICM notation and the ACM style, we give a couple of ASLan++ specification examples in both variants.

We first consider a client C who sends a request confidentially to a server S, and S sends a response authentically. This is a typical situation where the server possesses a key certificate (so the client can check that the response really comes from S), but the client does not have a certificate.

In fact, one can achieve stronger channels in such a case, for instance additionally guaranteeing that the response of the S is only visible to the principal who sent the request.

Because this principle is not authenticated, this is called a *pseudonymous secure channel* in CCM/ICM, or a *unilaterally authenticated channel* in ACM. However, we postpone this to the section on modelling TLS, see § 2.9. For the first example we consider a response channel where the message contents are not confidential, but only authentic.

2.8.1 CCM/ICM

For CCM/ICM, authentication of the sender is specified in ASLan++ by using "*->" in the send and receive instructions. Similarly, message transmission confidentiality is expressed by "->*". Thus, if an unauthenticated client submits a confidential request to a server and requires an authenticated reply from that server, this looks as follows.

```
specification CCM_example
channel_model CCM

entity Environment {

  symbols
    client, server : agent;

  entity Session (C, S: agent) {

    entity Client(Actor, S: agent) {
      symbols
        Nc, Ns: text;

      body {
        Nc := fresh();
        Actor ->* S: Actor.Nc;
        S *-> Actor: ?Ns;

        assert finished: false;
      }
    }

    entity Server(Actor: agent) {
      symbols
        C: agent;
        Nc, Ns: text;

      body {
        ? ->* Actor: ?C.?Nc; % Server learns C here!
        Ns := fresh();
        Actor *-> C: Ns;
      }
    }
    body { % of Session
      new Client(C, S);
      new Server(   S);
    }
  }

  body { % of Environment
    new Session(client, server);
  }
}
```

Note that the server learns the name of the client on reception of the first message. This is why in the receive statement `? ->* Actor: ?C.?Nc;` the sender side is marked with the dummy pattern '?' and the agent name of the client `C` is prepended by '?' to indicate that this value is received as part of the message sent by the client.

2.8.2 ACM

For the ACM, we have recently introduced named channels at ASLan++ level. We now present an illustrative example on how these named channels would be used when defining ASLan++ models.

In the ACM, the example from the previous section would look as follows.

```
specification ACM_example
channel_model ACM

entity Environment {
  symbols
    client, server : agent;
    ch_c2s, ch_s2c : channel;

  entity Session (C, S: agent,
                  Ch_C2S, Ch_S2C : channel) {

    entity Client(Actor, S : agent,
                  Ch_C2S, Ch_S2C : channel) {
      symbols
        Nc, Ns: text;

      body {
        Nc := fresh();
        Actor -C_C2S-> S: Actor.Nc;
        S -C_S2C-> Actor: ?Ns;

        assert finished: false;
      }
    }

    entity Server(Actor: agent,
                  Ch_C2S, Ch_S2C: channel) {
      symbols
        C: agent;
        Nc, Ns: text;

      body {
        ? -Ch_C2S-> Actor: ?C.?Nc; % Server learns C here!
        Ns := fresh();
        Actor -Ch_S2C-> C: Ns;
      }
    }

    body { % of Session
      new Client(C, S, Ch_C2S, Ch_S2C);
      new Server(   S, Ch_C2S, Ch_S2C);
    }
  }

  body { % of Environment
    ch_c2s->confidential_to(server);
    ch_s2c->   authentic_on(server);
    new Session(client, server, ch_c2s, ch_s2c);
  }
}
```

Two channels are defined, `ch_c2s` and `ch_s2c`. In the body of the entity `Environment`, the channel `ch_c2s` is declared confidential to the `server`, while the channel `ch_s2c` is specified as authentic on the `server` side. Note that the channel identifiers like `ch_c2s` are now passed as parameters like `Ch_C2S` to the two sub-entities and used there within the send and receive statements.

2.9 Modeling TLS

Like in the examples given so far, unless further action is taken, the security guarantees offered by assumed standard channel properties like confidentiality and authenticity only pertain to individual message transfer. Most of today's online applications, however, do not communicate via unrelated messages. Web applications and web services base themselves on at least HTTP or HTTPS request and response pairs, frequently combined via some notion of a session, forming multiple correlated message exchanges. The channel identifiers of ACM are already a step forward towards relating multiple transmissions.

2.9.1 Security Properties of TLS

Transport Layer Security (TLS) [22] provides stronger guarantees than the channels we have modeled in the previous section. Even if the client is not authenticated, TLS guarantees that the server can ensure that its answer goes only to the party that initiated the connection. This is the typical case where only the server has a public-key certificate. If the client also has a certificate, then we have a full secure channel (both sides authenticated, all messages confidential). Additional properties are that different parallel sessions between the same parties are distinguished, and that the order is preserved in which messages are sent and that message replay is prevented.

One way to model TLS channels would be to directly include the TLS handshake into the specification, but this would not only mean a lack of abstraction and clutter up the specification of a high-level protocol, but for many applications result in a system so complex that it would be infeasible for automated verification. The better way is using our channel notations for CCM/ICM and ACM, but special care must be taken to capture all relevant properties of TLS.

2.9.2 HTTP Request and Response Pairing

Let us first consider one of the most frequent application protocols that are run on TLS, namely HTTP. We model first how to ensure the correct pairing between requests and responses: the client generates a nonce and sends it to the server along with the payload and with its own name such that the server knows whom to respond to. The client then waits for a reply from that server containing the same nonce. This is sketched in the following model fragment. It shows how to model a very basic client-server request-response message exchange.

```
entity Client (Actor, S: agent) {
  ...
  PayloadC := <...>;
  Nc := fresh();             % generates nonce
  Actor -> S: Actor.Nc.PayloadC;
  ...
  S -> Actor: Nc.?PayloadS; % checks nonce
  ...
}

entity Server (Actor: agent) {
  ...
  while (true) {
    select {
      on(? -> Actor: ?C.?Nc.?PayloadC) { % learns C here
        PayloadS := <...>;
        Actor -> C: Nc.PayloadS; % reflects nonce
        ...
      }
      ...
    }
  }
}
```

Since we use insecure channels, with an intruder between client and server, this is completely open to attacks.

2.9.3 Server-Authenticated HTTPS

Next we model a simple stateless server-authenticated HTTPS (HTTP over TLS) message exchange in ASLan++, based on the above HTTP request-response pairing. That is, we want to ensure that each request sent by the client is confidential to the server and that the result received by the client is protected for authenticity, integrity, and freshness and is associated with the right request. For now, we do not address client authentication nor maintain session state or preserve message order.

In CCM, we express message transmission confidentiality using ->*, Similarly, authentication of the sender can be specified by using *->. The only thing we need to implement ourselves is the session disambiguation and the (one-sided) replay protection. Due to the confidentiality and integrity assumptions that we now make, it is sufficient to include a nonce like in the example given before: The client now sends the nonce confidentially with its request, and it is now reflected authentically by the server in its answer and checked by the client. This approach securely associates every response to the correct request, and ensures that the response was not a replay of the response to an older request.

```
entity Client (Actor, S: agent) {
  ...
  PayloadC := <...>;
  Nc := fresh();                            % C generates nonce
  Actor ->* S:          % S can NOT verify C's identity
                Actor.Client_HTTPS_Server:(Nc.PayloadC);
  ...
  S *-> Actor: Server_HTTPS_Client:(Nc.?PayloadS);
                                            % checks nonce
  ...
}

entity Server (C, Actor: agent) {
  ...
  while(true) {
    select {
      on(? ->* Actor:               % S learns C here
            ?C.Client_HTTPS_Server:(?Nc.?PayloadC)):
      {
        PayloadS := <...>;
        Actor *-> C:  % reflects nonce
```

```
            Server_HTTPS_Client:(Nc.PayloadS);
        ...
      }
      ...
    }
  }
}

...
goals
   Client_HTTPS_Server:(_) C  ->* S;
   Server_HTTPS_Client:(_) S *->> C;
   ...
```

The goals specified here reflect many aspects of what this model of the TLS channel provides. The client request is confidential to the server, the response is authentic and replay-protected. However, the fact that different parallel session are distinguished thanks to the nonce is not captured by the goals.

Note that the server accepts any (unauthenticated) client, learning the (alleged) client name C in the request message. Therefore, in the backwards direction confidentiality would not make sense.

2.9.4 Weak Client Authentication

The next aspect we model is a form of weak authentication: that the response can only be received by the client who sent the request. It is common that the client does not possess a certificate and thus cannot authenticate itself directly to the server. However, even not authenticated, TLS ensures that nobody else can read the response.

In CCM/ICM this is modeled using the concept of *pseudonymous secure channels*, the client not authenticated by its real identity (as it would appear in a certificate) but with respect to self-chosen, unauthenticated identifier, called *pseudonym*. This is indicated by square bracket the notation as in [Actor] *->... using the sender's default pseudonym and a sender-side signature. [19] One may also explicitly specify a pseudonym as in [Actor]_[MyNym] *->.... When receiving a message on a pseudonymous secure channels with receiver-side encryption, we write ...->* [Actor]. The server uses the term [CP] to represent the client's pseudonym.

Here is the ICM/CCM specification of this request-response pair:

```
entity Session (C, S: agent) {
  symbols % used for client_link LTL goal
    client_sent(agent,nat,message): fact;
    client_rcvd(agent,nat,message): fact;
```

[19] The digital signature implicitly deployed in CCM to implement "*->" serves as a proof of possession of the private key associated with the client's (alleged) name. Thus the server can check that in subsequent transmissions the client side (whoever this really is) uses the same private key matching the client's name.

```
entity Client (Actor, S: agent) {
  ...
  PayloadC := <...>;
  Nc := fresh();                    % C generates nonce
 [Actor] *->* S:        % S can NOT verify C's identity
                 Actor.(Nc.PayloadC);
  client_sent(Actor,IID,Nc); % used for client_link goal
  ...
  S *->* [Actor]: Nc.?PayloadS;     % checks nonce
  client_rcvd(Actor,IID,Nc); % used for client_link goal
  ...
}

entity Server (C, Actor: agent) {
  ...
  while(true) {
    select {
      on([?CP] *->* Actor:           % S learns C here
            ?C.(?Nc.?PayloadC)):
      {
        PayloadS := <...>;
        Actor *->* [CP]: % reflects nonce to same C only
            Nc.PayloadS;
        ...
      }
      ...
    }
  }
}
...
goals
  ...
  client_link: forall C C' IID IID' M.
               []((client_sent(C ,IID ,M) &
                   client_rcvd(C',IID',M)) =>
                  (C=C' & IID=IID'));
```

Here the `client_link` goal, given as an LTL formula, expresses that the client receiving the response is the same client who sent the request. The property actually achieved is even a bit stronger: the response can not be received by *any* party except the client who crafted the request, i.e. the pseudonymous-security. It is possible to use a similar pseudonymous-secure channel on the goal side [18], but currently such goals are not supported by ASLan++.

In this example, we have just one request-response pair. If we model applications with several transmissions, this kind of channel achieves another interesting property: even though the client is not authenticated, the server can be sure that all messages come from the *same* sender. This also called *sender invariance*. This is also captured by our model of pseudonymous secure channels, because they behave exactly like normal secure channels except that authentication is with respect to a pseudonym.

An almost equivalent model can be written in ACM, where we can take good advantage of *unilaterally authenticated* channels.

```
entity Session (C, S: agent) {
  symbols
    Ch_C2S, Ch_S2C: channel;

  entity Client (Actor, S: agent,
                 Ch_C2S, Ch_S2C: channel) {
    ...
    PayloadC := <...>;
    Actor -Ch_C2S-> S:    % S can NOT verify C's identity
                   Actor.PayloadC;
    ...
    S -Ch_S2C-> Actor: ?PayloadS;      % checks nonce
    ...
  }

  entity Server (C, Actor: agent,
                 Ch_C2S, Ch_S2C: channel) {
    ...
    while(true) {
      select {
        on(? -Ch_C2S-> Actor:          % S learns C here
              ?C.?PayloadC):
        {
          PayloadS := <...>;
          Actor -Ch_S2C-> C: % sends response to same C only
              PayloadS;
          ...
        }
        ...
      }
    }
  }
  body { % of Session
    Ch_C2S := fresh();
    Ch_S2C := fresh();
    % we assume unilaterally authenticated TLS from C to S
    unilateral_conf_auth(Ch_C2S,Ch_S2C,C,S);  ...
    new Client(C, S, Ch_C2S, Ch_S2C);
    new Server(C, S, Ch_C2S, Ch_S2C);
  }
```

```
 goals
   ...
```

Note that for each session of a client and a server, we create a fresh pair channels between the two, which is essential in case more then one session is created.

This ACM variant follows the general pattern of the CCM variant, but with some differences. For modeling the assumed weak client authentication, instead of using a pseudonym we use the abstract channel relation `unilateral_conf_auth(Ch_C2S,Ch_S2C,C,S)`. This in particular implies via `link(Ch_C2S,Ch_S2C)` that the client instance sending on the former channel is the same as the one receiving on the latter channel. Therefore, the `client_link` goal given for the CCM variant is automatically fulfilled. Also the replay protection is automatically fulfilled via the implied `weakly_confidential(Ch_S2C)` property. Thus, for the ACM variant we do not need to construct these properties with the nonce `Nc`, as they are part of the assumptions of the model. For more details on such channel types, see Table 5).

Finally, observe so far our model does not include a mechanism to ensure ordering in which messages are sent (and this is of course not necessary when every session contains only one message in each direction). Modeling such an order preservation is more involved and we come back to this later, in § 2.9.6.

2.9.5 Strong Client Authentication

A very common application of the unilateral TLS channels with weak client authentication that we have just modeled is the use for a secure login, i.e., the so far unauthenticated client sends his or her username and password to the server and thereby authenticates. The idea is that now to use the aforementioned properties: responses go to exactly the client who sent the requests (whoever this is) and sender invariance: over several transmissions of a session, it is ensured that it must be the same client. Together with the authentication that the login provides, it follows that all messages are securely transmitted between client and server. In other words, the TLS channel is effectively upgraded to bilateral authentication.

Note that here and above we do not exclude the case of dishonest clients or servers, e.g. we will include also the case that an attacker is a registered client at some server, with his own username and password. Note also that more complex matters may be attached to a login, such as an access control policy based on user names.

In order to model client authentication and authorization, upon receiving a request from a prospective client for using a certain service, the server should check (an abstraction of) the identity of the client and authorize the service use. To this end, the server needs to have a policy indicating which clients it is willing to serve and which credentials these clients have to supply.

We might model the authentication of the client with a challenge-response protocol: when the client requests a service, the server sends a challenge that the client must correctly react to. We could also use certificates at least for the server party, where (like above in the NSPK variant with certificates, see § 2.1.7) the peer checks if the party's certificate has been signed by some trusted authority.

Here, extending the example given in § 2.8.1, for simplicity we model a very basic and classic form of client authentication and authorization: the client sends along with his login message a password which is checked by the server. Every client that sends a correct password is assumed to be authorized.

We model the passwords as shared secrets between a user and a server; the most convenient way to achieve this is to globally declare a function that maps every pair of agents to their shared secret:

```
nonpublic noninvertible password(agent,agent): text;
```

This function is not public, of course, because the passwords are secret. Every participant simply initially knows all passwords that it is involved in. This allows us to model dishonest users and servers. To model dishonest users, we add the intruder knowledge facts

```
iknows(password(i,S)) % intruder knows its own password
```

for every server S. We model the knowledge of dishonest servers as well:

```
iknows(password(C,i)) % dishonest server knows client passwords
```

for every client C. We do not discuss issues like weak passwords or password-change protocols since we do not want to complicate things in this example.

If we want to model the credential handling more explicitly, we might let the server make use of a credentials database, in form of the server's session database. which may contain also information on the authorization of clients. how to model a database we have already seen in previous examples.

```
specification TLS_client_auth_CCM
channel_model CCM

entity Environment {
  symbols
    login, ok: text;
    nonpublic noninvertible password(agent, agent): text;

  entity Session (C, S: agent) {

    entity Client(Actor, S : agent) {
      symbols
        Payload: text;

      body {
       [Actor]*->> S      : login.Actor.
                            shared_secret:(password(Actor,S));
        S      *->>[Actor]: ok;
        Payload := fresh();
       [Actor]*->> S      : secure_transmission:(Payload);
      }
    }

    entity Server(C, Actor: agent) {
      symbols
        Payload: text;
        CP: public_key; % pseudonym of C
      body {
      [?CP]    *->> Actor: login.?C. % Server learns C here!
                           shared_secret:(password(?C,Actor));
                         % checks password, proceeds only if fine!
        Actor *->>[CP]   : ok;
       [CP]    *->> Actor: secure_transmission:(?Payload);
      }
    }

    body { % of Session
      iknows(password(i, S)); % intruder knows its own password
      iknows(password(C, i)); % dishonest server knows client password
    % we assume unilaterally authenticated TLS from C to S
      new Client(C, S);
      new Server(C, S);
    }
    goals
      shared_secret      :(_) {C, S};
      secure_transmission:(_) C *->> S;
  }

  body { % of Environment
    any C S. Session(C, S) where C != S;
  }
}
```

After the client's identity been verified by the server (assuming that the client did not share its password with anyone else), the unilateral authentication has been effectively transformed into a bilateral one, which is checked by the goal `secure_transmission:(_) C *->> S` for the payload. For the sake of completeness, we also check that the client's password is only shared with the server: `shared_secret:(_) {C, S}`

The idea of the construction using a login over a secure pseudonymous channel (where the client is not a priori authenticated) is that the result is a secure channel: i.e. we can run an application protocol (such as a webmail application) over this channel as if it was a standard (bilaterally authenticated) secure channel. Such a compositionality result is shown in [18], provided that certain conditions between channel and application protocol are fulfilled (so that they do not interfere with each other). [20]

[20] Note that after successful client authentication we could have directly used secure channels instead of pseudonymous channels, replacing e.g. `[Actor]*->> S` by `Actor *->> S` and `[CP] *->> Actor` by

If the focus is the verification of an application protocol (and not of a login mechanism), it is recommended to directly model secure channels (without pseudonyms and passwords transmission), as done below in § 2.9.6.

Note that here we employ one symbolic session between any client `C` and any server `S`, either of which may be the intruder. We just forbid that the client and the server are the same agent (in which case the client would talk to itself, which would not make much sense).

The ACM variant given next follows the design of the CCM variant, with the main difference being that for modeling the assumed weak client authentication, instead of using a pseudonym we use the more abstract channel relation `unilateral_conf_auth(Ch_C2S,Ch_S2,C,S)` already used in the previous example.

```
specification TLS_client_auth_ACM
channel_model ACM

entity Environment {
  symbols
    login, ok: text;
    nonpublic noninvertible password(agent, agent): text;

  entity Session (C, S: agent) {
    symbols
        Ch_C2S, Ch_S2C: channel;

    entity Client(Actor, S: agent,
                  Ch_C2S, Ch_S2C: channel) {
      symbols
        Payload: text;

      body {
        Actor -Ch_C2S-> S   : login.Actor.
                              shared_secret:(password(Actor, S));
        S     -Ch_S2C-> Actor: ok;
        Payload := fresh();
        Actor -Ch_C2S-> S   : secure_transmission:(Payload);
      }
    }

    entity Server(C, Actor: agent,
                  Ch_C2S, Ch_S2C: channel) {
      symbols
        Payload: text;
```

`[C] *->* Actor`, making use of the compositionality theorem for CCM/ICM.Yet we did not perform this transformation in order to avoid confusion between the assumed properties of the channels (namely, unilateral TLS) and the additional property (namely, client-side authentication) obtained by explicit authentication via the client's password.

```
      body {
            ?     -Ch_C2S-> Actor: login.?C. % Server learns C here!
                                   shared_secret:(password(?C, Actor));
                           % checks password, proceeds only if fine!
            Actor -Ch_S2C-> C    : ok;
            C     -Ch_C2S-> Actor: secure_transmission:(?Payload);
      }
    }

    body { % of Session
      iknows(password(i, S)); % intruder knows its own password
      iknows(password(C, i)); % dishonest server knows client password
      Ch_C2S := fresh();
      Ch_S2C := fresh();
    % we assume unilaterally authenticated TLS from C to S
      unilateral_conf_auth(Ch_C2S,Ch_S2C,C,S);
      new Client(C, S, Ch_C2S, Ch_S2C);
      new Server(C, S, Ch_C2S, Ch_S2C);
    }
    goals
      shared_secret      :(_) {C, S};
      secure_transmission:(_) C *->* S;
  }

  body { % of Environment
    any C S. Session(C, S) where C != S;
  }
}
```

After the client's identity has been verified, like in the CCM variant we again state and check the goal client_authentication.

If we did not want to implement the client authentication, we could have started with the stronger channel assumption bilateral_conf_auth(Ch_C2S,Ch_S2C,C,S) for ACM, or non-pseudonymous secure channels for CCM.

2.9.6 Dynamic sessions and message order preservation

If a dynamic session concept is required and if we need to ensure message order preservation as provided by TLS, we may proceed as outlined here.

We model the TLS concept of a session by dynamically creating session identifiers and including these in all messages belonging to the session. To additionally ensure message order preservation, the client uses a sequence number for each message, which the server checks against its session database. We model the sequence numbers in ASLan++ as terms of the form succ(succ(...succ(0)...)) of type nat. An introduction on how to model a shared database has been given in § 2.4.3.

This example contains a single but multi-threaded instance of the server. A client requests a new session, using a fresh session identifier, by starting with message sequence number 0. recognizes a new session by the session identifier being unknown and the message sequence number being 0. The server then enters the new session into its session database and spawns a new thread handling that session. The thread only accepts messages from the same client C that initially contacted the server using the same session identifier and a suitable consecutive message sequence number. Thus a client may have several simultaneous sessions with the server, each being handled by a different thread.

In this model, unlike in the ones before, we make use of the full power of channel assumptions regarding bilaterally authenticated channels and thus can simply assume that the client is authenticated to the server. The server accepts requests for a new session by any client. During a session the server will need to recognize the same client and the corresponding session, such that the reply and all further communication in this session will be with the original client.

To implement this in CCM, we make use of the authenticity assumption specified for all message transmissions by the ‘*’ on left-hand side of the arrow in any statement of the form ‘A *->* B’ where A is the real sender name known to the receiver.

```
% @clatse(--nb 2)

specification TLS_SessionsOrder_CCM
channel_model CCM

entity Environment {

  symbols
    succ(nat): nat; %%% workaround; declaration should not be needed
    server: agent;

  symbols
    ServerSessions: agent*text*nat set; % global for Server

  entity Session(C, S: agent) {

    entity Client (Actor, S: agent) {

      symbols
        SessID: text;
        Counter: nat;
        PayloadC,
        PayloadS: text;

      body {
        SessID := fresh();
      secrecy_goal secret_SessID: Actor, S: SessID;
        Counter := 0; % start session
        Actor *->* S: Actor.SessID.Counter;

        Counter := succ(Counter); %1
        PayloadC := fresh();
        Actor *->*  S: (SessID.Counter). PayloadC;
      channel_goal Client_TLS_Server:
        Actor *->>* S: (SessID.Counter). PayloadC;
        S *->*  Actor: (SessID.Counter).?PayloadS;
      channel_goal Server_TLS_Client:
        S *->>* Actor: (SessID.Counter). PayloadS;

        Counter := succ(Counter); %2
        PayloadC := fresh();
        Actor *->*  S: (SessID.Counter). PayloadC;
      channel_goal Client_TLS_Server:
        Actor *->>* S: (SessID.Counter). PayloadC;
        S *->*  Actor: (SessID.Counter).?PayloadS;
      channel_goal Server_TLS_Client:
        S *->>* Actor: (SessID.Counter). PayloadS;

        assert finished: false; %%% for excutability testing;
        % If the model checker does not find an attack at least
        % here, the model has an executability problem.
        % Once this pseudo-attack has been shown, the assertion
        % can be commented out to check for the other goals.
      }
    }

    body { % of Session
      new Client(C,S);
    }
  }


  entity Server(Actor: agent, Sessions: agent*text*nat set) {

    symbols
      C     : agent;
      SessID: text;

    entity Thread (Actor, C: agent, SessID: text,
                   Sessions: agent*text*nat set) {

      symbols
        N: nat;
        PayloadC,
        PayloadS: text;

      body {
        while(true) {
          select {
            on(C *->>* Actor: (SessID.?N).?PayloadC &
               Sessions->contains((C,SessID,?N))
              ):
            channel_goal Client_TLS_Server:
              C *->>* Actor:  (SessID. N). PayloadC;
            {
              PayloadS := fresh();
              Actor *->>  C:  (SessID. N). PayloadS;
            channel_goal Server_TLS_Client:
              Actor *->>* C:  (SessID. N). PayloadS;
              Sessions->remove((C,SessID,     N ));
              Sessions->add   ((C,SessID,succ(N)));
            }
          }
        }
      }
    }

    body { % of the Server
      while(true) {
        select {
          on(?C *->>* Actor: ?C.?SessID.0 &
                            % Server learns C here!
             !Sessions->contains((?C,?SessID,?))
            ): {
          secrecy_goal secret_SessID: Actor, C: SessID;
            Sessions->add((C,SessID,succ(0)));
            new Thread(Actor,C,SessID,Sessions);
          % The Thread's Actor is the same as for the Server!
          }
        }
      }
    }
  }

  body { % all sessions are with the same server!
    iknows(0); %%% workaround; should not be needed
    ServerSessions := {};
    new Server(server, ServerSessions);
    any C. Session(C, server) where C!=server;
```

```
    any C. Session(C, server) where C!=server;
  }
}
```

To demonstrate the guarantees we obtain by the construction just explained, this model uses the goal Server_TLS_Client for the channel between the server and the client and Client_TLS_Server for the opposite direction. Moreover, there is a new secrecy goal secret_SessID indicating that the session ID is a secret shared between the client and the server.

Due to limitations of the translation of the current ASLan++ channel and secrecy goal syntax, we had to resort to an earlier form of stating such goals that is more cumbersome, but on the other hand more flexible to cope with the nested Thread entity and the global server instance.

The channel goals for each direction involve integrity/authenticity, replay protection, and confidentiality of the session ID, sequence number, and payload. Note that in comparison to the model mentioned in § 2.9.3, there is now additionally authentication of the client to the server and freshness of messages in the same directions, as well as confidentiality in the opposite direction. Since both the clients and the server threads increase the sequence number upon each reception and the sequence number is part of the channel goals, this also checks for the preservation of the message order in both directions. Of course, apart from the replay protection aspect[21], the channel goals are obviously fulfilled by construction of the channels themselves[22].

To implement the session database, we use sets. With the definition Sessions: agent*text*nat set we specify that our session database will hold client, session identifier and current sequence counter for each session. With Sessions->add ((C,SessID,N)) we can add an entry to the database, with Sessions->remove((C,SessID,N)) we can remove an entry, and with Sessions->contains((C,SessID,N)) we can query the database for an entry being present (or not: !Sessions->contains((C,SessID,N))). For more details on sets and how to use them see the ASLan++ syntax definition in § 3 and the set examples in § 2.3.

Note that the last statement in the client's body is used to test the specified protocol is fully executable. The model checker should find an artificial attack on reaching this line. If the model checker does not find this "attack", then the protocol can not be executed as intended. In this case, we need to eliminate the blocking point(s), making sure for instance that all messages sent match the expectations at the receiver's side, etc. For more details on executability, please see § 2.2.1 within this document.

[21] The way to achieve replay protection in CCM is not that straightforward, in particular from the clients to the server threads. Indeed, an earlier version of this model had a bug which lead to a violation of the freshness property, found by CL-AtSe. The problem occurred because multiple instances of our single server each had their own session database. To fix this problem, all server instances now share the same session database.

[22] The distinction between the channels as assumptions and the corresponding communication goals would be more clear if in the model there was a distinction between the implementation of the TLS channels and their use at application level.

If you want the check this model yourself, you may copy the model into a file named TLS.aslan++ and use the following command for translation[23].

```
java -jar aslanpp-connector.jar TLS.aslan++ -o TLS.aslan
```

To check the model, with SATMC you need to set no special options. For OFMC, please use the options --classic --untyped. When you want to use CL-AtSe, you must state the option --nb 2. For instance, to check the model off-line with CL-AtSe, you can use the following commands.

```
cl-atse TLS.aslan --nb 2 >TLS.atk
java -jar aslanpp-connector.jar TLS.aslan -ar TLS.atk
```

Online you select your model checker of your choice, upload TLS.aslan, set the required options, and execute the model.

Using ACM, we can specify explicitly and very naturally that all payload messages among the client and the server are transmitted over the same pair of channels by the following variant of the above model.

The channel pairs are reflected by the functions

```
noninvertible ch_c2s(text): channel; % client to server
noninvertible ch_s2c(text): channel; % server to client
```

that, for any session ID, yield a channel from the client to the server or vice versa, respectively. Using these functions, we can dynamically create for each new session, referred to by a fresh value stored in the variable SessID, a new pair of channel names and the corresponding channels. These channels are assumed to be confidential and authentic in both directions, which we indicate by introducing the following facts:

```
bilateral_conf_auth(ch_c2s(SessID),ch_s2c(SessID),Actor,S);
```

For initiating a new session, we use a unique public channel provided by the server

```
ch_server: channel; % public initiation ch of server
```

where transmissions are confidential to the server and authentic at the server side:

```
ch_server->confidential_to(server);
ch_server->  authentic_on(server);
```

Apart from these declarations, the resulting ASLan++ model is basically the same as before:

```
% @clatse(--nb 2)

specification TLS_SessionsOrder_ACM
channel_model ACM
```

[23] For more information on connector and model checker options please see Appendix B.

```
entity Environment {

  symbols
    succ(nat): nat; %%% workaround; declaration should not be needed
    server: agent;
    ch_server: channel;    % public initiation channel of server
    noninvertible ch_c2s(text): channel; % from client to server
    noninvertible ch_s2c(text): channel; % from server to client

  symbols
    ServerSessions: agent*text*nat set; % global for Server

  entity Session(C, S: agent, ChS: channel) {

    entity Client (Actor, S: agent, ChS: channel) {

      symbols
        SessID: text;
        Counter: nat;
        PayloadC,
        PayloadS: text;

      body {
        SessID := fresh();
      secrecy_goal secret_SessID: Actor, S: SessID;
        bilateral_conf_auth(ch_c2s(SessID),ch_s2c(SessID),Actor,S);
        Counter := 0; % start session
        Actor -ChS-> S: Actor.SessID.Counter;

        Counter := succ(Counter); %1
        PayloadC := fresh();
        Actor -ch_c2s(SessID)-> S: Counter. PayloadC;
      channel_goal Client_TLS_Server:
        Actor *->>*             S: Counter. PayloadC;
        S -ch_s2c(SessID)-> Actor: Counter.?PayloadS;
      channel_goal Server_TLS_Client:
        S *->>*             Actor: Counter. PayloadS;

        Counter := succ(Counter); %2
        PayloadC := fresh();
        Actor -ch_c2s(SessID)-> S: Counter. PayloadC;
      channel_goal Client_TLS_Server:
        Actor *->>*             S: Counter. PayloadC;
        S -ch_s2c(SessID)-> Actor: Counter.?PayloadS;
      channel_goal Server_TLS_Client:
        S *->>*             Actor: Counter. PayloadS;
        assert finished: false; %%% for excutability testing;
        % If the model checker does not find an attack at least
        % here, the model has an executability problem.
        % Once this pseudo-attack has been shown, the assertion
        % can be commented out to check for the other goals.
      }
    }

    body { % of Session
      new Client(C,S,ChS);
    }
  }

  entity Server(Actor: agent, ChS: channel,
                Sessions: agent*text*nat set) {

    symbols
      C     : agent;
      SessID: text;

    entity Thread (Actor, C: agent, SessID: text,
                   Sessions: agent*text*nat set) {

      symbols
        N: nat;
        PayloadC,
        PayloadS: text;

      body {
        while(true) {
          select {
            on(C -ch_c2s(SessID)-> Actor: ?N.?PayloadC &
               Sessions->contains((C,SessID,?N))
              ):
            channel_goal Client_TLS_Server:
              C *->>* Actor:               N. PayloadC;
            {
              PayloadS := fresh();
              Actor -ch_s2c(SessID)-> C:   N. PayloadS;
            channel_goal Server_TLS_Client:
              Actor *->>*             C:   N. PayloadS;
              Sessions->remove((C,SessID,     N ));
              Sessions->add   ((C,SessID,succ(N)));
            }
          }
        }
      }
    }

    body { % of the Server
      while(true) {
        select {
          on(? -ChS-> Actor: ?C.?SessID.0 &
                             % Server learns C here!
             !Sessions->contains((?C,?SessID,?))
            ): {
          secrecy_goal secret_SessID: Actor, C: SessID;
            Sessions->add((C,SessID,succ(0)));
            new Thread(Actor,C,SessID,Sessions);
          % The Thread's Actor is the same as for the Server!
          }
        }
      }
    }
  }

  body { % all sessions are with the same server!
    ch_server->confidential_to(server);
    ch_server->   authentic_on(server);
    iknows(0); %%% workaround; should not be needed
    ServerSessions := {};
    new Server(server, ServerSessions);
    any C. Session(C, server, ch_server) where C!=server;
    any C. Session(C, server, ch_server) where C!=server;
  }
}
```

In analogy to the client authentication introduced above in 2.9.5, one could also model a variant for ACM as follows: some unilaterally authenticated channel between the client and the server gets promoted to a fully secure channel, by means of an explicit client authentication using some credentials.

3 ASLan++ concepts and syntax

We have developed ASLan++ to achieve the following design goals:

- The language should be expressive enough to model a wide range of service-oriented architectures while allowing for succinct specifications.
- The language should facilitate the specification of services at a high level of abstraction in order to reduce model complexity as much as possible.
- The language should be close to specification languages for security protocols and web services, but also to procedural and object-oriented programming languages, so that it can be employed by users who are not experts of formal protocol/service specification languages.

3.1 Introductory example

For those readers who have skipped the ASLan++ tutorial (§ 2), here is a simple example of how an ASLan++ specification looks like: the well-known Needham-Schroeder Public Key protocol [21] aiming at mutual authentication of two parties, A and B, which in Alice&Bob notation reads as:

```
1. A -> B: {Na.A}_Kb
2. A <- B: {Na.Nb}_Ka
3. A -> B: {Nb}_Kb
```

Here Na and Nb stand for nonces (i.e. non-guessable random values) produced by Alice and Bob, respectively. The first line says that Alice sends her nonce and her name to Bob, encrypted with his public key Kb, while the second and third line specify similar message transmissions. The second line implicitly requires that Alice checks the value Na she receives; similarly the third line implicitly requires that Bob checks the value Nb he receives.

In ASLan++, this protocol reads as follows, where the various language components will be introduced below.

```
% @satmc(--mutex=1)

specification NSPK_Unsafe
channel_model CCM

entity Environment {

  entity Session (A, B: agent) {

    entity Initiator (Actor, B: agent) {

      symbols
        Na, Nb: text;
      body {
        secret_Na:(Na) := fresh();
        Actor  ->  B: {Na.Actor}_pk(B);
        B  ->  Actor: {Initiator_authenticates_Responder:(Na).
                                              secret_Nb:(?Nb)}_pk(Actor);
        Actor  ->  B: {Responder_authenticates_Initiator:(Nb)}_pk(B);
      }
    }

    entity Responder (A, Actor: agent) {

      symbols
        Na, Nb: text;

      body {
        ? -> Actor: {?Na.?A}_pk(Actor);    % Responder learns A here!
        secret_Nb:(Nb) := fresh();
        Actor  ->  A: {Initiator_authenticates_Responder:(Na).Nb}_pk(A);
        A  ->  Actor: {Responder_authenticates_Initiator:(Nb)}_pk(Actor);
        secret_Na:(Na) := Na;    % Goal can only be given here,
                       % because A is not authenticated before!
      }
    }

    body { % of Session
      new Initiator(A,B);
      new Responder(A,B);
    }

    goals
     %secret_Na:(_) {A,B};                             % Okay.
     %secret_Nb:(_) {A,B};
% Attack found!
     %%% Commented out the above goals such that
     %%% the attack is printed on authentication.
      Initiator_authenticates_Responder:(_) B *->> A;  % Okay.
      Responder_authenticates_Initiator:(_) A *->> B;
% Attack found!
  }

  body { % of Environment
  % Two sessions needed for Lowe's attack.
    any A B. Session(A,B) where A!=B;
    any C D. Session(C,D) where C!=D;
  }
```

```
}
```

In this specification, two sessions of the protocol are launched, each involving one instance of Alice and Bob. The `Alice` entity, used to describe her behavior and security requirement (as long as she is played by an honest agent), has two parameters (of type `agent`): `Actor` is used to refer to herself, while `B` is the name of the party she is supposed to connect to. The `Bob` entity obtains the name of Alice via the first message received. The variables `Na` and `Nb` (of the general type `message`) are used to store the nonces produced by Alice and Bob, respectively. There are secondary secrecy goals expressing that the values of both variables are secrets shared between the two parties. The primary security goals are that Alice strongly authenticates Bob and vice versa, agreeing on the nonces.

A statement like `B -> Actor: {Na.?Nb}_pk(Actor)` reads as follows: the current agent `Actor`, which is Alice, receives a message from `B`, encrypted with the public key of `Actor`, containing three values. Alice learns the value of `Nb` at this point, while she already knows `Na` and thus can check if she received the correct value, which is used to identify her session with Bob.

Note that this specification contains details not expressible in the Alice&Bob notation: how and when values like nonces and agent names are computed and checked, and which sessions are created. Moreover, it contains several secrecy and authenticity goals.

3.2 Specifications

An ASLan++ specification of a system and its security goals has a name (which should agree with the file name it is contained in) and contains the declaration of a hierarchy of *entities*, described in detail below.

An entity may import other entities contained in separate files (known as *modules*, with their filename being the entity name with the extension `.aslan++` appended to it), which in turn contain a hierarchy of entity declarations. There is a "virtual" module available called `Prelude`, defined in § 3.13, that is implicitly part of all specifications.

The top-level (i.e. outermost) entity, usually called `Environment`, is not imported by any other entity, but serves as the global root of the system being specified, similarly to the "main" routine of a program.

Entities and variables have names starting with an upper-case letter, while types, constants and function names start with lower-case letters. The remaining characters in a name may be alphanumeric characters or underscores "`_`" or primes "`'`".

Comments in ASLan++, as well as in the EBNF grammar in § 3.12, start with a "`%`" symbol and extend until the end of the line.

3.3 Entities and agents

The major ASLan++ building blocks are *entities*, which are similar to classes in Java or roles in HLPSL [8]. Entities are a collection of declarations and behavior descriptions, which are usually known as *roles* in the context of security and communication protocols.

Entities may have parameters and contain — via import or textual inclusion — sub-entities with nested scoping. Variables and other items declared in outer entities are inherited[24] to the current one, where inner declarations hide any outer declarations of elements with the same name. However, redeclaring elements of the same kind (e.g. two function declarations or two sub-entity declarations with the same name) at the same level in the same entity is not allowed.

Entities may be instantiated to processes (or threads) executing the *body* of the entity, which contains statements like in procedural programming languages (see § 3.10 for details.)

Entity execution is usually "compressed", which means that the statements in its body are executed atomically, except for certain "breakpoints". Immediately before these breakpoints, compressed execution is broken up, such that other entities and the intruder may interleave. By default, there is just one breakpoint action, namely the reception of messages. Optionally, the modeler may specify for each entity a set of additional breakpoint actions before the entity's body. Each of these actions must be declared constants or functions of (result) type `fact`. This set is inherited to sub-entities unless these declare their own set of additional breakpoint actions.

One can specify *LTL constraints*[25] as assumptions in the root entity; therefore it is possible to specify constraints on all goals of the model, which from the security standpoint is equivalent to constraints on the execution of the overall system.

Each entity has an explicit or implicit formal parameter `Actor`, which is similar to `this` or `self` in object-oriented programming languages. The value of `Actor` is the name of the agent playing the role defined by the entity. This is important for defining the security properties of the entity. `Actor` must be of type `agent` or a subtype of it. If an entity does not have a formal parameter `Actor`, this parameter is defined implicitly. The root entity must not have explicit parameters. Its implicit `Actor` parameter has the value `root`.

3.4 Dishonest agents and the intruder

The attacker is known as the *intruder* and can be referred to by the constant `i` (of type `agent`). Yet we allow the intruder to have more than one "real name".[26] To this end, we use the predicate `dishonest` that holds true of `i` and of every pseudonym (i.e. alias name) `A` of `i`.

We will use `dishonest` also for pseudonyms freshly created by the intruder for pseudony-

[24] Since the OFMC backend does not support inherited variables and CL-AtSe requires at least the option `--nb 2` to handle them correctly, the translator gives a warning in this case.

[25] The only backend supporting this so far is SATMC.

[26] The intruder may have several names that he controls. This reflects a large number of situations, like an honest agent who has been compromised and whose long-term keys have been learned by the intruder, or when there are several dishonest agents who collaborate. This worst case of a collaboration of all dishonest agents may be simply modeled by one intruder who acts under different identities.

mous channels, which are discussed in more detail in § 3.8.3 below; see also [18, 19].[27]

As long as the actual value of the Actor parameter of an entity is an honest agent, the agent faithfully plays the role defined by the entity. If the Actor parameter value is dishonest already on instantiation of the entity, which is typically the case for some of the possibilities included in symbolic sessions, the body of the entity is ignored because the intruder behavior subsumes all honest and dishonest behavior.

We also allow that an entity instance gets compromised later, that is, the hitherto honest agent denoted by the Actor of the entity becomes dishonest. Once an agent has become dishonest, for instance because it has been corrupted, it can never become honest again. Therefore, a dishonest fact may not be retracted.

From the above follows that one may declare any number of dishonest agents at any time (to model, for instance, that an intruder has been able to compromise some machine at some point). The modeler must take care to correctly represent this situation by introducing the fact dishonest(a) and by giving the intruder access to *all* knowledge of agent a needed to behave like a. This may be achieved by introducing iknows facts for

- the private keys of a, in particular iknows(inv(pk(Actor))), and for CCM also iknows(inv(ak(Actor))) and iknows(inv(ck(Actor))),
- the shared keys known to a,
- values of other parameters and local variables that are not already known to the intruder, which may include values produced by a using the fresh() operator and set literals,
- nonpublic symbols used by a.

All other data known to a can then automatically be inferred.

In this way, it is ensured that for an entity that is “played” by a dishonest party, with the sufficient knowledge of long-term secrets (e.g., private keys), the intruder has enough information to perform every legal step of the entity, so that the behavior of that entity can be subsumed by what the intruder can do. [28]

One must then however be careful in the handling of the information that the intruder obtains and how that may affect the goals. For instance, consider a secret between a set of agents. If any of these agents is compromised, the secrecy goal may lead to spurious attacks, which can be avoided by stating just before the agent is declared dishonest:

```
select{ on(child(?IID',IID)): secrecyGoalName_set(IID')->add(i); }
```

[27] For instance, to ensure that the intruder can generate for himself new pseudonyms ψ at any time and can send and receive messages with these new pseudonyms, we use the predicate $\mathsf{dishonest}(\cdot)$ in the rule:

$$=[\psi]\Rightarrow \mathtt{iknows}(\psi).\mathtt{iknows}(\mathsf{inv}(\psi)).\mathsf{dishonest}(\psi)\,.$$

This includes $\mathsf{inv}(\psi)$, which we need for the CCM, where pseudonyms are simply public keys (as, e.g., in PBK). Creating a new pseudonym thus means generating a key pair $(\psi, \mathsf{inv}(\psi))$.

[28] Of course, the intruder does not necessarily stick to the prescribed steps of this entity as an honest agent would.

where the IID' variable[29] must be declared of type nat.

3.5 Facts and Clauses

Instead of the usual type *bool* for truth values, ASLan++ uses the type fact. Atomic propositions are represented by predicates of type fact, and the question whether an atomic proposition holds or not is expressed by the (non-)existence the respective predicate term in a global "fact space".

For truth values represented by facts, only the type fact is allowed. We do not allow variables of type fact.

Facts may be combined with the usual logical operators in LTL formulas to express goals, and they are also used in conditions (e.g. in if statements) known as *guards* (cf. § 3.11).

By default a fact does not hold, but it may be explicitly introduced (simply by writing it as an ASLan++ statement) and retracted. The constant true is introduced automatically and cannot be retracted, while the constant false cannot be introduced.

Facts may also be produced and used by so-called *clauses*[30]. These are rules for "spontaneously" producing facts whenever the conditions in body (i.e., on the right-hand side) are fulfilled, for example:

```
greater_direct   : forall N. succ(N)->greater(0);
greater_induct(M,N):  succ(M)->greater(succ(M)) :- M->greater(N);
```

These conditions are typically positive occurrences of facts, but we have some support [31] also for equalities as well as negative facts and inequalities.

Variables only appearing in the head (and only these) must be explicitly universally quantified using the keyword forall. Each variable occurring in the body must also occur in the head or in the parameter list. The variables occurring in both the head and the body are treated as universally quantified at the outermost level of the clause. The variables only occurring in the body are treated as existentially quantified and must be listed as formal parameters just after the name of the clause, which can be useful for documenting derivations that involve the clause.

When it comes to retracting facts from a state, there is a difference between facts implicitly produced by the clauses, and those explicitly introduced in the state. For instance, for the example clause given above, it would not make sense to manually retract (or introduce) further greater facts. In order to help the modeler to make this distinction, we partition the facts into two sets, called explicit and implicit facts. Implicit facts are those which appear in the head of a clause; the rest of the facts are explicit. It is required that the facts appearing in the head of the clauses cannot be explicitly introduced nor retracted in the specifications.[32] Conversely, facts explicitly introduced or retracted must not appear as the head of any clause. We consider the facts of the form iknows(M), used to model the

[29] The select statement binds it to the instance ID of the parent of the current entity.

[30] We use this name as a shorthand for *definite Horn clauses with side-conditions*.

[31] by the experimental option -not_hc of CL-AtSe

[32] The requirement follows the distinction between policy and state predicates in § A.2.

intruder knowledge, as the only exception to this restriction. These facts may appear both in the head of clauses, and may be freely introduced in specifications. Therefore, `iknows` facts are the only facts that are both explicit and implicit. However, `iknows` facts cannot be retracted, since intruders never lose their knowledge.

Note that while the aforementioned requirement prevents the modeler from explicitly introducing, e.g., `0->greater(0)` to the state, the modeler can alternatively introduce new facts, for example `0->greater'(0)`, to the state, and add the clause

```
greater_extra(X): greater(X,X) :- greater'(X,X);
```

Clauses may include in their bodies side-conditions like (in-)equalities and may be recursive, though this might lead to non-termination of some model checking tools.

3.6 Declarations

Within an entity, declarations of elements like types, variables, constants, functions, macros, clauses, and equations may be freely mixed.

Types and equations always have global scope (i.e. visibility and effect). Types may be declared in any entity, but must not be re-defined in sub-entities. All other declarations have lexical scope, i.e. the declared elements are visible and effective only in the declaring entity and in sub-entities (unless the element is hidden there by a new declaration of a homonymous element of the same kind).

Types may have subtypes, e.g. the (built-in subtype) relation `agent < message` means that any value of type `agent` may be used in a context where a value of type `message` is expected. The type `message` includes all those values that may be sent over the network, in particular concatenation and tuples of sub-messages. For atomic values in messages, it may be more efficient for model-checking with SATMC to use the subtype `text`. Sets are not a subtype of `message`, such that they cannot be directly sent as messages, but see § 2.5 for possibilities how to deal with this.

The only types that have (implicit) type parameters are tuples (e.g. `agent * message`) and sets (e.g. `nat set`).

There are also are *compound types*, which are a shorthand for specifying structural constraints on terms. These are mainly used for emphasizing the structure of messages sent or received. (This is also very helpful for the performance of SATMC.) For example, a variable declaration

```
X: crypt(public_key, agent.message)
```

is expanded to a set of declarations

```
X1: public_key; X2: agent; X3: message;
```

and every occurrence of the variable X in a term is replaced by

```
crypt(X1, X2.X3).
```

A compound type f(T1,T2,...) where f has result type T is regarded as a subtype of T. A compound type using the tuple constructor like (T1, T2) is equivalent to T1 * T2. Both forms of denoting a tuple type have higher syntactic precedence than general compounds and concatenation compounds.

The symbols section is used to declare constants and functions. The names of symbols must be unique, i.e. overloading is not permitted.

Macros are used to shorten and simplify recurring terms. The function (or constant) name on the left-hand side must not be declared in the symbols section or inherited from any enclosed entity.

User-defined symbols (as long as their (result) type and all of their argument types are subtypes of message or of some set type) are by default public, which means they are known and usable by the intruder. To avoid this default behavior, the keyword nonpublic must be given before the symbol name when declaring the function or constant. Similarly, user-defined functions are by default invertible in each argument (as long as their result type and the respective argument type are subtypes of message or of some set type), which means that the intruder can derive the arguments of a function call when he knows its result. To avoid this default behavior, the keyword noninvertible must be given before the function name when declaring the function. The special function symbol iknows, which is used to describe the knowledge of the intruder, may also be used to describe special cases of invertibility. For instance, to specify that the intruder can invert the function f in its first argument, one can write the following Horn clause involving iknows facts:

```
iknows_f1(X,Y): iknows(X) :- iknows(f(X,Y));
```

Equations are used to define algebraic properties, e.g. the property of exponentiation that exp(exp(g,X),Y) = exp(exp(g,Y),X) is necessary for Diffie-Hellman based key-exchange. They are only allowed at the root level, i.e. in the outermost entity, and thus are global. All variables occurring in them are implicitly universally quantified.

3.7 Terms

Terms may consist of

- literals for the Boolean values true and false,
- literals for natural numbers, e.g. 0 and 99, where values up to 9 are translated as the respective power of the predefined succ function, such that for instance 2 is translated to succ(succ(0)),
- variables, e.g. A,
- symbolic constants, e.g. b2,
- function applications (or to be more precise, function symbols applied to first-order terms, e.g. hash(Msg)), including infix right-associative message concatenation concatenation, e.g M1.M2 and tupeling, e.g. (A,b2,0).

Terms, as well as variables on the left-hand side of assignments, may be “labeled” with a goal name (e.g. `my_name:(My_term)`), which is used as *goal labels* for channel goals and secrecy goals, as described in § 3.9.3 and § 3.9.4.

Except when explicit equations are given for them, “Function calls” and constructors like concatenation of messages are not really evaluated but are kept as kind of “tags” around their argument list. [33]

To enhance readability for instance for functions that one would like to write as infix operators (like e.g. the binary predicate `weaker_than`), any function application of the form `f(X,Y,Z,...)` may alternatively be written “in object-oriented style” as `X->f(Y,Z,...)`.

Set literals are written as usual (e.g. `{A,B,C}`)., Variables of set type hold globally scoped references to the set they represent. These set references may be copied and even passed as arguments of `send` and `receive` instructions. If a reference gets known to the intruder, he may modify[34] the set by adding or removing the respective `contains` facts.

The basic operator on sets is the `contains` function, where the presence of the fact `Set->contains(X)` means that `X` is a member of `Set`. There is syntactic sugar for the introduction and retraction of `Set->contains(X)`, namely `Set->add(X)` and `Set->remove(X)`.

The precedence of operators in guards and other formulas is, in descending order: ’`(`’ ... ’`)`’, ’`->`’, ’`!`’, ’`!=`’, ’`=`’, ’`&`’, ’`|`’, ’`=>`’. Conjunction, disjunction and implication associate to the right.

Encryption of a message `M` with a symmetric key `K` can be written as `scrypt(K,M)` or equivalently as `{|M|}_K`. Asymmetric encryption with a public key `PK` can be written as `crypt(PK,M)` or as `{M}_PK`. The very same syntax is used also in receive operations where it means *de*cryption. A digital signature with the private key `PK’` corresponding to the public key `PK`, i.e. `PK’ = inv(PK)`, can be written as `sign(PK’,M)` or as `{M}_inv(PK)`. The very same syntax is used also in receive operations where it means signature *checking*.

3.8 Channels

ASLan++ offers a concept of *channels* that allows specifications to abstract from the technical details of the realization of communication and their protection, such as cryptographic mechanisms. First, we describe the meaning of channels *as assumptions*, i.e. when for the transmission of some of its messages a protocol or service relies on channels with particular properties. Second, we also introduce the meaning of channels as *goals*, i.e. when a protocol or service aims at establishing a particular kind of secured channel. Moreover, there is a compositionality aspect behind channels, mentioned below.

3.8.1 Channel models

For channels as assumptions, we currently support three different channel models (ACM, CCM, ICM; see [2]), as described in the subsequent sections. All our channel models are

[33] Formally, terms are interpreted in the quotient algebra $\mathcal{T}_{/E}$, where E are the specified algebraic equations. By default, when no algebraic equations are specified, all functions are uninterpreted (i.e. in the free algebra).

[34] At the moment, this is only implemented by CL-AtSe.

based on certain events in the system execution trace implementing unidirectional message-based communication from a single sender to a single (intended) receiver.

Abstract Channel Model (ACM). In ACM, a channel is referred to by a name and provides a communication link relating all messages sent and received using this name. The assumed properties of a channel are declared as LTL constraints over explicit send and receive events usually with the help of a set of predefined fact symbols.

Cryptographic Channel Model (CCM). In CCM, channels are realized by individual transmission of messages via the (Dolev-Yao style) intruder. Multiple transmissions may be related using pseudonyms and user-defined identifiers. Assumed security properties are defined via concrete implementations, like certain ways of encrypting or signing messages.

Ideal Channel Model (ICM). Like CCM, the ICM focuses on individual message transfers. In contrast, assumed security properties of channels are described by abstract fact symbols and special transition rules that model the intruder's limited ability to send and receive on those channels.

We have shown in [2, 18] that CCM and ICM are equivalent under certain realistic assumptions, while the ACM operates at a different abstraction level than CCM and ICM. We are working at establishing a formal comparison between the ACM and the CCM/ICM; establishing such equivalence results is fundamental as they allow us to use each model interchangeably, according to what fits best with certain analysis methods. A detailed comparison of the strengths of the different models is given in [2, 4], while here we just summarize the main differences.

There are slight differences in the way ACM and CCM/ICM are used to formulate the interesting channel properties and in terms of expressiveness and scope of the models.

In ACM the modeler can conveniently refer to channels via their names. This allows, for instance, to disambiguate several channels between the same participants, and to specify in a very convenient way that for several message transmissions the very same channel is used. In CCM/ICM, if needed this can be simulated by the modeler, by attaching channel names to the transmitted messages.

ACM has more freedom to specify assumed security properties via LTL constraints, allows in particular for the definition of resilient channels, which cannot be expressed in CCM/ICM. [35]

An important feature of CCM/ICM are the availability of compositionality results between channels as assumptions and channels as goals [2, 18], where the ICM serves as the abstract reference model. Since ICM and CCM are equivalent, at the ASLan++ level CCM channels and ICM channels are expressed in the same way; the differences of these two channel models manifest in their translation to ASLan.

[35] On the other hand, potentially there may be properties that can be described more adequately in CCM by means of cryptographic primitives.

3.8.2 Channel security notions

Since the term *channel* has been used in many different ways both in academia and in the ICT industry, in order to avoid any confusion, let us first clarify our notions for channels and their properties, depending on the particular channel model used.

For all channel models, the "default" communication medium is a channel without any protection (an *insecure* channel). Here, the communication medium is fully controlled by an intruder who, according to the Dolev-Yao model [11], can read and intercept all messages, as well as send arbitrary messages he can craft to any agent (as long as he has the appropriate keys, i.e. he does not break cryptography). One may imagine this situation as the intruder *being* the network.[36]

Inspired by the "bullet-notation" of [16], for CCM and ICM we introduce several kinds of channels that offer various forms of protection:

- $A \rightarrow B$, like it is commonly written in the analysis of simple security protocols, denotes the default *insecure channel from* A *to* B,

- $A \bullet\!\!\rightarrow B$ denotes an *authentic channel from* A *to* B. This means that B can rely that any message M he receives via this channel indeed comes from A. One may thus say the channel protects the identity of the sender and use the bullet to indicate this.

  Note that authentication is tied to the integrity of the transmitted message: B can be sure that A has sent exactly that message M. Furthermore, in all channel models, for channels that are (at least) authentic, B can be sure that the message was meant for him. This additional property, known as *directedness*, has been motivated in [2, §4.1.4]. For cases where the directedness does not make sense, for instance for broadcasts, there is an option to use *undirected* authentication.

  The authentic channel does not however protect message transmission against eavesdropping (the intruder can still see the message), disruption (the message does not arrive at the intended receiver), replay (the message may arrive several times, even much later), or re-ordering (messages may not arrive at the recipient in the order they were sent.

  The behavior of an authentic channel is thus like a digital signature on the message by the sender, including the intended recipient as part of the signed text.

- $A \rightarrow\!\!\bullet B$ denotes a *confidential channel from* A *to* B. This means that A can be sure that any message M sent over this channel can only be read by B, hence the bullet on the receiver side. Any intruder cannot learn the contents of M sent on this channel.

  Confidential channels do not however protect the message transmission against intercepting messages and replacing them with different ones, spoofing (the intruder may introduce messages on the channel in the name of other parties), disruption, replay (with CCM/ICM), or re-ordering.

[36] The situation is more complex, however, if there are several non-collaborating intruders.

The behavior of a confidential channel can thus be thought of as an encryption with the receiver's public key.

- $A \bullet\!\!\rightarrow\!\!\bullet B$ denotes a *secure channel from A to B*, in the sense that both end-points are protected, such that messages are transmitted authentically and confidentially.

All these kinds of channels can be considered also when one (or both) of the parties involved is identified by a pseudonym (e.g. for *sender* and/or *receiver invariance*, as described in [2, §4.1.4]).

For ACM there is the following abstract notation: $A \xrightarrow{\boxed{ch}} B$. This notation denotes a communication between A and B over the channel ch. The assumed properties of the channel ch can be specified using facts. For doing so, there is a set of predefined facts, which are described along with their informal definitions in § 3.8.3.

Besides types of channels mentioned above, we can consider further properties. These are of interest for protocols such as TLS, which additionally offers order preservation, replay protection, some limited protection against traffic analysis (e.g. the number of messages sent and so on), etc.

The following properties are partially supported in ASLan++, depending on the channel model, as described in the subsequent sections.

- *Resilient channels*: the intruder cannot disrupt the communication indefinitely and messages will thus eventually arrive at the intended recipient. With reference to the formalism presented in D3.3 [2, §4.2], this amounts to requiring that every message sent over the channel may be delayed by the intruder for an arbitrary, but finite amount of time, but will be eventually delivered to the intended recipient.

- *Pseudonymous channels*: one endpoint is not authenticated, but uses a unique pseudonym relative to which the channel is secured.

- Channels that are *protected against replay*. This is also known as *freshness* of a channel: each sent message can be received only once. In ASLan++, we directly support this notion as a communication goal. In ACM this is implied by the confidentiality and weak confidentiality channel assumptions, while in CCM/ICM it can be implemented by using sequence numbers or challenge-response.

- Channels that are *order-protecting*. We do not have this as an ASLan++ primitive so far, but it can be achieved by including sequence numbers into the messages.

- *Linked channels*, where we distinguish different channels between two end points. This is directly supported by ACM, while for CCM/ICM it can be achieved by including a unique identifier into the messages.

In addition to their use as assumptions, slightly abusing the notion (and notation) of channels, we also allow for *channels as goals*. This is inspired by a compositionality result [18]. The idea is that complementing security assumptions on which a system can rely when using channels, one can also ask what properties a system ensures when aiming to realize a certain kind of secured channel. In fact, as shown in that paper, when considering a fixed set of "application" messages being exchanged in the system modeled, we can identify the goals for authentic and confidential channels as the classical authentication and secrecy goals. [37]

3.8.3 Channel syntax

To express security properties of (potentially pseudonymous) channels, we use and partially extend the notation we introduced in Deliverables D2.2 [4] and D3.3 [2] and recalled above.

While [16] uses the bullet notation to reason about the existence of channels, we use it to specify message transmission in services in two ways.

- First, as described in this subsection, the modeler may use channel properties as *assumptions*, i.e. when a service relies on channels with particular properties for the transmission of some of its messages.
- Second, the service may have the *goal* of establishing a particular kind of channel. Details on how to specify channel goals are given in § 3.9.3.

Note that the properties for channels as goals refer to individual transfer of messages only, which also holds for CCM and ICM channel assumptions, whereas ACM channel assumptions may stretch over multiple message transmissions.

In an ASLan++ specification, the channel model is globally specified, right at the beginning after the specification name, using the keyword `channel_model`.

For ICM and CCM, but also for channels goals in all channel models including ACM, we adopt the intuitive notation from [16] briefly introduced above, where a secure endpoint of a channel is marked with a bullet. Just note that while so far we have used abstract bullet symbols, we now switch to the concrete ASLan++ syntax and write, e.g., `A *-> B: M` instead of $A \bullet\!\!\rightarrow B : M$.

- `A *->  B: M` represents an *authentic transmission of M*.
- `A  ->* B: M` represents a *confidential transmission of M*.
- `A *->* B: M` represents a *secure transmission of M*.

There is also a variant of the channel notation using a double-headed arrow like "`->>`" representing a *fresh channel*. This notation may be used for channel goals that include at least authenticity, e.g. `A *->>* B: M` represents the goal that a channel is both secure and fresh.

For authentic transmission (but not confidential/secure transmission), the undirected variant is indicated by a "`?`" just after the arrow, that is, either "`*->?`" or "`*->>?`".

Similarly, to represent the abstract syntax $A \!\longrightarrow\!\boxed{ch}\!\longrightarrow\! B : M$ for ACM channel assumptions in ASLan++, we use the concrete syntax `A -ch-> B: M`.

[37] In fact, this identification has shown a problem in many classical authentication definitions that had not been observed before; the details are found in [18].

Table 1 and Table 2 show the syntax notations that ASLan++ supports for CCM/ICM and ACM, respectively.

Channels may be specified in "OO-style" notation as well as in the annotated channel notation. For sends and receives, if the first argument is `Actor`, like in `send(Actor,B,M)` or `Actor->receive(B,M)`, it may be omitted such that only `send(B,M)` or `receive(B,M)`, respectively, is written.

The sender and receiver terms in the channel notation must be of type `agent` or a subtype of it.

CCM and ICM. In CCM and ICM, we allow agents to be identified by pseudonyms rather than by their real names. We denote this as `[A]_[P]` where `A` denotes the real name and `P` is the pseudonym. Authentication and confidentiality of a pseudonymous endpoint is only with respect to the pseudonym (and not with respect to the real name). When the real name is not used at all, one may also write `[?]_[P]`.

In order to make pseudonyms directly applicable for asymmetric encryption, they have type `public_key`. We regard `public_key` as a subtype of `agent`, such that pseudonyms can be used in place of an agent real name. By default, every agent will create a fresh pseudonym upon instantiation; we can write simply `[Actor]` to denote the current agent acting under this *default pseudonym*.

For instance, to model transmissions over a TLS channel without client authentication, one can simply use `[Actor] *->* Server: M` for a message `M` from the pseudonymous `Client` to the `Server`, and `Server *->* [Actor]: ?M'` for the opposite direction. This does not give all protections of TLS, but is usually sufficient. If the additional properties of replay protection, order preservation, and session distinction are needed, then they can be constructed as described in § 2.9.6.

In Table 1, `ActorP` stands for `Actor` or any pseudonym of `P` it, which is either the default pseudonym `[Actor]` or `[Actor]_[P]`. Similarly, `B` stands for the (intended) communication partner of `Actor` or any pseudonym `P` of this partner, which is denoted by `[B']_[P]` where `B'` is a term different from `Actor` with irrelevant value.

ACM. In ACM, channels are referred to by terms of type `channel`. Their properties are usually declared by introducing certain pre-defined facts, introduced below, which trigger pre-defined LTL constraints. One must of course be careful not to require non-fulfillable (e.g., contradictory) channel properties, which would lead to a non-executable model.

As mentioned before, Table 2 shows the general syntax that ASLan++ supports for ACM. Differently from the CCM and ICM, in ACM the arrow is labeled with the name of the channel. The name may be used for identifying the channel to relate several send/receive statements, as well as for stating the assumed properties of the channel.

The named channel syntax can be used only for ACM channels as assumptions, while – even when ACM is chosen – channel goals are always expressed and interpreted in the same way as in CCM/ICM.

Like in CCM and ICM, by default a channel does not guarantee any protection (insecure channel), and is controlled by a Dolev-Yao intruder. The other properties of the channels

“OO-style” notation	Annotated channels notation
ActorP->send(B,M)	ActorP -> B: M
ActorP->receive(B,M)	B -> ActorP: M
ActorP->send(?,M) over uauthCh	ActorP *->? ?: M
ActorP->receive(B,M) over uauthCh	B *->? ActorP: M
ActorP->send(B,M) over authCh	ActorP *-> B: M
ActorP->receive(B,M) over authCh	B *-> ActorP: M
ActorP->send(B,M) over confCh	ActorP ->* B: M
ActorP->receive(B,M) over confCh	B ->* ActorP: M
ActorP->send(B,M) over secCh	ActorP *->* B: M
ActorP->receive(B,M) over secCh	B *->* ActorP: M
ActorP->send(?,M) over fresh_uauthCh	ActorP *->>? ?: M
ActorP->receive(B,M) over fresh_uauthCh	B *->>? ActorP: M
ActorP->send(B,M) over fresh_authCh	ActorP *->> B: M
ActorP->receive(B,M) over fresh_authCh	B *->> ActorP: M
ActorP->send(B,M) over fresh_secCh	ActorP *->>* B: M
ActorP->receive(B,M) over fresh_secCh	B *->>* ActorP: M

Table 1: Channel notations in ASLan++ for CCM and ICM

“OO-style” notation	Annotated channels notation
Actor->send(B,M) over ch	Actor -ch-> B: M
Actor->receive(B,M) over ch	B -ch-> Actor: M

Table 2: Channel notations in ASLan++ for ACM

that are currently supported are given in this section. In Table 3, for each channel property the corresponding fact and an informal description are reported. The formal semantics is provided at the ASLan level, as defined in § 4.7.6, using LTL constraints.

The facts indicating the assumed security properties of a channel must be introduced before the channel is used and should not be retracted. They are typically stated in the body of the entity Environment or a session entity.

Note that the confidentiality and authenticity properties defined in the table refer to a pair of agents. In principle, multiple senders and/or recipients can be defined for a channel, but currently they are not supported. Further note that the weak form of confidentiality and authenticity are obtained in CCM by using pseudonyms.

For any given channel, the properties of (weak) confidentiality, (weak) authenticity, and resilience, etc. may be combined freely.

To give an example of ACM channels, let us consider the following specification. Two entities and the respective agents (client and server) are involved, and two channels are defined (lines 7) ch_c2s and ch_s2c. The channel ch_c2s is confidential to the server (line 43), while the channel ch_s2c is authentic for the server (line 44).

```
1  specification ACM_example
2  channel_model ACM
```

Property	Fact	Meaning
Confidential	`confidential_to(ch,B)`	A channel `ch` provides confidentiality if its *output* is exclusively accessible to the specified receiver `B`.
Weakly confidential	`weakly_confidential(ch)`	This is the relaxed notion of confidential channel. A channel `ch` provides weak confidentiality if its output is exclusively accessible to a single, yet unknown, receiver.
Authentic	`authentic_on(ch,A)`	A channel `ch` provides authenticity if its *input* is exclusively accessible to the specified sender `A`.
Weakly authentic	`weakly_authentic(ch)`	This is the relaxed notion of authentic channel. A channel `ch` provides weak authenticity if its input is exclusively accessible to a single, yet unknown, sender.
Resilient	`resilient(ch)`	The channel `ch` is normally operational but an attacker can succeed in delaying messages by an arbitrary, but finite amount of time. In other words, a message inserted into a resilient channel will eventually be delivered.

Table 3: Properties, channel facts, and their informal meaning

```

entity Environment {
  symbols
    client, server : agent;
    ch_c2s, ch_s2c : channel;

  entity Session (C, S: agent,
                  Ch_C2S, Ch_S2C : channel) {

    entity Client(Actor, S : agent,
                    Ch_C2S, Ch_S2C : channel) {
      symbols
        Nc, Ns: text;
      body {
        ...
        Actor -C_C2S-> S: Nc;
        S -C_S2C-> Actor: ?Ns;
```

Property	Fact	Meaning
Linked	link(ch_a2b,ch_b2a)	This relation requires that the entity sending messages over the channel ch_a2b is the same entity that receives messages from ch_b2a.

Table 4: Linked channel, fact, and its informal meaning

```
20       }
21     }
22
23     entity Server(Actor: agent,
24                   Ch_C2S, Ch_S2C: channel) {
25       symbols
26         C: agent;
27         Nc, Ns: text;
28       body {
29         ?C -Ch_C2S-> Actor: ?Nc;
30         ...
31         Actor -Ch_S2C-> C: Ns;
32       }
33     }
34
35     body {
36       new Client(C, S, Ch_C2S, Ch_S2C);
37       new Server(   S, Ch_C2S, Ch_S2C);
38     }
39   }
40
41   body {
42     ...
43     ch_c2s->confidential_to(server);
44     ch_s2c->   authentic_on(server);
45     new Session(client, server, ch_c2s, ch_s2c);
46   }
47 ...
48 }
```

If we want to define the channel ch_c2s as resilient, we may simply add the following line to the body of the entity session (line 41): resilient(ch_c2s);

It could also be important to constrain the behaviour of different channels, as done in the relation between channels defined in Table 4.

By leveraging the elementary properties defined above, it is possible to define more complex properties and channel relations, approximating real-world transport level communications such as SSL/TLS better than possible using CCM (see 2.9.5 and 2.9.6), like the

Property	Fact	Meaning
Bilateral confidential and authentic	`bilateral_conf_auth (ch_a2b,ch_b2a,a,b)`	The relation of the pair of channels `ch_a2b` and `ch_b2a`, for instance modeling a run of SSL/TLS in which entity `a` and `b` have a valid certificate with each other.
Unilateral confidential and authentic	`unilateral_conf_auth (ch_a2b,ch_b2a,b)`	A pair of channels `ch_a2b` and `ch_b2a`, where `ch_a2b` is confidential to `b` and weakly authentic and `ch_b2a` is weakly confidential and authentic for `b` with the additional requirement — referred to as the *link* requirement — that the principal sending messages on `ch_a2b` is the same principal that receives messages from `ch_b2a`. This can be used to model a run of SSL/TLS in which principal `b` has a valid certificate, but principal `a` does not.

Table 5: Properties, channel facts, and their informal meaning

ones given in Table 5.

With reference to the previous example, if we want to model a TLS transmission without client authentication, we may simply replace lines 43 and 44 with the following one: `unilateral_conf_auth(ch_c2s,ch_s2c,server)`.

Another feature of the ACM is the possibility to use either the same or different channel identifiers for different sessions. For instance, with reference to the previous example, a modeler can consider a further session, adding the following statement in the body of the entity session (line 41):

```
new Session(client, server, ch_c2s, ch_s2c);
```

In this way, the same pair of channels is shared between different sessions. Alternatively, the modeler can consider the following session:

```
new Session(client, server, ch_c2s_new, ch_s2c_new);
```

For instance, in case of the `unilateral_conf_auth` channel relation, the use of different channels allows to model different TLS sessions.

Channels may be created even dynamically, by producing fresh channel names and using these new values when stating their security properties and when sending and receiving on these channels. See 2.9.6 for an example.

Extensions. It is important to note that we have designed ASLan++ so to allow the modeler to choose the particular channel model he wishes to consider. We have therefore introduced the schematic notation considered above, which can then be translated to different models and is open to the extension with other features, such as further channel types (e.g. like the ones considered in [10]), a more fine-grained definition of the nature of a pseudonym, and so on, which may be realized in a different way depending on the particular underlying channel model. In other words, as we will see in more detail below, we deliberately did not fix the semantics of ASLan(++) on the channel model question but instead allow the modeler to "plug-in" different kinds of channels and models for them, which allows us to later define more and/or different models. Such extensions are currently in progress and can be integrated into the AVANTSSAR Platform in future.

3.9 Goals

Validation goals have a name and give rise to an LTL formula (or equivalently, an attack state) that is checked by the validation back-ends at appropriate times.

An LTL formula may contain predicates (of type `fact`), which may refer to the values of variables in the scope of the current entity (typically at the current state of execution), the usual first-order logic operators (namely, propositional connectives and quantifiers), and future and past temporal operators. Free variables, i.e. those not declared in the given entity nor inherited from any outer entity, are implicitly universally quantified. The LTL operators for ASLan++ are listed in Table 6. See § A.3 and § 4.12 for formal definitions of the operators.

Note that most LTL formulas that state a liveness property (e.g. `<>(P)`) would not hold for any ASLan++ (or ASLan) model, because the semantics of specifications are Kripke structures in which the transitions relations are reflexive (see § A.3). In order to verify liveness properties, therefore, fairness assumptions must be added to the Kripke structure, which can be done via LTL constraints.

In ASLan++ there are different kinds of goals, which we describe below.

3.9.1 LTL goals and invariants

LTL formulas that are expected to hold during the whole life of an entity instance can be stated in the `goals` section of the entity declaration. If the formula should hold initially or globally during the overall system execution, it should be given in the outermost entity.

By default, the formula is interpreted relative to the initial state of the system. If the formula is used to describe an *invariant*, which means a property that should hold initially and in all subsequent states, the formula should start with an outermost LTL "globally" operator, i.e., should have the form `[](f)`.

For LTL goals (including invariants), basically any LTL formula can be given, but, by now, none of the backends supports all LTL operators at all positions. Moreover, so far the use of local and inherited variables in LTL goals not supported by OFMC.

[38] Due to stuttering at the ASLan level and merging of transitions during translation, these operators make little sense at the ASLan++ level.

[39] Due to ASLan's stuttering semantics, this only makes sense with fairness constraints.

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[38]
Yesterday	`Y(f)`	in the previous state[38]
Finally	`<>(f)`	at some time in the future[39]
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 6: LTL operators for specifying goals

3.9.2 Assertions

Assertions are given as a statement in the body of an entity. They are like invariants, except that they are expected to hold only at the given point of execution of the current entity instance.

3.9.3 Channel goals

A channel goal has the form

```
name:(_) Sender Channel Receiver
```

where the goal name is augmented with a parameter placeholder `(_)` to indicate that in sub-entities the goal name will appear again, in the form of a goal label, with a message term as its argument.

Similar to the syntax of message transmission, `Sender` and `Receiver` can refer to real names or to a pseudonym, and `Channel` is a symbol for the kind of channel used. For instance, the goal of implementing TLS without client authentication to send a suitable payload message is `[Client] *->> Server`.

Symbol	Explanation
`->*`	confidentiality
`*->?`	undirected authenticity
`*->`	authenticity and directedness
`*->>`	authenticity and directedness plus freshness
`*->*`	authenticity, directedness, and confidentiality
`*->>*`	the above plus freshness
`*->>?`	undirected authenticity and freshness

Table 7: Channel goal symbols in ASLan++

A channel goal is usually stated at the level of a session entity and pertains to those message transmissions in sub-entities that contain goal labels using the same goal name. See also the example below.

When goals are specified using the channel notation, they refer to individual message transfer even when the ACM is chosen for channel assumptions. Independently of the chose channel model (which only affects channel assumptions), the shape of the arrow symbol in channel goals specifies the type of the goal. The shape is the same as for channels as assumptions in the ICM/CCM. This implies the intuitive meaning for each type of arrow as given in Table 7. More details can be found in [2, §4.1.4] and [18, 19].

As an example consider the following specification excerpt:

```
entity Session (A, B: agent) {

  entity Alice (Actor, B': agent) {
    symbols
      Payload: message;
    ...
    body {
      ...
      Actor -> B': ...secure_Alice_Payload_Bob:(Payload)...
      % send Payload to B, somehow fully secured
      ...
    }
  }

  entity Bob (A', Actor: agent) {
    symbols
      Payload: message;
    ...
    body {
      ...
      A' -> Actor: ...secure_Alice_Payload_Bob:(?Payload)...
      % receive Payload from A, somehow fully secured
      ...
    }
  }

  body {
    new Alice(A,B);
    new Bob   (A,B);
  }
  goals
    secure_Alice_Payload_Bob:(_) A *->* B;
}
```

A channel goal is given as follows:

- The goal is declared — including its goal name — in an entity enclosing both the sending and receiving entity. In the above example, the sender is `Alice`, the receiver is `Bob`, and the enclosing role is `Session`.

- The channel symbol (“arrow”) describes the property that the channel between the sender and the receiver is supposed to have; for details please refer to Table 7 and § 3.8.

- There must be at least one sub-entity containing a send statement that contains the goal label. In the above example, `Alice` is the (only) sending entity for the given channel goal. Unless a wildcard “`?`” is used (in which case the sender and receiver information is determined from the send statements containing the goal label), the agent term `A` occurring in the channel goal declaration must be handed over as an actual argument given for the `Actor` parameter of the sending entity, and the agent term `B` must be handed over as the actual argument given for a further parameter (referring to the receiver, named `B'` in the above example) of the sending entity. If more than one argument qualifies (as defined in more detail for secrecy goals in § 3.9.4) to refer to the sender or receiver, the translator chooses the one occurring first and gives a warning. The values of the two parameters (`Actor` and `B'` in the above example) determined to refer to the sender and receiver side of the channel goal are evaluated when executing the send statement.

- Analogously, there must be at least one sub-entity containing a receive statement that contains the goal label. In the above example, `Bob` is the (only) receiving entity for the given channel goal. Unless a wildcard “`?`” is used (in which case the sender and receiver information is determined from the receive statements containing the goal label), the agent term `B` occurring in the channel goal declaration must be handed over as the actual argument given for the `Actor` parameter of the receiving entity and the agent term `A` must be handed over as the actual argument given for a further parameter (referring to the sender, named `A'` in the above example) of the receiving entity. If more than one argument qualifies (as defined in more detail for secrecy goals in § 3.9.4) to refer to the sender or receiver, the translator chooses the one occurring first and gives a warning. The values of the two parameters (`Actor` and `A'` in the above example) determined to refer to the sender and receiver side of the channel goal are evaluated when the executing the receive statement.

- The payload terms labeled with the goal name on both the sender and receiver sides indicate the value to the protected during transmission. These terms may look different (e.g. due to different variable names), but their actual values, evaluated when the send or receive statement is executed, should of course be equal.

Note that it is allowed to declare channel goals that state e.g. authentic *indirect* transmission, for instance from `Alice` via `Bob` to `Charlie`:

```
entity Session (A, B, C: agent) {

  entity Alice (Actor, B, C: agent) {
      ...
      Actor -> B: ...authentic_Alice_Charlie:(Payload)...
      ...
  }

  entity Bob (A, Actor, C: agent) {
      ...
  }

  entity Charlie (A, B, Actor: agent) {
      ...
      B -> Actor: ...authentic_Alice_Charlie:(?Payload)...
      ...
  }


  ...
  goals
    authentic_Alice_Charlie:(_) A *-> C;
}
```

In case a more complex channel goal cannot be expressed given the above syntactic restrictions, advanced users may directly use the formulation using `witness` and `request` facts, as described in § 4.12.2.

3.9.4 Secrecy goals

A secrecy goal in an entity E has the form

`name:(_) {A1, A2, ...}`

where the goal name is augmented with a parameter placeholder `(_)` to indicate that in some (sub-)entities E' the goal name will appear again, in the form of a goal label, with a term as its argument.

Its interpretation is that the value referred to via the goal label parameters must not be known to the intruder, unless the intruder is in the set of “knowing” agents, specified by the set literal {A1, A2, ...}. Note that the goal is violated only if *the intruder* illegitimately learns the given secret. The case that an honest agent not listed in the set learns the secret is not checked using this type of goal. If one want sto check such a property, it needs to be given as a user-defined LTL formula.

While confidential transmission of a value between two parties can also be stated as a channel goal, a secrecy goal is more general because it may be used to state that the value is shared between more than two parties, and it is stronger in the sense that it requires the given value to remain confidential for the reset of the system run.

The goal labels should be given in all sub-entities that may know the secret value. Moreover, it should be given as early as possible, for instance when the value is produced or learned by the respective entity instance. Here is an example:

```
entity Session (A, B: agent) {

  entity Alice (Actor, B: agent) {
    symbols
      Nonce: message;
    ...
    body {
      ...
      secret_Nonce:(Nonce) := fresh();
      Actor -> B: ...Nonce...;
      % send Nonce to B, somehow confidentiality protected
      ...
    }
  }

  entity Bob (A, Actor: agent) {
    symbols
      N_A: message;
    ...
    body {
      ...
      A -> Actor: ...secret_Nonce:(?N_A)...;
      % receive Nonce from A, somehow confidentiality protected
      ...
    }
  }

  body {
    new Alice(A, B);
    new Bob  (A, B);
  }
  goals
```

```
    secret_Nonce:(_) {A,B};
}
```

For each secrecy goal, the message terms given in the sub-entities involved may look different (e.g. due to different variable names, like in the above example: `Nonce` vs. `N_A`), but their actual values should of course be equal.

Wherever a term labeled with the goal name appears, the set of the "knowing" agents is computed from the current values of those entity parameters that have been identified as follows.

All terms `A1`, `A2`, etc. must be handed over as actual arguments in the instantiations of all sub-entities sharing the secret. To be more precise, for each sub-entity E' where the goal label appears in its body and for each term `A1`, `A2`, etc. appearing in E, determine the set of all the instantiations (e.g, via `new` statements) of E' and follow to which argument position(s) of the entity E' the term is handed over. If E' is not a direct sub-entity of E, this may involve intermediate entities. For the set of instantiations constructed in this way, there must be at least one (and should be at most one) argument position at which the term is handed over. If there is more the one such position, maybe in different instantiations of the sub-entity E', the translator gives a warning and takes the first parameter found.

In the above example, for the secrecy goal `secret_Nonce:(_) {A,B}` the set of knowing agents within `Alice` is computed as {`Actor`, `B`} and within `Bob` as {`A`, `Actor`}. If `Alice` had one more parameter, for instance: `Alice(Actor, B, A': agent)` and `Alice` was instantiated in the session three times as follows:

```
   new Alice(A, B, A);
   new Alice(A, B, A);
   new Alice(i, B, A);
```

then according to the above description the translator again determines `Actor` as the "knowing" agent in `Alice` related to the term `A` given in the secrecy goal. Yet it issues warnings because `A` occurs more than once as an argument in the instantiation `new Alice(A, B, A)` and because `A` occurs in the first argument position of two instantiations but not in the third instantiation.

In case a more complex secrecy goal cannot be expressed given the above syntactic restrictions, advanced users may directly use the formulation using `secret` facts, as described in § 4.12.3. In particular, it is possible to dynamically change the set of agents that may know the value. For example, consider the secrecy goal

```
    secret1:(_) {A,B};
```

and suppose we want to manipulate the set of agents within a child entity of the entity declaring the secrecy goal. After binding an auxiliary local variable like `IID': nat` to the instance ID of the parent entity, which can be achieved by

```
  select{ on(child(?IID',IID)): {} }
```

we can dynamically add `c` to the set by introducing the fact

```
    secret1_set(IID')->add(c);
```

or remove B from the set by stating

```
    secret1_set(IID')->remove(B);
```

where the secret1 part is the name of the secrecy goal.

3.10 Statements

Statements may be the usual assignments, branches and loops, but also nondeterministic selections, assertions, the generation of fresh values and of new entity instances, the transmission of messages (i.e. send and receive operations), and the introduction or retraction of facts.

The select statement is typically used within the main loop of a server, which handles a variety of potential incoming requests or other events such as timeouts. It checks the guards given, blocking as long as no guard is fulfilled, then nondeterministically chooses any one of the fulfilled guards and executes the corresponding statement block. The evaluation of the guard chosen may assign variables as described in § 3.11.

Assertions induce goals to be checked at the current position in the control flow.

Entity instantiation, introduced by the keyword new or any, generates new instances of sub-entities. This is only allowed for direct sub-entities, such that static and dynamic scoping coincide.

Symbolic entity instantiations, introduced by any, are a convenient shorthand for loosely instantiating an entity, in the following sense: the bound parameters of the entity, as indicated by the given list of variables, allows to explore all possible values, from the domain of their type (which may be any subtype of message). An optional guard, which may refer to the variables listed, constrains the selection. This mechanism is typically used to produce so-called *symbolic sessions*, where the bound variables range over type agent, such that (unless further constraints exist) their values include i, the name of the intruder.

3.11 Guards

Formulas used as conditions in if, while, select, and symbolic entity instantiation statements are called *guards*. They may refer to the current state of the system via variables and facts, for example peer(?A)=B & dishonest(?A). The "?" symbol indicates implicit logical quantification and may yield to implicit variable assignment.

At pattern positions where the value to be matched is irrelevant, one may write a singleton "?", without giving a variable name, which is treated as a 'wildcard'.

For each variable name occurring in a guard, either all occurrences must be preceded by "?" or appear without a "?". In the following, we refer to a variable that has a "?" before its name as "?"-*variable*.

Due to the "natural" semantics of the underlying ASLan transition rules, a guard is translated by computing its Disjunctive Normal Form (DNF) and treating each disjunct as a separate rule condition. In order to satisfy the overall guard, any of its DNF disjuncts may be chosen non-deterministically as long as it is satisfiable as defined below.

When a disjunct of a guard is evaluated, variables are handled as follows.

- A variable name without “`?`” leads to matching according to the current value of the variable.

- A “`?`” not followed by a variable name matches any value. Logically speaking, it is an existentially bound variable, which cannot be referred to elsewhere.

- A “`?`”-variable that occurs at least once positively within the disjunct is treated as implicitly existentially quantified. Thus the above example guard is logically treated as `exists A. (peer(A)=B & dishonest(A))`. If the disjunct is chosen, each such variable is non-deterministically assigned a suitable value such that (after consistently replacing all occurrences of this variable with the chosen value) the overall condition of disjunct is satisfied.

- A “`?`”-variable that occurs only negatively is treated as existentially quantified under the negation within the disjunct. For example, `p(?M) & !q(?M,?N)` is treated as `exists M. (p(M) & !(exists N. q(M,N)))`. For variables like `?N`, no assignment occurs when choosing the given disjunct. Therefore, `N` cannot be referred to later on.

If there is more than one satisfiable disjunct or for the same disjunct different variable assignment solutions exist, then each of them is a nondeterministically possible successor state. Note that depending of the structure of the DNF, it may happen that a “`?`”-variable is assigned a value in some but not all situations. In this case, the translator issues a warning.

To avoid interference of implicit quantification with implicit negation of guards in `if` and `while` statements due to their — implicit or explicit — `else` branch, there is a syntactic restriction on “`?`”-variables (but not on singleton “`?`”s). It applies after constructing the disjunctive normal form (DNF) of the guard, which among others expands implications of the form “`P => Q`” as “`!P | Q`”:
For each disjunct of the DNF of a guard in `if` and `while` statements, any “`?`”-variable occurring positively in a conjunct must not occur in other conjuncts of the same disjunct (while single occurrences of such a variable in different disjuncts are fine). This rules out the occurrence of variables like, e.g., `?X` in `if (fact1(?X) & fact2(?X,Y))` because their negation in the `else` branch would require a universal quantifier ranging over a disjunction, namely

```
forall X. !fact1(X) | !fact2(X,Y)
```

If needed, the condition can be re-phrased using an auxiliary Horn clause, e.g.

```
aux_fact12(X,Y):  fact12(X,Y) :- fact1(X) & fact2(X,Y);
```

Alternatively, one can use the `select` statement instead of the `if` statement above, in case the `else` branch is immaterial.

3.12 Grammar in EBNF

The (more or less) context-free aspects of the ASLan++ grammar are defined using an Extended Backus-Naur Form (EBNF), where `(X #Y)+` stands for one or more occurrences of X, separated by Y, e.g. (Term #",")+ can be expanded to Term, Term, Term. Context-dependent restrictions are stated to the right of the respective BNF rule, as informal text that is preceded by the “%” symbols.

```
MainSpec    ::= "specification" Name
                "channel_model" ("CCM" | "ICM" | "ACM")
                EntityDecl
Module      ::= EntityDecl
EntityDecl  ::= "entity" UpperName Params? "{" Imports?
                 Decls* EntityDecl* Body? ConstrDecls? GoalDecls? "}"

UpperLetter ::= ["A" .. "Z"]
LowerLetter ::= ["a" .. "z"]
NonZeroDigit::= ["1" .. "9"]
Digit       ::= "0" | NonZeroDigit
Numeral     ::= "0" | NonZeroDigit Digit*
Alphanum    ::= UpperLetter | LowerLetter | Digit | "_" | "’"
UpperName   ::= UpperLetter Alphanum*
LowerName   ::= LowerLetter Alphanum*
Name        ::= LowerName | UpperName

Var         ::= UpperName
Vars        ::= (Var #",")+
Params      ::= "(" ((Vars ":" Type) #",")+ ")"

Imports     ::= "import" (UpperName #",")+

Decls       ::= TypeDecls | SymbolDecls | MacroDecls | ClauseDecls
              | EquationDecls

TypeDecls   ::= "types" (TypeDecl ";")+
TypeDecl    ::= LowerName ("<" LowerName )? % note the optional supertype
Type        ::= LowerName        % simple type name
              | Type "set"       % note the actual type parameter
              | Type ("*" Type)+ % tuple
              | "(" Type ("," Type)+ ")"                % tuple compound
              | (Type "->")? LowerName ("(" Types ")")? % general compound
              | Type ("." Type)+                        % concatenation compound
              | "(" Type ")"
Types       ::= (Type #",")+

SymbolDecls ::= "symbols" (SymbolDecl ";")+
SymbolDecl  ::= Vars ":" Type
              | "nonpublic"? (LowerName #",")+ ":" Type        % constants
              | "nonpublic"? "noninvertible"?
                              LowerName "(" Types ")" ":" Type % function symbol

MacroDecls  ::= "macros" (MacroDecl ";")+
MacroDecl   ::= (Var "->")? LowerName ("(" Vars ")")? "=" Term

ClauseDecls ::= "clauses" (ClauseDecl ";")+
ClauseDecl  ::= LowerName ("(" Vars ")")? ":"  % name and parameters
```

```
                ("forall" Var+ ".")? Term (":-" (Condition #"&")+)?
Condition ::=  Term | Term "=" Term | Term "!=" Term | "!" Condition

EquationDecls ::= "equations" (EquationDecl ";")+  %% only allowed in the root entity
EquationDecl  ::= Term "=" Term

Term        ::= LowerName
              | Numeral
              | Var
              | "?"Var  %% only allowed within guards
              | "?"     %% only allowed within guards
              | "[" "Actor" "]" | "[" Term "]_[" Term "]"  % pseudonym
              | FuncApp
              | Term ("." Term)+              % message concatenation
              | "(" Term "," Terms ")"        % tuple
              | "{" Terms? "}"  % set literal %% only allowed in variable assignment
              | "{|" Term "|}_" Term          % symmetric encryption
              | "{" Term "}_" Term            % asymmetric encryption
              | "{" Term "}_inv" Term         % digital signature
              | "(" Term ")"
              | Name ":(" Term ")"            % term annotated with goal label
Terms       ::= (Term #",")+
FuncApp     ::= (Term "->")? LowerName ("(" Terms ")")? %% transmission not allowed here
Body        ::= BreakDecl? "body" Stmt
BreakDecl   ::= "breakpoints" "{" ( LowerName #"," )+ "}"


Guard       ::= FuncApp
              | Transmission %% sending transmission not allowed here
              | "!" Guard
              | Term "!=" Term
              | Term "=" Term
              | Guard ("&" | "|" | "=>") Guard
              | "(" Guard ")"
          %% set literals are not allowed in guards
GuardNoRcv  ::= Guard %% any Guard but no transmissions allowed here

Stmt        ::= Assign
              | FreshGen
              | EntityInst
              | SymbEntityInst   % symbolic session
              | Transmission ";"
              | IntroduceFact
              | RetractFact
              | Branch
              | Loop
              | Select
              | Assert
              | "{" Stmt* "}"

Assign        ::= (Var | Name ":(" Var ")" ) ":=" Term ";"
FreshGen      ::= (Var | Name ":(" Var ")" ) ":=" "fresh()" ";"
```

```
EntityInst    ::= "new" Entity ";"
SymbEntityInst::= "any" (Var+ ".")? Entity ("where" Guard)? ";"
Transmission  ::= (Term "->")? ("send" | "receive") "(" Terms ")" ("over" Term)?
          % where "Actor ->" is optional
              | Term ChannelProps Term ":" Term ";"  % annotated channels for CCM and ICM
              | Term "-" Term "->" Term ":" Term ";" % annotated channels for ACM
          %% set literals are not allowed in transmission payloads
ChannelProps  ::= "*"? ("->" | "->>") "*"?

Entity       ::= UpperName ("(" Terms ")")?
IntroduceFact ::=         FuncApp ";" % only for "iknows" and facts not in any clause head
RetractFact ::= "retract" FuncApp ";" % only for "iknows" and facts not in any clause head
Branch       ::= "if" "(" GuardNoRcv ")" Stmt ("else" Stmt)? ";"
Loop         ::= "while" "(" GuardNoRcv ")" Stmt
Select       ::= "select" "{" ("on" "(" Guard ")" ":" Stmt)+ "}"
                                   % Guard may include receive
Assert       ::= "assert" GoalDecl ";"  %% set literals are not allowed here

ConstrDecls ::= "constraints" (ConstrDecl ";")+  %% only allowed in the root entity
ConstrDecl   ::= Name ":" Formula

GoalDecls    ::= "goals" (GoalDecl ";")+
GoalDecl     ::= Name ":" Formula | ChanGoal | SecrGoal
ChanGoal     ::= Name ":(_)" Term ChannelProps Term
SecrGoal     ::= Name ":(_)" "{" (Term #",")+ "}"
Formula      ::= FuncApp
               | "!" Formula
               | LTLOp1 "(" Formula ")"
               | LTLOp2 "(" Formula "," Formula ")"
               | Term "!=" Term
               | Term "=" Term
               | Formula ("&" | "|" | "=>") Formula
               | ("forall" | "exists") Var+ "." Formula
               | "(" Formula ")"
% neXt | Yesterday | Finally | Once | Globally | Historically
LTLOp1       ::= "X" | "Y" | "<>" | "<->" | "[]" | "[-]"
% Until | Release | Since
LTLOp2       ::= "U" | "R" | "S"
```

3.13 ASLan++ prelude

The syntax of ASLan++ gives the user some freedom in declaring new symbols such as functions and facts. For instance, one may declare an encryption function as

```
crypt(public_key,message): message
```

Their meaning may be specified using Horn clauses and equations. In fact, it is an important feature of the semantics of ASLan that we need only a few built-in symbols with a hard-wired meaning but can express the meaning of most symbols in ASLan itself. For instance, we use the symbol iknows to specify that the intruder knows a particular value (of type message or a subtype of it, or also of any set type), and to define the deductions of the intruder in the style of Dolev-Yao by a set of Horn clauses, e.g.

```
iknows(crypt(K,M)) :- iknows(K) & iknows(M)
iknows(M) :- iknows(crypt(K,M)) & iknows(inv(K))
```

There are two reasons, however, to fix the meaning of some symbols with a kind of "standard interpretation". First, some symbols like in the above examples have been consistently used over a long time with a fixed meaning and serious misunderstandings may occur if somebody attaches a different meaning to them. Second, validation tools may have specialized techniques for some symbols, such as intruder deduction with a predefined set of operators; in order to capture the subclass of models for which some tools are specialized, it is necessary to have fixed symbols.

For both these reasons, we have defined an *ASLan++ standard prelude* that contains standard definitions such as the above ones and that is considered imported by all ASLan++ specifications. The semantics section § 4 assumes the declaration of several standard symbols in the prelude, such as the type symbol agent.

The types and functions that all back-ends should support are collected in a module with the name Prelude, defined as follows.

```
entity Prelude {

  % very basic types and related symbols
  types
    fact;
    protocol_id;
    message;
    channel; % for ACM
    nat < message;
  symbols
    i,root: agent;
    nonpublic noninvertible dishonest(agent): fact;
    nonpublic noninvertible iknows(message): fact;
    % its argument may also be of any set type
    true, false: fact;
    0,1,2,3, ...: nat;
    succ(nat)   : nat; % successor function
    noninvertible leq(nat,nat):fact; % less-or-equal relation
    "."  (message, message): message; % non-associative concatenation
  % subtypes of message and related symbols
  types
    text          < message;
    agent         < message;
    symmetric_key < message;
    private_key   < message;
    public_key    < agent;
  symbols
    noninvertible hash   (message): message;
    noninvertible scrypt (symmetric_key, message): message;
    noninvertible crypt  (public_key, message): message;
    noninvertible sign   (private_key, message): message;

    noninvertible defaultPseudonym(agent,nat): public_key;
    nonpublic noninvertible inv(public_key): private_key;
    noninvertible pk(agent): public_key; % generic public key

  symbols % used for implementing CCM
    noninvertible ak(agent): public_key; % authentication
    noninvertible ck(agent): public_key; % confidentiality
    atag : slabel; % authentication  tag
    ctag : slabel; % confidentiality tag
    stag : slabel; % security (A+C)  tag

  symbols % used for implementing ICM
    nonpublic noninvertible athCh(agent, agent, message): fact
    % msg on an authentic channel
    nonpublic noninvertible cnfCh(       agent, message): fact
    % msg on a confidential channel
    nonpublic noninvertible secCh(agent, agent, message): fact
    % msg on a secure channel

  symbols % used for implementing ACM
    nonpublic noninvertible sent(agent, agent, agent, message, channel):
                fact;
    nonpublic noninvertible rcvd(       agent, agent, message, channel):
                fact;
    nonpublic noninvertible confidential_to  (channel,agent): fact;
    nonpublic noninvertible weakly_confidential (channel): fact;
    nonpublic noninvertible authentic_on (channel,agent): fact;
    nonpublic noninvertible weakly_authentic (channel): fact;
    nonpublic noninvertible resilient (channel): fact;
    nonpublic noninvertible link (channel,channel): fact;
    nonpublic noninvertible bilateral_conf_auth (channel,channel,agent,
                agent): fact;
    nonpublic noninvertible unilateral_conf_auth(channel,channel,
                agent): fact;

  macros
   {|M|}_K    = scrypt(K,M)
   {M}_K      = crypt(K,M)
   {M}_inv(K) = sign(inv(K),M)

  % parameterized tuple type and related symbols
  types
    "A * B" < message;
  symbols
    "( ,... )": "A*B"; % tuple constructor

  % paramerized type of sets and related symbols
  types
    "A set";
  symbols
    "{ ,... }": "A set";        % set literal
    contains ("A set","A"): fact;
  % "add" and "remove" are implemented by
  % introducing and retracting "contains" facts

  clauses
    true_holds:
      true;
    dishonest_intruder:
      dishonest(i);
    analysis_crypt(K,M):
      iknows(M) :- iknows(crypt(K,M))  & iknows(inv(K));
    analysis_scrypt(K,M):
      iknows(M) :- iknows(scrypt(K,M)) & iknows(K);
    analysis_sign(K,M):
      iknows(M) :- iknows(sign(inv(K),M)) & iknows(K);
} % end entity Prelude
```

The modifiers `private` and `noninvertible` are explained more formally in § 4.3; intuitively, they are used to override the default behavior of function symbols that terms can be arbitrarily composed (that is, functions are public) and decomposed (that is, functions are invertible) by the intruder. Note that cryptographic operators are usually public and non-invertible: there are special intruder deduction rules for their analysis. An alternative specification of analysis using algebraic equations is not considered here.

The agent constants `i` and `root` denote the intruder and the root entity instance, respectively. The term `pk(A)` is used to denote the canonical public key of an agent `A`. The type `public_key` is a subtype of type `agent` for convenience in models involving pseudonyms.

The term defaultPseudonym(A,IID) stands for the default pseudonym of the agent A and entity instance IID, i.e. when using pseudonymous channels without specifying a particular pseudonym. The use of the instance ID is simply to use a fresh pseudonym for each instance of an entity that one plays in.

Note that the Horn clause analysis_sign is precise only for RSA-like signatures where the signed part can be recovered by applying the public key. For other signature implementations, e.g. ElGamal, it is too pessimistic in the sense that it may lead to false attacks because in general one can not recover M from sign(inv(K),M). Moreover, in case M is the hash of the actual payload with suitable padding (e.g. following state-of-the-art standards like PKCS), the Horn clause is not helpful in the sense that knowing the value of M has no use apart from verifying the signature.

4 ASLan++ semantics

In this section, we give a procedure for translating ASLan++ specifications into ASLan specifications. This procedure therefore defines the semantics of ASLan++ in terms of ASLan, and serves as a basis for the implementation of the translator which we call the ASLan++ connector, described in [6, §4.1].

We provide a high-level overview of the translation procedure, which consists of a number of steps, each focusing on different aspects of the mapping from the feature-rich, process-based ASLan++ into the rewrite-based ASLan.

The first step of the translation takes care of file imports and macro unfolding and the like. This phase is known as the *preprocessing phase* and is described in § 4.1.

Next, we have the static part of the specification, that is, everything that is not code, is translated. This translation step covers entities (§ 4.2), types and symbols (§ 4.3), clauses (§ 4.4), equations (§ 4.5) and security goals (§ 4.12).

Some preliminaries for the translation of code sections are described in § 4.6 (for control flow) and in § 4.14 (for compression).

Then, the code sections are translated in three sequential phases:

- translation of statements and guards (described respectively in § 4.7 and § 4.8), where the rewrite rules for the ASLan specification are generated (although in a temporary form allowing for ASLan++ terms),
- translation of terms (§ 4.9), applied to the rules generated in the first phase, converting ASLan++ terms into ASLan ones, and
- replacement of step label terms with numbers (as described in § 4.13).

Finally, § 4.10 describes the translation of the encapsulating body section.

In order to decrease the number of transition rules that are generated by the translation procedure, the set of transition rules can be, optionally, optimized. § 4.15 describes the two levels of optimization that can be employed.

4.1 Preprocessing

As a first step, we consider a group of preliminary operations on the input specification, which do not generate ASLan specifications but rather modify the input specification to ease the translation procedure. These operations are:

Import of external modules. In the `import` section a modeler can specify any number of external entities (called *modules*) to be included into the current entity specification. Importing a module corresponds to merging the sections of the module with the analogous sections of the current entity specification. This step results in a single entity, without any remaining `import` section. Note that imported modules may contain imports themselves, treated recursively.

Global disambiguation of elements. Since entities may have nested sub-entities, local declarations of elements (e.g. symbols, sub-entities, types, goals, . . .) hide any homonymous elements of the same kind that are defined in outer entities. Moreover, sub-entities of a given entity may declare independent elements that happen to have the same name, but which should not conflict (or in other ways interfere) with each other. Because ASLan only supports a flat name space, each hidden or potentially conflicting element must be renamed (for instance, by prepending the name of the entity in which it is defined) in order to make them unique within the global scope of ASLan. Hence the element is known under the new disambiguated name for all further translation steps and at the ASLan level.

Expansion of macros. The specification is parsed and any term matching the LHS of a macro defined in the `macros` section is replaced by the corresponding RHS. After this step, the `macros` section can be removed.

Expansion of shorthands. All function applications in object-oriented style notation, namely of the form `t_1 -> func(t_2,..,t_n)`, are converted to the form `func(t_1,t_2,..,t_n)`. For the special cases of `send` and `receive` operations where just two arguments are given, i.e. the optional argument `Actor` has been omitted, `Actor` is inserted in this step as an additional first argument. In a similar way, all bullet-style transmission (i.e. annotated channel) facts are converted into the appropriate predicates as specified in Table 1.

4.2 Translation of Entities

For every entity (at any level of nesting) in the specification, we declare in the translation a fresh *state predicate*, in the ASLan `section signature`. The new predicate will be parameterized with respect to a *step label*, an *instance ID* to uniquely identify every instance, parameters and variables of the entity, and will yield a fact. Parameters are treated exactly like local variables, except that they are initialized at entity instantiation. The state predicate has multiple purposes:

- “record” each instance of the entity,
- express the control flow of the entity, by means of the step label,
- keep track of the local state of an instance (namely, the current value of its variables),

and will be used later in the generation of rewrite rules for the translation.

While such facts store the values of parameters and other variables *local* to the given instance of an entity, these may be accessed (read and written) by children entities as well. In these cases, we will refer to the *owner* of a variable as the parent/ancestor instance that declares this variable locally and therefore has the variable stored in its state fact.

By convention, the state predicate for an entity `E` has the name `state_E` and will always have the first three parameters of fixed types and meanings, as follows:

- The first parameter will be of type `agent` (or a subtype of it) and will represent `Actor`, i.e. the agent who is playing the role of the entity to which the state fact is associated. For the root entity instance, its value is `root`.
- The second parameter will be of type `nat` and will represent the instance ID, with implicit name `IID`. For the root entity instance, its value is `0`.
- The third parameter will be of type `nat` and will represent the step label. Its initial value is `sl_0` (which is later replaced by `1`, as defined in § 4.13).

Consequently, if the ASLan++ root entity has the name `Environment`, it the state predicate given in the initial ASLan state will be

```
state_Environment(root, 0, 1)
```

Any other parameters and local variables of the entity will be listed as parameters of the state fact after these special three parameters. If the entity has an `Actor` parameter, it will be assimilated with the first of the three special parameters just listed.

Example 1. Consider the following entity declaration for the *Bid Manager* in the *Public Bidding* model

```
entity BidM(Actor, BP: agent) {

  symbols
    M: message; % a variable
}
```

then the following state predicate is created

```
state_BidM: agent * nat * nat * agent * message -> fact
```

in `section signature` in the translation.

The arguments of the predicate are an agent name (for parameter `Actor`), a unique *instance* ID (of type `nat`) created at instantiation, a step label of type `nat`, an agent name (for parameter `BP`), and finally a message (for variable `M`).

At instantiation of the entity,

```
new BidM(bm, bp)
```

where terms are passed to the entity in place of its parameters, a new fact

```
state_BidM(bm, iid, sl_0, bp, dummy_message)
```

will be added to the state. From now on, this fact can be used to identify this particular entity (via its instance ID `iid`), the value of its variables (`M`, initially set to `dummy_message`, to indicate that it is uninitialized) and parameters (`Actor` and `BP`, here replaced by the terms `bm` and `bp`), and its execution progress (step label, `sl_0`). □

As the above example shows, this `state` fact stores all necessary information regarding the execution state of a particular instance of an entity in the system, namely its instance `ID`, its step label, and the value of all its variables (including also parameters). Yet, nested entities inherit these values from their ancestors, meaning that variables of an instance are visible (unless overridden) to all its descendants. Thus, when an entity E refers to a variable of its ancestor A, then it is A's state fact, rather than E's, that needs be used.

To this end, in the following we will speak of the *owner* of a variable, indicating the instance whose state fact contains the value of that variable. While identifying which entity a variable belongs to can be done in the translation phase, identifying the exact runtime instance depends on the execution of the system and can be done only by binding each instance to its ancestors.

Exploiting the restriction that entities can instantiate only direct sub-entities, we implement this binding as follows:

- At instantiation of an entity, let `iid` and `pid` be the IDs of the new instance and its parent, respectively. We introduce the (persistent) fact `child(pid,iid)`.
- When referring to an inherited variable owned by some entity `E`, we actually write both on the LHS and RHS of the generated rule the state fact of `E` with its run-time instance ID `OID` and a chain of predicates `child(OID,ID1)`, ..., `child(IDn,IID)` where `IID` is the ID of the entity instance being executed at this step, thus enforcing the binding. See example 3 for more details.

4.3 Translation of Types and Symbols

ASLan++ has several built-in types, constants and functions, as defined in § 3.13. These are immediately reflected at the ASLan level.

After having (in the preprocessing phase) renamed all homonymous elements so to have globally unique names, declarations of new types (both basic and compound) are immediately reflected in the ASLan file in `section typeSymbols` (modulo minor syntactic adjustments).

Symbols must be partitioned into variables, constants and functions, and then be transcribed to `section types` (the first two) and to `section signature` (the latter) in the ASLan specification. Their types are also immediately reflected, except in the case of parametric types (i.e. `tuple` and `set`), where a more involved translation is required, fully detailed in § 4.9.

Symbols can be declared using the modifiers `nonpublic` and `noninvertible`.

- If a symbol (which may be a constant) $f(\tau_1, \ldots, \tau_n) : \tau$, where all $\tau_1, \ldots, \tau_n, \tau$ are subtypes of `message` or of some set type, is not declared as nonpublic, it is considered public. The semantics of being public can be defined by the following clause:

  ```
  hc f_public(M1,..,Mn):
    iknows(f(M1,...,Mn)) :- iknows(M1), ..., iknows(Mn);
  ```

  where the M_i are variables of type τ_i, respectively.

- Similarly, if a function $f(\tau_1, \ldots, \tau_n) : \tau$ is not declared to be non-invertible, it is considered invertible in those arguments where both τ_i and τ are subtypes of `message` or of some set type. The semantics of being invertible in the i-th argument (for $i \in \{1, \ldots, n\}$) can be defined by the following rule:

  ```
  hc f_invertible_i(M1,..,Mn):
    iknows(Mi) :-  iknows(f(M1,...,Mn))
  ```

  where again the M_i are variables of type τ_i, respectively.

  Note that most functions are typically public, i.e. everybody can apply them to known terms. To be on the safe side, user-defined functions are by default invertible. This makes sense at least for message constructors, which can be seen as operators in the free term algebra. Few other functions are invertible, for instance `pair` and `inv` in the prelude. `inv` is an example of an invertible function that is not public.

4.4 Translation of Clauses

Although each clause is declared inside an entity, it is applicable globally because (for simplicity of the translation and due to bugs in CL-AtSe and OFMC when referring to state facts) it cannot directly refer to local variables (including parameters) of the entity or any enclosing ones.

Example 2. Take the entity declaration for the `BidM` above, extended with a clause asserting that the *bid manager* trusts the *bidding portal* on any message, as follows

```
entity BidM(Actor, BP: agent) {

  clauses bmTrustBpOnAnyMsg(X):
          trusts(Actor, BP, X) :- saidTo(BP, Actor, X);
}
```

This clause is basically transcribed to an ASLan specification as is:

```
hc bmTrustBpOnAnyMsg(Actor, BP, X):
  trusts(Actor, BP, X) :- saidTo(BP, Actor, X);
```

We assume a set `PolicyPredicates` that stores all implicit facts, i.e. `iknows` and all other predicate symbols appearing in the head of the clauses. This set serves in later phases of the translation to ensure that no implicit fact, except for `iknows`, is introduced explicitly and no implicit fact is retracted explicitly, cf. § A.2.

4.5 Translation of Equations

The syntax for equations in ASLan++ closely resembles the syntax for the equalities in ASLan, and their semantics are identical, therefore their translation will consist in minor syntactic adjustments (applying a procedure for conversion of terms that we will present in § 4.9) and transcription in the `section equations` of the file.

4.6 Representing the Control Flow

As anticipated in § 4.2 we represent the control flow of the entity via a step label stored in its state fact. We introduce here an abstract encoding for step labels which we will refer to in the next phases of the translation procedure. Note that step labels are symbolic and do not need to be numbers.

We assume a predefined step label `sl_0` standing for the default initial step label and the following functions, which are total, injective and have disjoint range:

- `succ(sl)`, returning the successor for `sl`, and
- `branch(sl, n)`, returning the nth branch for `sl`.

These two symbols are used only for presentation purposes. For a realistic ASLan++ model the number of applications of these predicates would be too high and would unnecessarily increase the complexity of the generated ASLan translation. Thus, based on the two predicates, numeric step numbers are generated and used in the translation. See § 4.13 for a description on how this is done.

4.7 Translation of Statements

We define here, in pseudo-code, a procedure `ParseCode` that recursively parses a list of statements and generates equivalent rewrite rules as output. Its arguments are the list of statements `Stmts` to parse, the current step label `sl` and a `return` step label (for returning from a branch).

For ease of understanding, the procedure will call different sub-procedures according to the kind of the statement examined, each of which is treated in a subparagraph.

```
ParseCode(Stmts, sl, return_sl)

  if (Stmts = stmt.rest) { % Stmts not empty
                           % stmt: first statement,
                           % rest: remaining statements

    case stmt {
     - assign      : Assign(stmt, rest, sl, return_sl)
     - freshGen    : FreshGen(stmt, rest, sl, return_sl)
     - entityGen   : EntityGen(stmt, rest, sl, return_sl)
     - symbEntGen  : SymbEntGen(stmt, rest, sl, return_sl)
     - transmission: Transmission(stmt, rest, sl, return_sl)
     - introduceFact : IntroduceFact(stmt, rest, sl, return_sl)
     - retractFact   : RetractFact(stmt, rest, sl, return_sl)
     - branch      : Branch(stmt, rest, sl, return_sl)
     - loop        : Loop(stmt, rest, sl, return_sl)
     - select      : Select(stmt, rest, sl, return_sl)
     - assert      : Assert(stmt, rest, sl, return_sl)
     - grouping    : Grouping(stmt, rest, sl, return_sl)
    }
  } else {                 % No statement left to parse
    if (return_sl != null) {
      LHS = state fact for this entity, with step label sl
      RHS = state fact for this entity, with step label return_sl

      add
          LHS "=>" RHS
      to "section rules" in the translation
    }
  }
```

The procedure analyzes the first statement in the list given, calling the appropriate sub-procedure according to the statement's type. When no statements are left to parse, either the execution of the main thread is finished or that of a branch is. In the latter case, a *return step label* is provided, and a new idle rule will be added to the translation to redirect the control flow to the former thread's execution.

In the following sub-paragraphs, we will explain every sub-procedure individually, and use the first instruction `stmt = form` to describe the syntactic form of statement `stmt`.

Furthermore, it is necessary that in translating the guards, facts from the state are not removed; therefore we use a function `renewPositiveFactsIn` that renews the positive explicit facts in the clause. Implicit facts are not reintroduced on the right-hand side of the corresponding rewrite rule because implicit facts (stored in the set `PolicyPredicates`, see § 4.4) must not appear on the RHS of rewrite rules. The function `renewPositiveFactsIn` is used in § 4.7.5, § 4.7.9, § 4.7.10, etc.

4.7.1 Grouping

```
Grouping(stmt, rest, sl, return_sl) {

  stmt = "{" InnerStmts "}"

  ParseCode(InnerStmts.rest, sl, return_sl)
}
```

In the case of a series of statements grouped by brackets, the internal statements are prepended to the remaining statements, and the parsing is resumed on this new list of instructions.

4.7.2 Variable assignment

```
Assign(stmt, rest, sl, return_sl) {
  stmt = Var ":=" Term

  LHS = state fact for this entity, with step label sl.
        state fact for the owner of Var
  RHS = state fact for this entity, with step label succ(sl).
        state fact for the owner of Var with Var set to Term

  add
      LHS "=>" RHS
  to "section rules" in the translation

  ParseCode(rest, succ(sl), return_sl)
}
```

In case of assignment to a variable Var, we create a rewrite rule as given above, with one exception. Since Var may belong to an ancestor of the entity, we need to change its state fact rather than the current entity's (for which we will update the step label nonetheless). Yet in case Var belongs to the current entity, the two state facts in the LHS are collapsed into one, and there is only one state fact on the RHS that combines the change to the step label with the assignment to the variable. A similar strategy is always followed in order to avoid unintended duplication of state facts on the right-hand side of a rule, e.g. when multiple variables (and possibly step numbers) on the right-hand side of a rule need to be updated.

Example 3. Let us extend further the BidM declaration above, with a variable assignment, as follows

```
M := crypt(pk(BP),m);
```

The variable assignment can be expressed via a rule as the following one, which replaces M with the term assigned to it:

```
state_BidM(Actor,IID,sl,BP,M)
  =>
state_BidM(Actor,IID,succ(sl),BP,crypt(pk(BP),m))
```

Note that this is the case for assignment to a variable M local to the entity BidM, whereas if it was another variable E belonging to the parent entity of BidM, e.g. Env, the resulting rewrite rule would be

```
state_BidM(Actor,IID,sl,BP,M).
  state_Env(..,OID,..,E,..).
  child(OID,IID)
  =>
state_BidM(Actor,IID,succ(sl),BP,M).
  state_Env(..,OID,..,crypt(pk(BP),m),..).
  child(OID,IID)
```

□

4.7.3 Generation of fresh values

```
FreshGen(stmt, rest, sl, return_sl) {

  stmt = Var ":=" "fresh()"

  N is fresh variable name
  declare N of same type as Var in "section types" in
  the translation

  LHS = state fact for this entity, with step label sl.
        state fact for owner of Var
  RHS = state fact for this entity, with step label succ(sl).
        state fact for owner of Var with Var set to N

  add
      LHS =[exists N]=> RHS
  to "section rules" in the translation

  ParseCode(rest, succ(sl), return_sl)
}
```

Generation of a fresh value is straightforward: transparently to the modeler, a new variable `N` is created, and a rewrite rule is added that substitutes `Var` in its owner's state fact with `N`, instantiated by the `exists` of the rewrite rule, and advances the step label.

Like for regular assignments, since `Var` may belong to an ancestor of the entity, we need to change its state fact rather than the current entity's (for which we will update the `step label` nonetheless). Yet, in case `Var` belongs to the current entity, the two state facts in the LHS are collapsed into one, and there is only one state fact on the RHS that combines the changes to the step label with the assignment to the variable.

Example 4. Let's extend the above Bid Manager specification by adding in its body the following instruction to instantiate variable `M` of type `message` with a fresh value of agreeing type

```
M := fresh();
```

then we create a new variable `M_1` in the translation, and express the fresh generation via the rule

```
state_BidM(Actor,IID,sl,BP,M)
  =[exists M_1]=>
state_BidM(Actor,IID,succ(sl),BP,M_1)
```

As for the variable assignment case, if the generated value is assigned to a variable belonging to an ancestor, the state fact of the latter must be changed instead. □

4.7.4 Entity instantiation

```
EntityGen(stmt, rest, sl, return_sl) {

  stmt = "new" entity "(" t_1 "," .. "," t_n ")"

  IID = fresh variable

  declare IID of type nat in "section types" in
  the translation

  LHS = state fact for the current entity, with step label sl.
        state facts for the owners of variables appearing in
          t_1,..,t_n
  RHS = state fact for the current entity, with step label
          succ(sl).
        state facts for the owners of variables appearing in
          t_1,..,t_n.
        state fact for new entity instance, such that
         - its step label is sl_0
         - its instance ID is IID
         - all parameters p_1,..,p_n set to t_1,..,t_n
         - all variables v_1,..,v_m set to "dummy" of the right type

  add
      LHS =[exists IID]=> RHS
  to "section rules" in the translation

  ParseCode(rest, succ(sl), return_sl)
}
```

Conceptually, instantiating an entity corresponds to creating a new state fact for the instance, running from this point on in parallel to all other entity instances in the system.

To this end, we create a rule that adds to the current state the state fact for the new instance, with a fresh instance ID to make it distinct from any other already in the system and step label `sl_0` from which execution of its body can start. Its parameters `p_1,..,p_n` are assigned the terms passed, namely `t_1,..,t_n`, while its internal variables `v_1,..,v_m` are assigned the value `dummy_T` standing for an *uninitialized* value of the respective type `T`.

Example 5. Imagine that during the parsing of the entity `Env`, at step label `sle`, the following instantiation for `BidM` is encountered

```
new BidM(bm,bp);
```

The rule generated will be

```
state_Env(...,OID,sle,..)
  =[exists IID]=>
state_Env(...,OID,succ(sle),..).
  state_BidM(bm,IID,sl_0,bp,dummy_message).
  child(OID,IID)
```

that will create the fact for the `BidM` instance, with step label `sl_0`. □

Recall that, as we remarked in § 3.3 above, we assume that the modeler is responsible to ensure, for each entity that may also be "played" by a dishonest party, that with the sufficient knowledge of secrets, the intruder has enough information to perform every legal step of the entity, so that the behavior of that entity is subsumed by what the intruder can do. Therefore, we can block, as described in § 4.10, any entity instances with a dishonest player, i.e. where `dishonest(Actor)`. This is safe because all transitions that the dishonest agent can perform are covered by the intruder's general ability (expressed by the built-in Dolev-Yao behavior), and we therefore spare the tools the explicit *transition rules* associated with executing "honest" steps of the entity description for a dishonest agent.

4.7.5 Symbolic entity instantiation

```
SymbEntityGen(stmt, rest, sl, return_sl) {

  stmt = "any" A_1 .. A_m "."
           entity "(" t_1 "," .. "," t_n ")" "where" Guard

  g_1..g_k = positiveGuards(Guard)
  apply adaptGuard to g_1..g_k

  IID = fresh variable

  declare IID of type nat
      and a_1 .. a_m of suitable type
  in "section types" in the translation

  for (i from 1 to k) {

    LHS = state fact for the current entity, with step label sl.
          state facts for the owners of variables appearing in
            t_1,..,t_n.
          state facts for the owners of variables appearing in g_i
    RHS = state fact for the current entity, with step label succ(sl).
          state facts for the owners of variables appearing in
            t_1,..,t_n.
          state facts for the owners of variables appearing in g_i
          state fact for new entity instance, such that
           - its step label is sl_0
           - its instance ID is IID
           - all parameters p_1,..,p_n set to t_1,..,t_n
           - all variables v_1,..,v_m set to "dummy" of the right type

    for (j from 1 to m) {
      LHS = LHS.iknows(V_j)
      RHS = RHS.iknows(V_j)
    }

    add
        LHS.g_i =[exists IID]=> RHS.renewPositiveFactsIn(g_i)
    to "section rules" in the translation
  }

  ParseCode(rest, succ(sl), return_sl)
}
```

A *symbolic* entity instantiation consists in a partially specified instantiation of an entity. Its purpose is modelling a whole class of instances, i.e. for all possible instantiations of the bound variables A_1,..,A_m with constants taken from the domain. Usually, these variables and constants are of type agent, but it is only required that each of their type is a subtype of message.

The instruction is translated to a rule that introduces the new entity instance on the RHS. The variables A_1,..,A_m will be bound to ground constants by the predicates iknows(A_1)..iknows(A_m) appearing in the LHS of the rule (and also renewed in the RHS). The binding allows to explore all possible instantiation of the variables A_1,..,A_m with concrete values. To this end, m new constants are declared, of the types determined from from the respective parameters of the entity.

In addition to this, the rule will also enforce that the chosen values for A_1,..,A_m satisfy Guard, whose treatment, using the auxiliary function positiveGuards, we will introduce in detail in § 4.7.9.

4.7.6 Send, receive, and channel models

```
Transmission(stmt, rest, sl, return_sl) {

  if stmt = "receive" params
  then
    Select("select { on (" stmt "): {}}", rest, sl, return_sl)
  else  % stmt = "send" params
    IntroduceFact(stmt, rest, sl, return_sl)
}
```

An independent receive statement `R`, i.e. a receive operation that has not been specified as a guard in an `on` part of a `select` statement, is translated at first as if it appeared in `select { on(R): {}}` (see § 4.7.11 for details).

A `send` statement is translated at first like a fact introduction (see § 4.7.7 for details).

As described in detail in Deliverable D3.3 [2], we have formalized different models of communication and definitions of channels as assumptions or goals, respectively. In particular, for channels as assumptions

- The *Cryptographic Channel Model CCM*: here, channels are realized by passing messages via the intruder, where security is achieved by certain ways of encrypting and signing the messages.

- The *Ideal Channel Model ICM*: here, we have abstract fact symbols and special transition rules that model the intruder's limited ability to send and receive on those channels. (For instance, he can see everything that is transmitted on an authentic channel, but he can only send under any identity that he controls.)

- The *Abstract Channel Model ACM*: is at a similar abstraction level as ICM, but here, the notion of a channel is per "session" between two parties. It uses named channels and explicit send and receive events that are constrained by LTL formulas over the traces.

We have shown in [2,18] that CCM and ICM are equivalent under certain realistic assumptions, and we are working to obtain a similar result between ICM and ACM. Establishing such equivalence results is fundamental as they allow us to use each model interchangeably, according to what fits best with certain analysis methods. In this regard, each of these models has its strengths:

- The CCM allows one to model channels within tools that do not have support for channels, because it requires only the standard cryptographic primitives and the intruder deduction machinery that is integrated in all the back-ends of the AVISPA Tool that are providing a basis for the AVANTSSAR platform. Also, it allows for using the optimization that an insecure network and the intruder can be identified, i.e. we have a compressed transition where the intruder sends a message that is received and answered to by the receiver, and the intruder immediately intercepts this answer.

- The ICM is more helpful in a different class of tools where the number of transitions is less problematic, but the complexity of terms is an issue. Also, it is the abstract reference model for our compositionality results between channels as assumptions and channels as goals.

- The ACM, finally, allows for arbitrary LTL constraints on the sending and receiving, e.g. resilient channels (i.e. every message is eventually received). Thus, the assumed security properties of the channels are expressed abstractly, and — differently from CCM — they do not need to be defined via concrete implementations.

Translation of channel goals is discussed in § 4.12.2. Here, we focus on the translation of channels when used as an assumption, which depends on the choice of the channel model as we discussed in detail in D3.3 [2]. In this case, we take into account the `send` and `receive` predicates generated after the preprocessing phase described in § 4.1. In order to make explicit that both the translation and the interpretation of such predicates can be different according to the channel model chosen, let us illustrate the translation(s) at hand of a concrete example (we proceed analogously for the other cases).

Each model introduces a number of symbols, facts, and rules necessary to express the different channels as defined in D3.3. Note that freshness is not supported for CCM/ICM channels as assumptions, while for ACM it is implied by confidentiality and weak confidentiality.

Assume that the translation process has proceeded up to the point of producing a transition rule that includes an agent receiving a message `M1` on a confidential channel from `A` and sending a message `M2` to `B` on an authentic channel. For CCM/ICM let us further assume that the agent sends `M2` under pseudonym `P`:

```
 A            ->* Actor: M1;
[Actor]_[P] *->  B     : M2;
```

while for ACM let us further assume that the reception is on channel `ch1` and transmission is on channel `ch2`:

```
ch1->confidential_to(Actor);
ch2->    authentic_on(Actor);
...
A     -ch1-> Actor: M1;
Actor -ch2-> B    : M2;
```

The final translation into ASLan in the three models is as follows.

CCM:

```
iknows(crypt(ck(Actor),ctag.M1)).L
=[V]=>
R.iknows(sign(inv(P),atag.B.M2)))
```

Here, `ck` is the confidentiality key of the receiver, and `ctag` and `atag` are tags describing confidentiality and authenticity, respectively. The agent name `B` given for authentic transmissions after the `atag` identifies the intended receiver, which captures directedness for *authentic* CCM channels.

ICM:

```
cnfCh(Actor,M1).L
=[V]=>
R.athCh(P,B,M2)
```

ASLan++ predicate	ASLan translation for CCM
`A->send    (B,M)`	`iknows(M')`
`A->receive(B,M)`	`iknows(M')`
`A->send    (B,M) over authCh`	`iknows(sign (akP(A'),atag.B'.M'))`
`A->receive(B,M) over authCh`	`iknows(sign (akP(B'),atag.A'.M'))`
`A->send    (B,M) over uauthCh`	`iknows(sign (akP(A'),atag.   M'))`
`A->receive(B,M) over uauthCh`	`iknows(sign (akP(B'),atag.   M'))`
`A->send    (B,M) over confCh`	`iknows(crypt(ckP(B'),ctag.   M'))`
`A->receive(B,M) over confCh`	`iknows(crypt(ckP(A'),ctag.   M'))`

`A->send    (B,M) over  secCh`	`iknows(crypt(ckP(B'),`
	`sign (akP(A'),stag.B'.M')))`

`A->receive(B,M) over  secCh`	`iknows(crypt(ckP(A'),`
	`sign (akP(B'),stag.A'.M')))`

Table 8: Translation from ASLan++ to the CCM

Here, `cnfCh` indicates a confidential channel, where its first argument is the receiver. `athCh` indicates an authentic channel, where its first argument is the sender and the second argument identifies the intended receiver, which captures directedness.

ACM:

```
rcvd(Actor,A,M1,ch1).L
=[V]=>
R.sent(Actor,Actor,B,M2,ch2)
```

Here the fact `ch1->confidential_to(Actor)`, given earlier, captures confidentiality to the receiver, while `ch2->authentic_on(Actor)` captures authentiticy of the sender and the third argument of the `sent` fact identifies the intended receiver, which captures directedness.

In the following paragraphs, one for each channel model, the translation of each channel security property is listed in detail.

CCM. Table 8 shows the translation of send and receive terms, obtained by replacing the predicates with the corresponding encoding for the CCM.

In the translation table, the following auxiliary symbols are used, where the type `slabel` stands for "security label".

```
atag : slabel; % authentication  tag
ctag : slabel; % confidentiality tag
stag : slabel; % security (A+C)  tag
```

```
iknows : message -> fact       % send/receive on CCM (or unprotected ICM) channel
ak     : agent -> public_key % noninvertible authentication  key function
ck     : agent -> public_key % noninvertible confidentiality key function
```

ASLan++ predicate	ASLan translation for ICM
`A->send    (B,M)`	`iknows(M')`
`A->receive(B,M)`	`iknows(M')`
`A->send    (B,M) over authCh`	`athCh(A',B',M')`
`A->receive(B,M) over authCh`	`athCh(B',A',M')`
`A->send    (B,M) over confCh`	`cnfCh(B'   ,M')`
`A->receive(B,M) over confCh`	`cnfCh(A'   ,M')`
`A->send    (B,M) over  secCh`	`secCh(A',B',M')`
`A->receive(B,M) over  secCh`	`secCh(B',A',M')`

Table 9: Translation from ASLan++ to the ICM

In the ASLan++ term pattern in the left-hand column of the table, A stands for Actor or any pseudonym of it, i.e. [Actor] or [Actor]_[P] for any P. B stands for an agent different from Actor or any pseudonym of it, i.e. [B]_[P] for any P. In the translation pattern in the right-hand column,

- akP(X) stands for inv(X) if X is a pseudonym and inv(ak(X)) otherwise,
- ckP(X) stands for X if X is a pseudonym and ck(X) otherwise,
- A' stands for the translation of A,
- B' stands for the translation of B, and
- M' stands for the translation of M,

where the translation of the agent, pseudonym, and message subterms is as defined in § 4.9.

In cases where a pseudonym P and is in the *sending* role, i.e. for transmissions of the form P->send(B,M) ... (or the equivalent P ...->... B: M), and the form B->receive(P,M) ... (or the equivalent P ...->... B: M), the value of the pseudonym is additionally transmitted in unprotected fashion. For instance, [Actor] *->* B : M gets translated to

```
iknows(pair(defaultPseudonym(Actor, IID),
            crypt(pk(B), sign(inv(defaultPseudonym(Actor, IID),
                             pair(stag, pair(B, M))))))
```

The transmission of the pseudonym is needed so that the receiver can send an answer or recognize different messages sent by the same pseudonymous sender.

ICM. Table 9 shows the translation of send and receive terms, obtained by replacing the predicates with the corresponding encoding for the ICM.

Like for CCM, the subterms A', B', and M' stand for the translation of A, B, and M, respectively, including any pseudonyms. Also the pseudonyms in the sending role are transmitted in an analogous way.

In the translation table, the following auxiliary symbols are used:

ASLan++ predicate	ASLan translation for ACM
`A->send    (B,M) over Ch`	`sent(A',A',B',M',Ch')`
`A->receive(B,M) over Ch`	`rcvd(   A',B',M',Ch')`

Table 10: Translation from ASLan++ to the ACM

```
iknows :                 message -> fact % message on insecure channel
athCh  : agent * agent * message -> fact % message on authentic channel
cnfCh  :         agent * message -> fact % message on confidential channel
secCh  : agent * agent * message -> fact % message on secure channel
```

As specified in [2, §4.1.4: Standard Channels as Assumptions], and [18, 20], where more details on these symbols may be found, the following intruder rules are given (at ASLan level) for sending and receiving on ICM channels:

$$\begin{aligned}
\mathtt{iknows}(B).\mathtt{iknows}(M).\mathsf{dishonest}(A) &\Rightarrow \mathtt{athCh}(A,B,M)\\
\mathtt{athCh}(A,B,M) &\Rightarrow \mathtt{iknows}(M)\\
\mathtt{iknows}(B).\mathtt{iknows}(M) &\Rightarrow \mathtt{cnfCh}(B,M)\\
\mathtt{cnfCh}(B,M).\mathsf{dishonest}(B) &\Rightarrow \mathtt{iknows}(M)\\
\mathtt{iknows}(B).\mathtt{iknows}(M).\mathsf{dishonest}(A) &\Rightarrow \mathtt{secCh}(A,B,M)\\
\mathtt{secCh}(A,B,M).\mathsf{dishonest}(B) &\Rightarrow \mathtt{iknows}(M)
\end{aligned}$$

ACM. Table 10 shows the translation of send and receive terms, obtained by replacing the predicates with the corresponding encoding for the ACM.

Similarly to CCM, the subterms `A'`, `B'`, `M'`, and `Ch'` stand for the translation of `A`, `B`, `M`, and `Ch`, respectively. Note that pseudonyms are not used for ACM, such that in all cases, `A` is identical to `Actor`.

In the translation table, the following auxiliary symbols are used, where the type `channel` stands for an ACM channel.

```
sent: agent * agent * agent * message * channel -> fact
rcvd:         agent * agent * message * channel -> fact
```

Reception and sending of messages are captured as two separated actions. This allows for a finer-grained model where resilient channels can be captured. In particular, the following rule is added to section `rules` in the translation:

```
step deliver(RS,OS,R,M,Ch):=
sent(RS,OS,R,M,Ch)
=>
rcvd(R,OS,M,Ch)
```

The first argument of sent is the real name of the sender RS, whereas the second argument is the official sender OS, which is relevant for the receiver, and might be faked/manipulated by the intruder as given in the rules below. The third argument of sent identifies the intended receiver R. The first argument of rcvd is the real name of the receiver R, whereas the second argument is the official sender. The remaining two arguments are the message M transmitted and the name of the channel Ch, respectively. Note that this rule entails directedness: the only (honest) receiver of a message is the intended one.

The abilities of the intruder are augmented by the following rules:

```
step fake(OS,R,M,Ch):=
iknows(M). iknows(OS). iknows(R)
 =>
iknows(M). iknows(OS). iknows(R).
sent(i,OS,R,M,Ch)

step intercept(A,R,M,Ch):=
sent(A,A,R,M,Ch)
 =>
rcvd(i,A,M,Ch). iknows(M)

step overhear(A,R,M,Ch):=
sent(A,A,R,M,Ch)
 =>
sent(A,A,R,M,Ch).
rcvd(i,A,M,Ch). iknows(M)
```

For instance, the first rule models the intruder that exploits its knowledge to impersonate an agent OS in sending a message M to an agent R on a communication channel Ch.

The facts used at the ASLan++ level (cf. Table 3, Table 4, and Table 5) to state assumed channel properties are kept at the ASLan level such that the backends can use them as triggers for LTL constraints, described next.

The condition that a channel Ch is confidential to a principal p can be formalized by the following LTL formula:

```
confidential_to(Ch,p) :=
forall A. forall B. forall M.
 G(implies(rcvd(B,A,M,Ch),equal(B,p)))
```

In our framework this boils down to adding the following constraint in the ConstraintsSection of the ASLan prelude (see § A.2.6):

```
forall Ch. forall P. forall A. forall B. forall M.
 G(implies(confidential_to(Ch,P),
  G(implies(rcvd(B,A,M,Ch),equal(B,P)))))
```

The other definitions are treated in a similar way.

Weak confidentiality is translated as the LTL constraint

```
forall Ch.
forall A . forall B . forall M .
forall A'. forall B'. forall M'.
 G(implies(weakly_confidential(Ch),
  G(implies(and(rcvd(A,B,M,Ch), F(rcvd(A',B',M',Ch))),
            equal(A,A')))))
```

A channel provides authenticity if its input is exclusively accessible to a specified sender. Authenticity is translated as the LTL constraint

```
forall Ch. forall P. forall RS. forall A. forall B. forall M.
 G(implies(authentic_on(Ch,P),
  G(implies(sent(RS,A,B,M,Ch),
          and(equal(A,P), equal(RS,P))))))
```

Weak authenticity is translated as the LTL constraint

```
forall Ch.
forall RS . forall A . forall B . forall M.
forall RS'. forall A'. forall B'. forall M'.
 G(implies(weakly_authentic(Ch),
  G(implies(and(sent(RS,A,B,M,Ch), F(sent(RS',A',B',M',Ch))),
          and(equal(A,A'), equal(RS,RS'))))))
```

In ACM, resilience amounts to requiring that every message sent over the channel will be eventually delivered to the intended recipient. This is translated as the LTL constraint

```
forall Ch. forall RS. forall A. forall B. forall M.
 G(implies(resilient(Ch),
  G(implies(sent(RS,A,B,M,Ch),
          F(rcvd(B,A,M,Ch))))))
```

The fact that the principal sending messages on `Ch1` is the same principal that receives messages from `Ch2` is captured by the `link(Ch1, Ch2)` property, which is translated as the LTL constraint

```
forall Ch1. forall Ch2.
forall RS. forall A. forall B . forall M.
forall R .           forall B'. forall M'.
```

```
 G(implies(link(Ch1,Ch2),
  G(implies(and(F(sent(RS,A,B,M,Ch1)), F(rcvd(R,B',M',Ch2))),
            equal(RS,R)))))
```

The relation between two channels Ch1 and Ch2 capturing a bilateral confidential and authentic communication between A and B, is expressed by the fact bilateral_conf_auth(Ch1,Ch2,A,B). This is implemented as shorthand that the translator expands on introduction to the facts:

```
  confidential_to(Ch1,B)
  authentic_on   (Ch1,A)
  confidential_to(Ch2,A)
  authentic_on   (Ch2,B)
```

Similarly, the relation between two channels Ch1 and Ch2 capturing a unilateral confidential and authentic communication between A and B, expressed by the fact unilateral_conf_auth(Ch1,Ch2,B)), is expanded to the facts:

```
  confidential_to (Ch1,B)
  weakly_authentic(Ch1)
  weakly_confidential (Ch2)
  authentic_on        (Ch2,B)
  link            (Ch1,Ch2)
```

4.7.7 Fact introduction

When translating a fact introduction statement as defined next, a check is done whether the fact symbol is contained in the set PolicyPredicates; cf. § 3.5 and § 4.4. In this case, with the exception of iknows, an error message is produced by the translator because implicit facts are not allowed here.

```
IntroduceFact(stmt, rest, sl, return_sl) {

  stmt = funcapp

  LHS = state fact for this entity, with step label sl
  RHS = state fact for this entity, with step label succ(sl).
        funcapp

  add
      LHS "=>" RHS
  to "section rules" in the translation

  ParseCode(rest, succ(sl), return_sl)
}
```

An element addition statement of the form add(Set,X) is handled as an introduction of contains(Set,X).

4.7.8 Fact retraction

When translating a fact retraction statement as defined next, a check is done whether the fact symbol is contained in the set PolicyPredicates; cf. § 3.5 and § 4.4. In this case, an error message is produced by the translator because implicit facts are not allowed here.

```
RetractFact(stmt, rest, sl, return_sl) {

  stmt = "retract" funcapp

  LHS = state fact for this entity, with step label sl.
        funcapp
  RHS = state fact for this entity, with step label succ(sl)

  add
      LHS "=>" RHS
  to "section rules" in the translation

  ParseCode(rest, succ(sl), return_sl)
}
```

If the fact to be retracted is not present, execution is blocked (as long as the fact is not introduced).

An element removal statement of the form remove(Set,X) is handled as a retraction contains(Set,X).

4.7.9 Branch

```
Branch(stmt, rest, sl, return_sl) {

  stmt = "if" Guard "then" LeftStmt ("else" RightStmt)?

  p_1..p_n = positiveGuards(Guard)
  n_1..n_m = negativeGuards(Guard)

  apply adaptGuard to p_1..p_n and n_1,..,n_m

  % positive branches, i.e. Guard satisfied
  for (i from 1 to n) {
    LHS = state fact for this entity, with step label sl.
          state facts for the owners of variables appearing in
```

```
            p_i
    RHS = state fact for this entity, with step label
            branch(sl, 0).
          state facts for the owners of variables appearing in
            p_i
    add
        LHS.p_i "=>" RHS.renewPositiveFactsIn(p_i)
    to "section rules" in the translation
  }

  % negative branches, i.e. Guard not satisfied
  for (i from 1 to m) {
    LHS = state fact for this entity, with step label sl.
          state facts for the owners of variables appearing in
            n_i
    RHS = state fact for this entity, with step label
            branch(sl, 1).
          state facts for the owners of variables appearing in
            n_i
    add
        LHS.n_i "=>" RHS.renewPositiveFactsIn(n_i)
    to "section rules" in the translation
  }

  ParseCode(LeftStmt, branch(sl, 0), succ(sl))
  ParseCode(RightStmt, branch(sl, 1), succ(sl))
  ParseCode(rest, succ(sl), return_sl)
}
```

Conceptually, an `if` statement corresponds to two rewrite rules branching the execution of the model in one direction if the `Guard` is satisfied, or another if it is not. Then, for each of these branches, the corresponding statements are parsed, and the control in both cases goes back to the original thread of execution.

This holds, though, only for (possibly negated) literals, while in the general case a Boolean expression can be used. Since ASLan rewrite rules admit only conjunctions of conditions, we use an auxiliary function `positiveGuards` for computing the DNF (disjunctive normal form) and returning the list of clauses in it. Analogously, `negativeGuards` computes the DNF of the negated guard, and returns its clauses. For computing the DNF, we expand implications of the form `P => Q` as `!P | Q`.

For each of these clauses a *branching* rule will be created and added to the translation, leading to the apposite branch. Furthermore, evaluation of the guards should not remove facts from the state, so we assume a further function `renewPositiveFactsIn` that renews the positive explicit facts in the clause. Implicit facts are not reintroduced on the right-hand side of the corresponding rewrite rule because implicit facts (stored in the set PolicyPredicates, see § 4.4) must not appear on the RHS of rewrite rules.

Example 6. Let p, q and q' be predicates defined in BidM on agents, and the following branch occurs at step sl:

```
if (p(Actor)) then
  q(Actor);
else
  q'(Actor);
```

We will then create one rule leading to the first branch if p(Actor) is satisfied

```
state_BidM(Actor,IID,sl,BP,M).p(Actor)
  =>
state_BidM(Actor,IID,branch(sl, 0),BP,M).p(Actor)
```

and another leading to the second branch if instead p(Actor) is not satisfied

```
state_BidM(Actor,IID,sl,BP,M).not(p(Actor))
  =>
state_BidM(Actor,IID,branch(sl, 1),BP,M)
```

Parsing the two branches separately will give rise to the following rules

```
state_BidM(Actor,IID,branch(sl, 0),BP,M)
  =>
state_BidM(Actor,IID,succ(branch(sl, 0)),BP,M).q(Actor)
state_BidM(Actor,IID,succ(branch(sl, 0)),BP,M)
  =>
state_BidM(Actor,IID,succ(sl),BP,M)

state_BidM(Actor,IID,branch(sl, 1),BP,M)
  =>
state_BidM(Actor,IID,succ(branch(sl, 1)),BP,M).q'(Actor)
state_BidM(Actor,IID,succ(branch(sl, 1)),BP,M)
  =>
state_BidM(Actor,IID,succ(sl),BP,M)
```

□

Example 7. A *guard* can be a general Boolean expression, which will always be reduced to a disjunctive normal form like

$$c_1 \vee \ldots \vee c_n$$

where the clauses $c_1 \ldots c_n$ are conjunctions of (possibly negated) predicates.

For each of these clauses, we need a rewrite rule checking it on its LHS, and leading to a common RHS (except for the positive facts of the clause that are renewed).

Consider the following branch, where p, r and s are constant facts here.

```
if ((p | not(r)) & not(s)) then
  q(Actor);
else
  q'(Actor);
```

Here the DNF obtained is

$$(p \wedge not(s)) \vee (not(r) \wedge not(s))$$

Therefore the obtained rules will be

```
state_BidM(Actor,IID,sl,BP,M).p.not(s)
  =>
state_BidM(Actor,IID,branch(sl, 0),BP,M).p

state_BidM(Actor,IID,sl,BP,M).not(r).not(s)
  =>
state_BidM(Actor,IID,branch(sl, 0),BP,M)

state_BidM(Actor,IID,branch(sl, 0),BP,M)
  =>
state_BidM(Actor,IID,succ(branch(sl, 0)),BP,M).q(Actor)

state_BidM(Actor,IID,succ(branch(sl, 0)),BP,M)
  =>
state_BidM(Actor,IID,succ(sl),BP,M)
```

as to the case where the guard holds. For the opposite case, i.e. guard evaluation fails, we will proceed similarly but considering the negated guard, whose DNF is

$$(not(p) \wedge r) \vee s$$

generating the following rules

```
state_BidM(Actor,IID,sl,BP,M).not(p).r
  =>
state_BidM(Actor,IID,branch(sl, 1),BP,M).r

state_BidM(Actor,IID,sl,BP,M).s
  =>
state_BidM(Actor,IID,branch(sl, 1),BP,M).s

state_BidM(Actor,IID,branch(sl, 1),BP,M)
  =>
state_BidM(Actor,IID,succ(branch(sl, 1)),BP,M).q'(Actor)
state_BidM(Actor,IID,succ(branch(sl, 1)),BP,M)
  =>
state_BidM(Actor,IID,succ(sl),BP,M)
```

□

Note, finally, that, before any actual translation of the statement is put in place, the function adaptGuard, which we will present in detail in § 4.8, is applied to each clause of the guard's DNF to convert all logical connectives into ASLan equivalent predicates. This approach will be applied to all statements using guards.

4.7.10 Loop

```
Loop(stmt, rest, sl, return_sl) {

  stmt = "while" Guard Body

  p_1..p_n = positiveGuards(Guard)
  n_1..p_m = negativeGuards(Guard)

  apply adaptGuard to p_1..p_n and n_1,..,n_m

  % positive branches, i.e. Guard satisfied
  for (i from 1 to n) {
    LHS = state fact for this entity, with step label sl.
          state facts for the owners of variables appearing in p_i
    RHS = state fact for this entity, with step label branch(sl, 0).
          state facts for the owners of variables appearing in p_i
    add
        LHS.p_i "=>" RHS.renewPositiveFactsIn(p_i)
    to "section rules" in the translation
  }

  % negative branches, i.e. Guard not satisfied
  for (i from 1 to m) {
    LHS = state fact for this entity, with step label sl.
          state facts for the owners of variables appearing in n_i
    RHS = state fact for this entity, with step label succ(sl).
          state facts for the owners of variables appearing in n_i
    add
        LHS.n_i "=>" RHS.renewPositiveFactsIn(n_i)
    to "section rules" in the translation
  }
  ParseCode(Body, branch(sl, 0), sl)
  ParseCode(rest, succ(sl), return_sl)
}
```

A `while` construct translates in a very similar way to the `if` construct: two sets of rules are created, one set of which leading to the body of the loop (if the guard is satisfied) and the other leading outside the loop (if the guard is not satisfied). Then the body is parsed, and given as return step label the step label for evaluation of the guard, so that it will be reevaluated. Ultimately, the remaining statements are parsed.

Example 8. Consider the following loop

```
while (p) {
  q;
}
```

We will create one rule leading inside the loop if `p` is satisfied

```
state_BidM(Actor,IID,sl,BP,M).p
  =>
state_BidM(Actor,IID,branch(sl, 0),BP,M).p
```

and another leading outside of the loop if instead `p` is not satisfied

```
state_BidM(Actor,IID,sl,BP,M).not(p)
  =>
state_BidM(Actor,IID,succ(sl),BP,M)
```

Parsing the inner statements of the loop, we will obtain the following rules

```
state_BidM(Actor,IID,branch(sl, 0),BP,M)
  =>
state_BidM(Actor,IID,succ(branch(sl, 0)),BP,M).q
state_BidM(Actor,IID,succ(branch(sl, 0)),BP,M)
  =>
state_BidM(Actor,IID,sl,BP,M)
```

□

4.7.11 Select

```
Select(stmt, rest, sl, return_sl) {

  stmt = "select" "{"
           "on" Guard_1 ":" Stmt_1
           ...
           "on" Guard_g ":" Stmt_g
```

```
         "}"

  for (i from 1 to g) {
    p_1..p_n = positiveGuards(Guard_i)

    apply adaptGuard to p_1..p_n

    % positive branches only, i.e. Guard satisfied
    for (j from 1 to n) {
      LHS = state fact for this entity, with step label sl.
            state facts for the owners of variables appearing in p_j
      RHS = state fact for this entity, with step label branch(sl, i).
            state facts for the owners of variables appearing in p_j
      add
          LHS.p_j "=>" RHS.renewPositiveFactsIn(p_j)
      to "section rules" in the translation
    }

    ParseCode(LeftStmt, branch(sl, i), succ(sl))
  }

  ParseCode(rest, succ(sl), return_sl)
}
```

The `select` construct allows one to nondeterministically pick one code block provided its guard is satisfied. If no guard is satisfied, it blocks until one is.

The`on(...)` guards in a `select` statement, which typically contain receive conditions, are translated with the `adaptGuard` function as described in § 4.8.

This translates in a way similar to an `if` construct, differing in the following two aspects:

- the number of branches varies according to the number of different code blocks contained, and
- there is no branching for failed evaluation of guards, since the construct blocks until one is satisfied.

Example 9. The methods

```
select {
  on(p_1) : q_1;
  ...
  on(p_n) : q_n;
}
```

will be translated to the following rules

```
state_BidM(Actor,IID,sl,BP,M).p_1
  =>
state_BidM(Actor,IID,branch(sl, 1),BP,M).p_1
state_BidM(Actor,IID,branch(sl, 1),BP,M)
  =>
state_BidM(Actor,IID,succ(branch(sl, 1)),BP,M).q_1
state_BidM(Actor,IID,succ(branch(sl, 1)),BP,M)
  =>
state_BidM(Actor,IID,succ(sl),BP,M)
...
state_BidM(Actor,IID,sl,BP,M).p_n
  =>
state_BidM(Actor,IID,branch(sl, n),BP,M).p_n
state_BidM(Actor,IID,branch(sl, n),BP,M)
  =>
state_BidM(Actor,IID,succ(branch(sl, n)),BP,M).q_n
state_BidM(Actor,IID,succ(branch(sl, n)),BP,M)
  =>
state_BidM(Actor,IID,succ(sl),BP,M)
```

□

4.7.12 Assert

```
Assert(stmt, rest, sl, return_sl) {

  stmt = "assert" assertion-name ":" "exists" V_1,..,V_n "." Guard

  W_1,..,W_m = intersection of variables of Guard (but excluding
               V_1,..,V_n) and the variables in the scope of the entity

  check = fresh predicate, the name of which includes
          assertion-name, on m variables of types agreeing
          with W_1,..,W_m, plus the entity instance ID (IID)
          and the next step label (succ(sl)).

  LHS = state fact for this entity, with step label sl.
        state facts for the owners of W_1,..,W_m
  RHS = state fact for this entity, with step label succ(sl).
        state facts for the owners of W_1,..,W_m.
        check(W_1,..,W_m,IID,succ(sl))

  add
      LHS "=>" RHS
  to "section rules" in the translation

  LHS = state fact for this entity, with step label succ(sl).
        check(W_1,..,W_m,IID,succ(sl))
  RHS = state fact for this entity, with step label succ(succ(sl)).

  add
      LHS "=>" RHS
  to "section rules" in the translation

  add
      [] (check(W_1,..,W_m,IID,SL) => exists V_1,..,V_n. Guard)
  to the "section goals" in the translation.

  ParseCode(rest, succ(succ(sl)), return_sl)
}
```

An `assert` statement is translated into two rules, one adding a fresh predicate `check`, whose purpose is just to tag the state so to make references possible in the ASLan `goals` section, and the other removing the predicate. In fact, the assertion itself (i.e. the `Guard` whose satisfaction is required) is checked via an extra goal, checking that when this predicate appears, the `Guard` holds. The `exists V_1,..,V_n. Guard` part in the above definition is further translated as given in § 4.12.

4.8 Translation of Guards

In some of the previous sub-procedures, we made use of a function `adaptGuard`. This is a minor and straightforward procedure that only replaces all applications of logical connectives with ASLan predicates, as specified in Table 11. Note that the `|` (disjunction) and `=>` (implication) operators are not included since there is no corresponding predicate in ASLan, and coherently the function will be only applied to separate clauses of disjunctive normal forms. The function `adaptGuard` recursively follows the term structure of the guard and does the substitutions defined in Table 11. Note that the translation of sub-terms (i.e. 'atoms' from the logical point of view) is specified in § 4.9.

ASLan++ operator	ASLan predicate
`!G`	`not(G)`
`G_1 & G_2`	`G_1.G_2`
`T_1 = T_2`	`equal(T_1,T_2)`
`T_1 != T_2`	`not(equal(T_1,T_2))`

Table 11: Substitutions done by the `adaptGuard` function

4.9 Translation of Terms

In § 4.7, we presented a procedure for translating ASLan++ statements into ASLan transitions, but we did not apply any translation of terms. There are two reasons behind this:

- Split the translation into a *higher* level (of statements) and a *lower* one (of terms contained in them), allowing for cleaner exposition and ease of understanding.
- In most cases, ASLan++ terms can be easily replaced by equivalent terms in ASLan by means of semantically equivalent predicates. However, there are a few exceptions (e.g. set literals and assignment of variables by pattern matching) that can not be handled by simply replacing the term with another, as they affect the entire resulting rule for the statement that contains the term.

The approach followed is therefore to defer translation of terms, carrying over the original ones during translations of statements (and guards included in them), and then apply a term translation procedure to the generated rule, which we describe in the following.

Consider a rule R of the form

```
LHS =[exists E]=> RHS
```

and let us apply the procedure to R. Suppose that R is now in an intermediate state where both ASLan and ASLan++ predicates (namely, *transmission events*) may appear, of which only the latter need be translated but all need be processed recursively for contained terms. For each predicate in R, we recursively traverse each (sub-)term `T` according to its syntactic form, obtaining the following cases:

- `T = c`, i.e. `T` is a constant `c`. In this case we leave `c` as is.
- `T = V`, i.e. `T` is a variable `V`. The case is similar to the above one for constants, except that the variable name must be bound to its value. Therefore we add the state fact of the owner of `V` to both `LHS` and `RHS` of R.
- `T = [Actor]`, i.e. `T` is the default pseudonym, is translated to `defaultPseudonym(Actor,IID)` (where `IID` is the instance ID of the current entity).
- `T = [T_1]_[T_2]`, i.e. `T` represents an explicitly given pseudonym `T_2` for the agent represented by term `T_1`. In this case, the result is simply the recursive translation of `T_2` — that is, `T_1` is ignored.
- `T = T_1.T_2..T_n`, i.e. `T` is the concatenation of terms `T_1,T_2,.., T_n`. In this case, we apply the binary predicate `pair` to `T_1` and the recursive application of `pair` to `T_2..T_n`, and finally apply the terms translation procedure recursively on each of `T_1,..,T_n`.
- `T = (T_1,T_2,..,T_n)`, i.e. `T` is a tuple containing the terms `T_1,T_2, ..,T_n`. This case is translated exactly as the one for concatenation, also as right-associative pairing.

- `T = { T_1,T_2,..,T_n }`, i.e. `T` is a set literal of type 'τ `set`', containing the elements `T_1,T_2,..,T_n` all of type τ. We create a fresh function symbol `sf` of type τ `set`, parameterized by the instance ID of the entity which owns the set literal. Then we substitute `T` by `sf(IID)`, and add to the `RHS` of R the facts `contains(sf(IID),T_1)`,..., `contains(sf(IID),T_n)`.
- `T = ?`, i.e. `T` pattern-matches anything, which is allowed only in guards, i.e. on the `LHS` of R. In this case, we just create a fresh variable name `V'` and replace `T` by `V'`.
- `T = ?V`, i.e. `T` represents an assignment of `V` by pattern-matching, which is allowed only in guards, i.e. on the `LHS` of R. We add the state fact of the owner of `V` to both the `LHS` and `RHS` of R where we replace the variable name `V` in the left state fact by a fresh variable name `V'`. For example, `if p(?V) then q(V)` is translated to

  `state_X(...,V',...). p(V) => state_X(...,V,...).p(V).q(V).`
- `T = f(T_1,..,T_n)`, i.e. `T` is the application of a function `f` to the terms `T_1,..,T_n`. In the general case, `f` will be left as is, and we need only apply the terms translation procedure recursively to `T_1,..,T_n`. As to the case where `f` is a `send` or a `receive`, its translation will change according to the channel model employed, discussed in depth in § 4.7.6.

4.10 Translation of the Body Section

Now that we have introduced the procedure `ParseCode`, parsing the `Body` section is straightforward. In particular, we just need invoke

```
ParseCode(Stmts, sl_0, null)
```

where `Stmts` is the body and `sl_0` the initial step label (while no `return` step label is provided being this the main block of instructions).

In the case of the root entity, it is also necessary to add its state fact with step label `sl_0` in `section inits` in the translation, in order for the execution to start. For all other entities, all rules that have on their left side the state fact with step label `sl_0` need to contain the extra precondition `not(dishonest(Actor))` to avoid executing superfluous transitions, which would be needlessly inefficient and could lead to spurious attacks.

Note that, as a final step of the translation, the step label terms, built up from the `sl_0`, `succ` and `branch` symbols, are replaced with numbers. § 4.13 describes how this is done.

4.11 Translation of the Constraints Section

The translation of constraints in the constraints section is immediate, consisting in minor syntactical adjustments.

4.12 Translation of the Goals Section

4.12.1 Invariants

In the case of invariants given as LTL goals, in a similar way as for guards, we need to translate the LTL operators to ASLan LTL syntax (given in § A.3), according to Table 12. After doing so, the terms contained in the goal formulae must be converted, by means of a slight variant of the procedure introduced in § 4.9 (which we omit).

Operator	ASLan++ connective	ASLan predicate
$\neg$	`!f`	`not(f)`
$=$	`f_1 = f_2`	`equal(f_1,f_2)`
$\neq$	`f_1 != f_2`	`not(equal(f_1,f_2))`
$\wedge$	`f_1 & f_2`	`and(f_1,f_2)`
$\vee$	`f_1 | f_2`	`or(f_1,f_2)`
$\Rightarrow$	`f_1 => f_2`	`implies(f_1,f_2)`
$\forall$	`forall V_1 .... V_n.f`	`forall(V_1,..forall(V_N,f)..)`
$\exists$	`exists V_1 .... V_n.f`	`exists(V_1,..exists(V_N,f)..)`
neXt	`X(f)`	`X(f)`
Yesterday	`Y(f)`	`Y(f)`
Finally	`<>(f)`	`F(f)`
Once	`<->(f)`	`O(f)`
Globally	`[](f)`	`G(f)`
Historically	`[-](f)`	`H(f)`
Until	`U(f_1,f_2)`	`U(f_1,f_2)`
Release	`R(f_1,f_2)`	`R(f_1,f_2)`
Since	`S(f_1,f_2)`	`S(f_1,f_2)`

Table 12: Translation of goals

4.12.2 Channel Goals

As we described above, we can use the bullet annotations to specify different kinds of transmission goals of a service. Intuitively, this means that the service should ensure the authentic, confidential, or in other ways protected transmission of the respective message. Note that all these goals are independent of the channel model, which only affects the assumptions on channels.These definitions are close to standard authentication and secrecy goals of security protocols, e.g. [9, 14, 17]. As we remarked, we focus here on these kinds of channels as goals (and to the kinds of channels as assumptions discussed above), but of course additional kinds might be possible by extending the syntax and semantics we give here.

Auxiliary events. In order to formulate the goals in a service-independent way, we use, in the translation, a set of *auxiliary events* (modeled as ASLan facts) of the service execution

Channel kind	Events used	Rule for deriving `protocol_id`
Authentic	`witness`, `request`	"`auth_`" + channel goal name
Fresh	`request`	"`fresh_`" + channel goal name
Confidential	`secret`	"`secr_`" + channel goal name

Table 13: Events used in the translation of channel goals and rules used for deriving the protocol ID from the name of channel goals

as an interface between the concrete service and the general goals. The use of such auxiliary events is common to several approaches (including, most notably, AVISPA's IF and Casper [15]). More specifically, we consider the following kinds of fact symbols:[40]

```
witness: agent * agent * protocol_id * message       -> fact
request: agent * agent * protocol_id * message * nat -> fact
secret : message       * protocol_id * agent set     -> fact
```

where the `protocol_id` type is used to hold an identifier for the particular goal.[41]

These events provide an interface over which we define service properties in LTL formulae. Each specification of a channel as a goal (as described in § 3.8) implies the creation of such events within the execution of a particular entity; in particular the translation to ASLan decorates appropriate transition rules with these facts as described here. Remember that channel goal labels can be used only in the context of a transmission statement.

The auxiliary events are used depending on the kind of the channel.

- If the channel is *authentic* (there is a bullet at the left of the arrow) then the `witness` and `request` events are used.

- If the channel is *authentic* and *fresh* (the arrow is double-headed) then an additional rule involving `request` is used.

- If the channel is *confidential* (there is a bullet at the right of the arrow) then the `secret` event is used.

The value for the `protocol_id` parameter is derived from the name of the channel goal, depending on the kind of the channel. Table 13 summarizes the events that are used for each kind of channel, together with the rule for deriving the value for `protocol_id`.

Declarations. For ease of description, consider the example similar to the one given in § 3.9.3, where in some entity E there is the following channel goal declaration.

```
secure_Alice_Payload_Bob:(_) tA *->* tB;
```

[40] Other fact symbols could be considered for the specification of other channel goals, such as the facts `whisper` and `hear` that are considered for a different notion of secrecy for the compositionality results given in [18].

[41] For the fact symbol `secret`, the parameter of type `protocol_id` is not actually needed.

According to the syntactic restrictions given in § 3.9.3, in each sub-entity in which the goal label re-appears, there will be exactly one parameter that corresponds to the sender term `tA` and one parameter that corresponds to the receiver term `tB`. In a sending sub-entity $E1$, the former parameter must be `Actor`, and let us assume that the latter parameter has the name `B`. In a receiving sub-entity $E2$, the former parameter must be `Actor`, and let us assume that the latter parameter has the name `A`.

Labels in send statements. Given a transmission statement where the `Actor` appears on the left of the arrow such that it is a send statement, located in a sub-entity $E1$, with the same channel goal label:

```
Actor -> B': ...secure_Alice_Payload_Bob:(Payload)...
```

the translator will first translate the send statement `Actor -> B': ...Payload...` where the goal label has been stripped off.

Then it generates suitable auxiliary events, for instance

```
witness(Actor,B,auth_secure_Alice_Payload_Bob,Payload)
```

and inserts them into the ASLan transition rule representing the send event. For undirected authentication (written e.g. `A *->? ?`), the second parameter of `witness` is replaced by `i` indicating that the receiver information is irrelevant.

Note that here `B`, the parameter of $E1$ corresponding to the term `tB`, usually still has the value of `tB`, but in fact `B` might have changed due to an assignment to `B` that may have occurred in the meantime. Moreover, typically the term `B'` that occurs in the send statement usually is identical to `B`. These two agreements are typically intended, but not enforced, which gives the modeler more freedom but also more responsibility.

When using a pseudonym `Phi`, the sender (or receiver) name will simply be replaced by that in the event:

```
witness(Phi,B,auth_secure_Alice_Payload_Bob,Payload)
```

The confidentiality part, if any, is basically translated as described in § 4.12.3, adding the fact

```
secret(Payload,secr_secure_Alice_Payload_Bob,
            secr_secure_Alice_Payload_Bob_set)
```

in the ASLan transition rule representing the send event within $E1$ and the necessary `contains` facts in the (ancestor) entity E.

Labels in receive statements. Given a transmission statement where the `Actor` appears on the right of the arrow such that it is a receive statement in a sub-entity $E2$ with the same channel goal label:

```
A' -> Actor: ...secure_Alice_Payload_Bob:(?Payload)...
```

the translator will first translate the receive statement `A' -> Actor: ...?Payload...` where the goal label has been stripped off.

Then it generates suitable auxiliary events, for instance

```
request(Actor,A,auth_secure_Alice_Payload_Bob,Payload,IID)
```

and inserts them into the ASLan transition rule representing the receive event. Note that, in analogy to the above, here `A` and `A'` usually have the value of `tA`, but this is not guaranteed.

When using a pseudonym `Phi`, the receiver (or sender) name will simply be replaced by that in the event:

```
request(Phi,A,auth_secure_Alice_Payload_Bob,Payload,IID)
```

If the channel goal includes confidentiality, the fact

```
contains(i, secr_secure_Alice_Payload_Bob_set(IID'))
```

is introduced in the translation of the receiving statement, where `IID'` is the instance ID of the (ancestor) entity E. In this way, the intruder is added to the set of agents allowed to know the secret, because the confidentiality channel goal is required to hold only until reception of the secret.

Authenticity and freshness. We can now define the semantics of authenticity and freshness as LTL formulae (to be used as LTL goals, or equivalently, used in negated form as attack states) over the given facts without referring to the particularities of the specified entities such as the message formats.

The built-in LTL semantics[42] of the authentication goal is

```
    forall A B M IID. [](request(B,A,P,M,IID) =>
      (<->(witness(A,B,P,M))
       | dishonest(A)))
```

where `P` is the protocol ID of the goal, as defined above. For undirected authentication (written e.g. `A *->? ?`), the second parameter of `witness` is replaced by `i` because the receiver information is irrelevant from the sender's point of view.

This is a standard authentication goal (non-injective agreement [14]), which, in the case of CCM and ICM, includes directedness.

For the channel compositionality results of [18], it is necessary strengthen this with the additional condition that, if the sender is dishonest, then the intruder must know this message. This involves negation of `iknows` facts, which so far is supported by SATMC only. Therefore we did not define it as part of the built-in authentication translation. If needed, it may be user-defined as

```
    forall A B P M IID. [](request(B,A,P,M,IID) =>
                           (  witness(A,B,P,M)
                            | (dishonest(A) => iknows(M))));
```

[42] In the translation, `<->(witness(A,B,P,M))` is simplified to `witness(A,B,P,M)`, which is equivalent because `witness` facts are stable, i.e. they are never retracted.

The LTL semantics[43] of the freshness goal, which may be given in addition to an authenticity goal, is

```
forall A B P M IID IID'. [] (request(B,A,P,M,IID) =>
  (!(<-> (request(B,A,P,M,IID') & !(IID=IID'))
   | dishonest(A))))
```

This says that the agent B executing an entity instance IID should not accept the same value M in a different session (identified by IID') from the same honest communication partner A at the same point of B's execution identified by A and P: that is, B has never previously accepted the same value M at that point.

4.12.3 Secrecy Goals

Each secrecy goal will generate at the ASLan level several transition rules.

Declarations. For ease of description, consider the following secrecy goal:

```
secrecyGoalName:(_) {tA, tB, tC};
```

declared in some entity E.

According to the syntactic restrictions given in § 3.9.4, in each sub-entity in which the goal label re-appears, there will be parameters that correspond to the terms tA, tB, and tC. Let us assume that these parameters have the names A, B, and C.

Labels in sub-entities. In any direct sub-entity E', each term that bears a matching goal label, for example,

```
...secrecyGoalName:(Token)...
```

is translated recursively, and then the following procedure is applied, using the *secret* predicate available in ASLan. It has the following signature:

```
 secret: message * protocol_id * agent set -> fact
```

The protocol ID is derived from the name of the secrecy goal, while the message value is taken from the translation of the term, which in our example is Token.

For rendering the set of agents, the procedure for translating set literals (described in § 4.9) is applied, as if the agents were specified using a set literal. A new function symbol is introduced for the set. The function symbol is parameterized by the instance ID of the parent entity E and has the type agent set. The name of the function symbol is derived from the name of the secrecy goal by appending the suffix "_set". The signature of the function symbol is therefore:

```
secrecyGoalName_set: nat -> set(agent)
```

[43] In the translation, <->(request(B,A,P,M,IID') & !(IID=IID')) is simplified to request(B,A,P,M,IID') & !(IID=IID'), which is equivalent because request facts are stable, i.e. they are never retracted.

The membership of each agent to the set is stated using the `contains` predicate. The following transition rule is generated within entity E':

```
child(IID', IID).
state_...(Actor, IID, ...)
=>
child(IID', IID).
contains(A, secrecyGoalName_set(IID')).
contains(B, secrecyGoalName_set(IID')).
contains(C, secrecyGoalName_set(IID')).
secret(Token, secrecyGoalName, secrecyGoalName_set(IID')).
state_...(Actor, IID, ...)
```

where `A`, `B`, and `C` are the entity parameters corresponding to the set of agents specified in the secrecy goal, `IID` is the instance ID of E' and, `IID'` is the instance ID of its parent entity E. Note that here `A` usually has the value of `tA`, and similarly for `B` and `C`, but they may have changed (in case the entity parameters have been assigned to in the meantime).

On the ASLan level the secrecy goal is stated as an LTL goal over the *secret* predicate:

```
    forall M,P,As. []
      ((secret(M, P, As) & iknows(M)) => contains(i, As))
```

Informally the meaning is that if a certain message `M` is a secret shared between a set `As` of agents, and the intruder knows `M`, then the intruder must be part of the set `As` of agents.

Note that this is possibly too strong if among the members of `As` there is a dishonest agent that is not the intruder itself. For such cases, spurious attacks can be avoided by the ASLan++ modeler stating just before any agent in `As` is declared dishonest:

```
select{ on(child(?IID',IID)): secrecyGoalName_set(IID')->add(i); }
```

where the `IID'` variable must be declared of type `nat`.

4.13 Assignment of numbers to step label terms

This phase of the translation generates and uses numbers for the step labels in the state facts of entities. This is done because even for an relatively small number of statements in an entity the successive application of `succ` and `branch` creates very complex terms, which would only place an unnecessary burden on the verifiers.

In order to simplify the generated ASLan output, in this last phase of the translation, after all transition rules are generated, we assign numbers to the generated step label terms, so that each step label term receives a unique number. This is done by a procedure that starts at the initial step label `sl_0`, which is replaced by `1`, and proceeds recursively on the `succ` and `branch` symbols (the procedure is omitted). Then in all generated transitions the step label terms are replaced by their assigned numbers.

4.14 Step Compression

4.14.1 Motivation and example

The translation we define in § 4.7 produces for each entity transition rules expressing progress at statement-level granularity. For instance, the three lines

```
receive(?, ?A);
N := fresh();
send(A, N);
```

would get translated into three individual transitions:

```
state_responder(Actor, IID, 1, dummy_agent, dummy_message).iknows(A)
  =>
state_responder(Actor, IID, 2, A, dummy_message)

state_responder(Actor, IID, 2, A, dummy_message)
  =[exists N]=>
state_responder(Actor, IID, 3, A, N)

state_responder(Actor, IID, 3, A, N)
  =>
state_responder(Actor, IID, 4, A, N).iknows(N)
```

After each such progress step, other processes or the intruder may make progress. This gives rise to the following issues:

- Race conditions may appear in case of shared knowledge (e.g. all instances of a service running on the same server, accessing its database). If we do not want to explicitly specify mutual exclusion algorithms within the model to prevent race conditions, we thus need to be able to specify that certain sequences of transitions are *atomic*. This is of course an abstraction step that the specifier may choose to use, at the risk of losing certain behaviors of the model.

- On the practical side, even for relatively small processes, this fine-grained model produces a large set of interleavings even for a few processes. Specifications can thus easily get infeasible for the validation tools. Note that in many cases these interleavings are irrelevant because they concern internal computations of a service and the relative ordering with other internal computations is not really relevant. On the other hand, it is almost impossible to determine which interleavings could give rise to new attacks on the tool side.

For this reason we introduce here the concept of *step compression*, allowing modelers to consider different (coarser) granularities for their models. The idea is that some actions (instructions in the ASLan++ code) can be considered internal and not relevant in the interleaving of the entities, hence they can be safely lumped together. The effect of lumping is similar to *atomic* blocks, i.e. nothing else can happen before the atomic section is finished.

4.14.2 Step granularity and breakpoints

The step granularity of a model is controlled by the set of *breakpoint* actions. By default, the only breakpoint is the action `receive`. The modeler may optionally specify for each entity additional breakpoints with the declaration `breakpoints {bp_1,...,bp_n}`. This set of breakpoints is inherited to sub-entities unless they declare their own set of breakpoints.

If needed, a modeler can specify further individual breakpoints in the model by placing a breaking instruction (e.g., `receive(?,?)`) at the desired points of the code. This will artificially force the translation to stop compressing and start a new transition.

The meaning is that all transitions within the entity are compressed up to, but not including, any breakpoint action. This corresponds to declaring that everything from a breakpoint action up to the next one is considered as an internal computation. This in particular allows for compressions up to an event in a loop without unrolling this loop for the specification of the compression. We believe that this is a useful compromise between the goal of a declarative, intuitive, easy-to-use specification language of services and the needs of the validation tools that have to work on these specifications.

4.14.3 Translation of compressed steps

The compression of several rules would in many cases considerably complicate the exposition of this translation process and is in some cases impossible (e.g. when an entire loop is compressed). Therefore, we describe the translation by using the fine-grained transitions but with annotations specifying what should be compressed as follows. These annotations are the special `state!` facts of ASLan (see § A.3.5): like standard `state` facts, they represent the local state of honest agents where the exclamation mark expresses an intermediate state of a compressed transition. Recall that the semantics is to consider *macro-transitions* that compress all the intermediate *micro-transitions* where honest agents are in a local state denoted by a `state!` fact.

In the translation from ASLan++ to ASLan, the compression can now be straightforwardly integrated into the normal translation process. Consider a transition rule that is produced by the translation and that has the following form:

```
state_X(MSGs,IID).facts | conditions
=>
state_X(MSGs',IID).facts'
```

where `IID` is the identifier of the current entity instance. To add the compression, the translator would replace the left-hand side or right-hand side `state` fact, or both, with a `state!` fact, depending on the location of this transition with respect to the compression:

- If the transition enters a compressed section, then only the right-hand side is changed:

  ```
  state_X(MSGs,IID).facts | conditions
  =>
  state!_X(MSGs',IID).facts'
  ```

- If the transition is within the compressed section, i.e. neither entering nor exiting it, both sides are changed:

  ```
  state!_X(MSGs,IID).facts | conditions
  =>
  state!_X(MSGs',IID).facts'
  ```

- If the transition is exiting the compression section, only the left-hand side is changed:

  ```
  state!_X(MSGs,IID).facts | conditions
  =>
  state_X(MSGs',IID).facts'
  ```

Compressing the three step example above gives thus the following rules:

```
state_responder(Actor, IID, 1, dummy_agent, dummy_message).iknows(A)
  =>
state!_responder(Actor, IID, 2, A, dummy_message)

state!_responder(Actor, IID, 2, A, dummy_message)
  =[exists N]=>
state!_responder(Actor, IID, 3, A, N)

state!_responder(Actor, IID, 3, A, N)
  =>
state_responder(Actor, IID, 4, A, N).iknows(N)
```

According to the macro step semantics introduced in § A.3.5, a macro transition would not stop at the intermediate states that contain a `state!` fact, but continues, with any applicable rule, until reaching a state without a `state!` fact (and cannot apply any rule that does not change the `state!` fact, which guarantees that progress is made only in one process instance).

Note that in the above simple sequence of atomic actions, as well as in many similar cases, there is an equivalent *compressed rule*:

```
state_responder(Actor, IID, 1, dummy_agent, dummy_message).iknows(A)
  =[exists N]=>
state_responder(Actor, IID, 4, A, N).iknows(N)
```

Whenever feasible, the translator actually produces such compressed rules, since it makes verification easier for the back-end tools. As mentioned above, this is sometimes a complex procedure, e.g. if the sequence of actions contains branches, and there are cases when this is not possible, e.g. the compression of an entire loop. We have chosen not to complicate our semantics exposition with a complete description of the compression procedure but to regard it as an optimization step (that must be semantics-preserving, as in the above example) of the translation from ASLan++ to ASLan. See also § 4.15.2.

4.15 Optimizations

The translation procedure presented so far produces a set of transition rules that quickly increases in size, even for small ASLan++ specifications. In order to ease the work of the verifier tools, in the following we introduce two optimization techniques for the translation. The optimizations decrease the number of generated transition rules.

4.15.1 Elimination of empty transitions and redundant guards

Intuitive explanation. The first level of optimization aims at eliminating empty transitions and redundant guards. An empty transition does not change the state, except for the step label. A redundant guard is a guard which is either always true or always false.

Removing empty transitions and redundant guards alters the semantics of the ASLan++ model with respect to the *neXt* and *Yesterday* LTL operators. As an example, lets consider the following ASLan++ code:

```
body {
  M := fresh();
  while (true) {
    Actor -> B : M;
  }
}
```

Without optimizations, the generated set of transition rules is:

```
state_Alice(Actor, IID, 1, B, M)
=[exists M_1]=>
state_Alice(Actor, IID, 2, B, M_1)

state_Alice(Actor, IID, 2, B, M).true
=>
state_Alice(Actor, IID, 3, B, M).true

state_Alice(Actor, IID, 3, B, M)
=>
state_Alice(Actor, IID, 4, B, M).iknows(M)

state_Alice(Actor, IID, 4, B, M)
=>
state_Alice(Actor, IID, 2, B, M)

state_Alice(Actor, IID, 2, B, M).not(true)
=>
state_Alice(Actor, IID, 5, B, M)
```

After eliminating empty transitions and redundant guards, the set of transition rules becomes:

```
state_Alice(Actor, IID, 1, B, M)
=[exists M_1]=>
state_Alice(Actor, IID, 2, B, M_1)

state_Alice(Actor, IID, 2, B, M)
=>
state_Alice(Actor, IID, 2, B, M).iknows(M)
```

Empty transitions. Most transition rules generated by the translation procedure contain in their LHS and RHS *state* facts that refer to the same entity. This means that the transition expresses some progress in the state of an entity. We can write such transitions as

$$PF.NF.state_S\ C =\![V]\!\Rightarrow R.state_{S'}$$

where PF and NF are sets of facts and negated facts respectively, $state_S$ represents a state of an entity instance, C is a conjunction of possibly negated atomic conditions, R is the set of facts introduced by the transition the entity is taking, $state_{S'}$ is the next state of the entity, while V is a finite set of existential variables. See § A.3.2 for ASLan transition rules.

A transition rule is considered empty if $PF = \emptyset$, $NF = \emptyset$, $C = \emptyset$, $V = \emptyset$ and $R = \emptyset$. Informally this means that the transition does not change the state, except for moving one entity from a step label to another. Empty transitions can be written as

$$state_S \Rightarrow state_{S'}$$

Empty transitions are introduced by the translation procedure whenever a branch is exited.

An empty transition $state_{S'} \Rightarrow state_{S''}$ can be eliminated when it is the only transition that takes the entity from $state_{S'}$ to $state_{S''}$ and one of the following holds:

- there exists one or more other transitions that take the entity into state $state_{S'}$, so that we can write

$$\begin{array}{rcl} PF.NF.state_S\ C & =\![V]\!\Rightarrow & R.state_{S'} \\ state_{S'} & \Rightarrow & state_{S''} \end{array}$$

- there exists one or more other transitions that take the entity out of state $state_{S''}$, so that we can write

$$\begin{array}{rcl} state_{S'} & \Rightarrow & state_{S''} \\ PF.NF.state_{S''}\ C & =\![V]\!\Rightarrow & R.state_{S'''} \end{array}$$

In this case the existing chains of two transitions of the above form can be replaced by single transitions of the form

$$PF.NF.state_S\ C =\![V]\!\Rightarrow R.state_{S''}$$

or

$$PF.NF.state_{S'}\ C =\![V]\!\Rightarrow R.state_{S'''}$$

Redundant guards. A transition rule is considered to be a redundant guard if $PF = \emptyset$, $NF = \emptyset$, $C \subset \{\mathit{true}, \mathit{false}\}$, $V = \emptyset$ and $R = \emptyset$. Intuitively this means that the transition does not change the state, except for moving an entity from one step label to another, and additionally the transition is guarded by a guard which is either always *true*, either always *false*. Such transitions can be written as

$$\mathit{state}_S.\texttt{true} \Rightarrow \mathit{state}_{S'}.\texttt{true}$$

or

$$\mathit{state}_S.\texttt{not}(\texttt{true}) \Rightarrow \mathit{state}_{S'}$$

Such redundant guards can be introduced by the translation procedure when entering branches guarded by expressions which can be reduced to the logical constants *true* or *false*.

Transitions of the form $\mathit{state}_S.\texttt{not}(\texttt{true}) \Rightarrow \mathit{state}_{S'}$ can be eliminated if there exists no other transition taking the entity out of state_S. It can happen that, after dropping such a transition, other transition rules will become unreachable, in which case they can be also eliminated.

Transitions of the form $\mathit{state}_S.\texttt{true} \Rightarrow \mathit{state}_{S'}.\texttt{true}$ are equivalent to empty transitions and can be eliminated under the same conditions as empty transitions.

4.15.2 Merging of transitions

Following the notion of step compression explained in § 4.14, a second level of optimization is introduced. Instead of generating the transition rules with the `state!` facts and leaving the handling of atomicity to the verifier tools[44], these transition rules are actually lumped into one rule wherever possible.

This process of lumping transitions is guided by the breakpoints defined in the ASLan++ model. Intuitively, each macro-step should result in a single transition rule. However this is not always possible, for example if there are loops in the ASLan++ model. Thus in certain situations the lumping of transitions cannot replace using the `state!` fact.

As an example consider the following ASLan++ code:

```
body {
  B -> Actor : ?Req;
  Actor -> B : Resp;
  some_fact(Req, Resp);
}
```

Without any optimizations, the generated transition rules are:

```
state_Alice(E_A_Actor, E_A_IID, 1, B, Req_1, Resp).iknows(Req)
=>
state_Alice(E_A_Actor, E_A_IID, 2, B, Req, Resp)

state_Alice(E_A_Actor, E_A_IID, 2, B, Req, Resp)
```

[44] The handling of the `state!` facts has not yet been implemented for any of the back-ends.

```
=>
state_Alice(E_A_Actor, E_A_IID, 3, B, Req, Resp).iknows(Resp)

state_Alice(E_A_Actor, E_A_IID, 3, B, Req, Resp)
=>
state_Alice(E_A_Actor, E_A_IID, 4, B, Req, Resp).some_fact(Req, Resp)
```

Since no breakpoints are explicitly defined, the default breakpoint is the *receive* predicate. After merging transition rules where possible, only one transition rule results:

```
state_Alice(Actor,IID,1,B,Req_1,Resp).iknows(Req)
=>
state_Alice(Actor,IID,4,B,Req  ,Resp).iknows(Resp).some_fact(Req,Resp)
```

5 Conclusion

We have presented ASLan++, the AVANTSSAR specification language. The low-level language ASLan given in Appendix A is the input to the back-ends of the AVANTSSAR Platform. ASLan++ has the look and feel of procedural and object-oriented programming languages, and thus can be employed by users who are not experts in formal specification languages. The semantics of ASLan++ is defined by translation to ASLan. Both ASLan and ASLan++ are supported by the AVANTSSAR Platform.

ASLan++ allows us to formally specify services and their policies in a way that is close to what can be achieved with specification languages for security protocols and web services. The flexibility and expressiveness of ASLan++ and the underlying low-level language ASLan is demonstrated via the formalized and validated problem cases, reported in Deliverable D5.3 [7].

As discussed in the AVANTSSAR description of work (Annex I), the refined versions of ASLan++ and ASLan, as presented in this document, cover static and dynamic service and policy composition aspects. The specification languages and the entire AVANTSSAR Platform [6] support full-fledged specification and analysis of the case studies and the industrial-strength applications that we have considered.

A ASLan

This appendix describes the syntax and semantics of ASLan, the low-level input language for the back-ends of the AVANTSSAR Platform. We proceed as follows. In § A.1, we motivate why, and describe how, ASLan extends and refines the specification language Intermediate Format (IF) of the AVISPA Tool [9]. In § A.2, we give the syntax of ASLan, and we then describe the semantics of the language in § A.3.

Background on ASLan. The first version of ASLan (called ASLan v.1) was described in deliverable D2.1 [1] and then it was extended (and called ASLan v.1.1) in deliverable D2.2 [4]. In this deliverable, we refine ASLan in particular with respect to the notion of dynamic policies. We introduce implicit closure of states under policy rules, which allows us to specify the revocation of policy facts in ASLan in a natural way.

ASLan is defined by extending and refining the Intermediate Format IF [9], a specification language that several of the partners of the AVANTSSAR consortium developed in the context of the FP5 project AVISPA. IF is an expressive language for specifying security protocols and their properties, based on set rewriting. Moreover, IF comes with mature tool support, namely the AVISPA Tool and all of its back-ends, which provide the basis for the back-ends of the AVANTSSAR Platform that we have been developing. As described in detail in [1], ASLan extends IF with a number of important features so as to express diverse security policies, security goals, communication and intruder models at a suitable abstraction level, and thereby allow for the formal specification and analysis of complex services and service-oriented architectures. Most notably, ASLan extends IF with:

Horn Clauses: In ASLan, invariants of the system can be defined by a set of (definite) Horn clauses. Horn clauses allow us not only to capture the deduction abilities of the attacker in a natural way, but also, and most importantly, they allow for the incorporation of authorization logics in specifications of services.

LTL: In ASLan, complex security properties can be specified in Linear Temporal Logic. As shown, for instance, in [3], this allows us to express complex security goals that services are expected to meet as well as assumptions on the security offered by the communication channels.

ASLan is the actual input language to the validation tools that comprise the AVANTSSAR Platform.

A.1 Motivation

We define ASLan by extending the Intermediate Format IF [9], a specification language that several of the partners of the AVANTSSAR consortium developed in the context of the AVISPA project: it provides the user with an expressive language for specifying security protocols and their properties, based on set rewriting. Moreover, IF comes with mature tool support, namely the AVISPA Tool and all of its back-ends. However, the IF language has the following major shortcomings that make it unsuited for the analysis of complex services:

- In IF, the behavior of the system can only be described by means of transitions and this makes IF inadequate to express security policies, which are usually best described as invariants.
- In IF, security goals can only be expressed as reachability properties and this makes IF inadequate to express complex security goals (e.g. fair exchange) that occur in complex services.

To overcome these shortcomings, we have defined ASLan by extending IF with the following two important features:

Horn Clauses: In ASLan, invariants of the system can be defined by a set of Horn Clauses. Horn clauses allow us not only to capture the deduction abilities of the attacker in a natural way, but also, and most importantly, they allow for the incorporation of authorization logics in specifications of services.

LTL: In ASLan, complex security properties can be specified in LTL. As shown, for instance, in [3], this allows us to express complex security goals that services are expected to meet as well as assumptions on the security offered by the communication channels.

Moreover, in IF, the intruder can always overhear or intercept messages and fake new ones. This is not always possible in recent protocols and services, which often assume for their proper functioning that communication is carried out over secure (confidential and/or authentic) channels. This issue is discussed in detail in Deliverable D3.3 (Attacker models) [2]. ASLan++ natively supports various channels models, as described in § 3.8.

We have opted for ASLan to be an extension of IF with Horn clauses and LTL because in that way ASLan is expressive enough that many high-level languages can be translated to it. Moreover, defining ASLan as an extension of IF made it possible to extend the suite of tools provided in AVISPA, rather than building new tools from scratch, aiming thus at an already stable and effective tool support.

A.2 ASLan Syntax

A.2.1 Grammar in BNF

We first present the entire ASLan syntax in BNF with the usual conventions.

The grammar has two start symbols, `Prelude` and `ASLanFile`. Intuitively, the prelude file contains all declarations that are service-independent, while the ASLan file contains only service-specific declarations. An example of a prelude file in ASLan++ is given in § 3.13, while an example of a prelude file in ASLan is given in § A.2.6.

The symbol for comments is %, both for ASLan and in the BNF below.

```
Prelude ::=  TypeSymbolsSection
             SignatureSection
             TypesSection
             EquationsSection
             IntruderSection
             ConstraintsSection

ASLanFile  ::=  SignatureSection
                TypesSection
                InitsSection
                ClausesSection
                RulesSection
                ConstraintsSection
                GoalsSection

TypeSymbolsSection ::= "section typeSymbols:" Types
SignatureSection   ::= "section signature:"
                       ( SuperType | OpDecl )*
TypesSection       ::= "section types:" TypeDecl*
EquationsSection   ::= "section equations:" Equation*
InitsSection       ::= "section inits:"
                       ("initial_state" ConstId ":=" Facts)+
RulesSection       ::= "section rules:" Rule*
ConstraintsSection ::= "section constraints:" Constraint*
GoalsSection       ::= "section goals:" Goal*
IntruderSection    ::= "section intruder:"
                       (Clause | Rule)*
ClausesSection ::= ("section hornClauses:" Clause+)?

Clause ::=  "hc" ConstId ("(" Vars ")")?
                ":=" ("forall" Vars ".")? Fact
                     (":-" Fact Conditions ("," Fact Conditions)*)?
Rule ::= "step" ConstId ("(" Vars ")")?
         ":=" PNFacts Conditions ExistsVars? "=>" Facts
Facts   ::= Fact ("." Fact)*
Conditions ::= ("&" Condition)*
Condition ::=  AtomicCondition | "not(" AtomicCondition ")"
AtomicCondition ::=  "equal(" Term "," Term ")"
                     | "leq(" Term "," Term ")"

% PNFact stands for Possibly Negated Fact
PNFacts ::= PNFact ("." PNfact)*
PNFact ::= Fact | "not(" Fact ")"
% A Fact must be a Term of type 'fact'
Fact ::= Term
ExistsVars ::= "=[exists" Vars "]"
```

```
Constraint::= "constraint" ConstId ("(" Vars ")")? ":=" Formula

Goal ::= LTLGoal | AttackStates

LTLGoal ::= "goal" ConstId ("(" Vars ")")? ":=" Formula
AttackStates ::= "attack_state" ConstId ("(" Vars ")")? ":="
                 PNFacts Conditions

Formula ::= Fact |
            "equal(" Term "," Term ")" |
            "not" "(" Formula ")" |
            "and" "(" Formula "," Formula ")" |
            "or" "(" Formula "," Formula ")" |
            "implies" "(" Formula "," Formula ")" |
            "forall" Vars "." Formula |
            "exists" Vars "." Formula |
            LTLOp1 "(" Formula ")" |
            LTLOp2 "(" Formula "," Formula ")"

% neXt | Yesterday | Finally | Once | Globally | Historically
LTLOp1 ::= "X" | "Y" | "F" | "O" | "G" | "H"
% Until | Release | Since
LTLOp2 ::= "U" | "R" | "S"

Equation ::= Term "=" Term
% The number and types of Terms must match
% the declared arity of OpId
Term ::= Const | VarId | OpId "(" Terms ")"

% In TypeDecl of the form "VarId : Type" Type cannot be 'fact'
TypeDecl ::= VarConsts ":" Type
Type ::= TypeId | OpId "(" Types ")" | "{" Consts "}"

SuperType ::= TypeId ">" TypeId
OpDecl ::= OpId ":" TypeStar "->" Type
TypeStar ::= Type | Type "*" TypeStar

Var ::= VarId
Vars ::= Var ("," Var )*
VarConsts ::= (Var | Const) ("," (Var | Const))*
Terms ::= Term ("," Term )*
Types ::= Type ("," Type )*
```

```
OpId ::= ConstId
TypeId ::= ConstId

VarId ::= [A-Z_][a-zA-Z0-9_]*
ConstId ::= [a-z][a-zA-Z0-9_]*
Numeral ::= "0" | [1-9][0-9]*
Const ::= Numeral | ConstId
Consts ::= Const ("," Const )*
```

A.2.2 Structure of an ASLan File

An ASLan specification (and, similarly, a prelude file) consists of a sequence of sections.

Section Type Symbols (`TypeSymbolsSection`). In this section, all basic (message) types are declared, for example `nonce`.

In the sections signature (`SignatureSection`) and types (`TypesSection`), the types of variables, constants, function symbols and fact symbols are declared. See also Section A.3.4 for further discussions on types in ASLan.

Section Signature (`SignatureSection`). This section contains declarations of the used function and fact symbols, and, more specifically, their types. It also contains supertype declarations.

Section Types (`TypesSection`). In this section, the types for all constants and variables can be specified. This implies that throughout an ASLan file an identifier cannot be used with two different types (while the scope of each variable is limited to the rule it appears in).

Notice that a declaration of the form $\mathtt{M} : op(t_1, ..., t_n)$ is equivalent to the declarations $\mathtt{M}_1 : t_1, \ldots, \mathtt{M}_\mathtt{n} : t_n$ where $\mathtt{M}_i$ (with $i = 1, \ldots, n$) are fresh variables (i.e. that do not appear in the ASLan file) and every occurrence of `M` in the ASLan file is replaced with the term $op(\mathtt{M}_1, \ldots, \mathtt{M}_\mathtt{n})$.

Section Equations (`EquationsSection`). The equations contained in this section define the algebraic properties of the function symbols.

The sections inits (`InitsSection`), rules (`RulesSection`), intruder section (`IntruderSection`), and clauses (`lausesSection`) describe the service as a transition system, the section constraints (`ConstraintsSection`)[45] describes the constraints the transition system will always enjoy, and the section goals (`GoalsSection`) describes the security goals or the attack states.

[45] This is so far supported only by SATMC.

Section Inits (`InitsSection`). In this section, we specify one or more initial states of the service.

Section Rules (`RulesSection`). This section specifies the transition rules of the honest agents executing the service.

Rules may contain positive facts on both their LHS and RHS, while conditions and negative facts may only appear on the LHS.

A rule can be labeled with a list of existentially quantified variables. Their purpose is to introduce new constants representing fresh data (e.g. nonces).

Section Clauses (`ClausesSection`) A finite set of clauses is given in this section. These describe, for instance, the authorization logic.

Section Constraints (`ConstraintsSection`). Constraints can be defined as LTL formulae. We assume that the variables occurring in the constraints and in the LTL formulas in the goal section are disjoint.

Section Goals (`GoalsSection`). Security goals can be defined as attack states or by means of LTL formulas.

Section Intruder (`IntruderSection`). The rules and clauses in this section describe the abilities of the intruder, namely composition and decomposition of known messages. As these abilities are again independent of the service, they are included in the prelude.

A.2.3 Constraints on identifiers

All used identifiers must be different from the ASLan keywords (`step`, `section`, `intruder`, `equal`, `leq`, `not`, `state`). The identifiers for types used in declarations can only be those identifiers that have been introduced as type identifiers in the prelude. Identifiers for operators are only those that have been declared in the signature section of the prelude as having range type message. Similarly, fact symbols are only the ones declared in the signature section of the prelude or the ASLan file as having range type fact. The identifiers that name initial states, rules, or goals must be unique and distinct from all constants and variables and declared identifiers.

A.2.4 Constraints on variables

For each rule declaration, the variables in the parameter list must contain exactly those variables that occur on the LHS and in the `exists` clause of the rule. The variables appearing in the `exists` clause must be disjoint from the variables appearing in the LHS. The variables of the RHS must be a subset of the variables in the positive facts of the LHS (excluding those variables that occur only in the conditions or the negative facts of the rule) and the existentially quantified variables. Analogous restrictions apply for initial states. Further, variables cannot occur in an initial state as it can be seen as the RHS of a transition rule with an empty LHS.

For each declaration of an LTL goal, attack state, clause, or LTL constraint, the parameter list must contain those variables that occur freely in the formula.

For each clause declaration, variables on the LHS of the "`:-`" that do not appear in the RHS, if any, must be declared in the prefix `forall` *var_list*.. They are interpreted as *universally quantified* over their type set.

A.2.5 Constraints on fact symbols

We partition the set of predicates defined in `SignatureSection` into *state* predicates and *policy* predicates: the predicates appearing in the head of clauses are called policy predicates; the rest of the predicates are called state predicates.

In any transition rule of an ASLan specification, policy predicates are seen as conditions. This implies that they must not appear in the RHS of a transition rule. The predicate `iknows`, used to model the intruder knowledge, is treated special in the following sense: while facts of the form `iknows(M)` appearing on the LHS are seen as conditions, they may also appear on the RHS of rules. Note that if a policy predicate or `iknows` fact appears only on the LHS of a transition rule, this cannot mean retraction of the fact, but its usage as a condition (cf. ASLan execution model, § A.3.2).

A.2.6 ASLan prelude

Here we give the entire specification of a typical ASLan prelude file, that reflects the abilities of the Dolev-Yao intruder. Moreover it includes symbols, facts, rules, and constraints necessary to express the three different models of communication channels (ICM, CCM, ACM).

```
% PRELUDE ASLan

section typeSymbols:

  agent, text, symmetric_key, public_key, private_key, hash_func,
  message, fact, nat, protocol_id, set,
  slabel, % for CCM
  channel % for ACM

section signature:

  message > agent
  message > text
  message > hash_func
  message > nat
  message > protocol_id
  message > symmetric_key
  message > public_key
  agent    > private_key

  pair      : message * message -> message
  crypt     : message * message -> message
  inv       : public_key -> private_key
  scrypt    : symmetric_key * message -> message
  sign      : private_key * message -> message
  exp       : message * message -> message
  xor       : message * message -> message
  apply     : message * message -> message
  s         : nat -> nat % used by SATMC

  iknows : message -> fact     % send/receive on CCM (or unprotected ICM) channel
  contains : message * set -> fact
  witness  : agent * agent * protocol_id * message -> fact
  request  : agent * agent * protocol_id * message * nat -> fact
  wrequest : agent * agent * protocol_id * message * nat -> fact
  secret   : message * protocol_id * agent set -> fact

  % used for implementing CCM
  ak        : agent -> public_key % noninvertible authentication  key function
  ck        : agent -> public_key % noninvertible confidentiality key function

  % used for implementing ICM
  athCh   : agent * agent * message -> fact % message on authentic channel
  cnfCh   :         agent * message -> fact % message on confidential channel
  secCh   : agent * agent * message -> fact % message on secure channel

  % used for implementing ACM
  sent      : agent * agent * agent * message * channel -> fact
  rcvd      :         agent * agent * message * channel -> fact
  confidential_to      : channel * agent   -> fact
  weakly_confidential : channel           -> fact
  authentic_on         : channel * agent   -> fact
  weakly_authentic     : channel           -> fact
  link                 : channel * channel -> fact
  resilient            : channel           -> fact


section types:

  K,M,M1,M2,M3 : message
  0,1,2,3, ...: nat
  i: agent
  Xvar: nat % used by SATMC

  % used for implementing CCM
  atag : slabel; % authentication  tag
  ctag : slabel; % confidentiality tag
  stag : slabel; % security (A+C)  tag


section equations:

  pair(M1,pair(M2,M3)) = pair(pair(M1,M2),M3)

  inv(inv(M)) = M

  exp(exp(M1,M2),M3) = exp(exp(M1,M3),M2)
  exp(exp(M1,M2),inv(M2)) = M1

  xor(M1,xor(M2,M3)) = xor(xor(M1,M2),M3)
  xor(M1,M2) = xor(M2,M1)
  xor(xor(M1,M1),M2) = M2


section intruder:

  % generate HCs for intruder

  hc gen_pair (M1,M2) :=
     iknows(pair(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_crypt (M1,M2) :=
     iknows(crypt(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_scrypt (M1,M2) :=
     iknows(scrypt(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_exp (M1,M2) :=
     iknows(exp(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_xor (M1,M2) :=
     iknows(xor(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_apply (M1,M2) :=
     iknows(apply(M1,M2)) :- iknows(M1), iknows(M2)

  % analysis HCs for intruder

  hc ana_pair1 (M1,M2) :=
     iknows(M1) :- iknows(pair(M1,M2))
  hc ana_pair2 (M1,M2) :=
     iknows(M2) :- iknows(pair(M1,M2))
  hc ana_crypt (K,M) :=
     iknows(M) :- iknows(crypt(K,M)), iknows(inv(K))
  hc ana_scrypt (K,M) :=
     iknows(M) :- iknows(scrypt(K,M)), iknows(K)
  hc ana_sign (K,M) :=
     iknows(M) :- iknows(sign(inv(K,M))), iknows(K)

  step generate (M) :=
    =[exists M]=> iknows(M)

  % used for implementing ICM
  step ICM_rule1(A,B,M) :=
       iknows(B). iknows(M). dishonest(A) => athCh(A,B,M)
  step ICM_rule2(A,B,M) :=
       athCh(A,B,M) => iknows(M)
  step ICM_rule3(B,M) :=
       iknows(B). iknows(M) => cnfCh(B,M)
  step ICM_rule4(B,M) :=
       cnfCh(B,M). dishonest(B) => iknows(M)
  step ICM_rule5(A,B,M) :=
       iknows(B). iknows(M). dishonest(A) => secCh(A,B,M)
  step ICM_rule6(A,B,M) :=
       secCh(A,B,M). dishonest(B) => iknows(M)

  % used for implementing ACM
  step fake(OS,R,M,Ch):=
       iknows(M). iknows(OS). iknows(R) => sent(i,OS,R,M,Ch)
  step intercept(A,R,M,Ch):=
       sent(A,A,R,M,Ch) => rcvd(i,A,M,Ch). iknows(M)
  step overhear(A,R,M,Ch):=
       sent(A,A,R,M,Ch) => sent(A,A,R,M,Ch). rcvd(i,A,M,Ch). iknows(M)


section constraints:

constraint ACM_confidential_channels :=
forall Ch. forall P. forall A. forall B. forall M.
 G(implies(confidential_to(Ch,P),
  G(implies(rcvd(B,A,M,Ch),equal(B,P)))))
```

```
constraint ACM_weakly_confidential_channels :=
forall Ch.
forall A . forall B . forall M .
forall A’. forall B’. forall M’.
 G(implies(weakly_confidential(Ch),
  G(implies(and(rcvd(A,B,M,Ch), F(rcvd(A’,B’,M’,Ch))),
            equal(A,A’)))))

constraint ACM_authentic_channels :=
forall Ch. forall P. forall RS. forall A. forall B. forall M.
 G(implies(authentic_on(Ch,P),
  G(implies(sent(RS,A,B,M,Ch),
          and(equal(A,P), equal(RS,P))))))

constraint ACM_weakly_authentic_channels :=
forall Ch.
forall RS . forall A . forall B . forall M.
forall RS’. forall A’. forall B’. forall M’.
 G(implies(weakly_authentic(Ch),
  G(implies(and(sent(RS,A,B,M,Ch), F(sent(RS’,A’,B’,M’,Ch))),
          and(equal(A,A’), equal(RS,RS’))))))

constraint ACM_resilient_channels :=
forall Ch. forall RS. forall A. forall B. forall M.
 G(implies(resilient(Ch),
  G(implies(sent(RS,A,B,M,Ch),
          F(rcvd(B,A,M,Ch))))))

constraint ACM_link_channels :=
forall Ch1. forall Ch2.
forall RS. forall A. forall B . forall M.
forall R .           forall B’. forall M’.
 G(implies(link(Ch1,Ch2),
  G(implies(and(F(sent(RS,A,B,M,Ch1)), F(rcvd(R,B’,M’,Ch2))),
            equal(RS,R)))))
```

A.3 ASLan Semantics

A.3.1 Equalities

All terms in ASLan, including facts, are interpreted in the quotient algebra $\mathcal{T}_\Sigma/_E$, where E is the set of algebraic equations declared in the prelude specification. Thus, in the following, we consider two terms as equal if and only if this is a consequence of the algebraic equations.

To distinguish from syntactical equivalence of terms, we write $t \approx s$ for two equivalent terms t and s. Also, we assume in the following that the type declarations of the ASLan file are satisfied in all substitutions of variables, e.g. variables of type agent are only substituted for constants (or other terms) of type agent.

A.3.2 Execution model

Let $\mathcal{F}$ be the set of ground (i.e. variable free) facts. An ASLan specification defines a transition system

$$M = \langle \mathcal{S}, \mathcal{I}, \rightarrow \rangle \ ,$$

where $\mathcal{S}$ is the set of states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the (reflexive) transition relation.

The states in $\mathcal{S}$ are represented by sets of ground facts, i.e. $\mathcal{S} = 2^{\mathcal{F}}$. If $S \in \mathcal{S}$, then we interpret the ground facts in S as the propositions holding in that state, all other ground facts being false (closed-world assumption). $\mathcal{I}$ is defined by the `InitsSection` in the obvious way.

Let C be a conjunction of (possibly negated) atomic and ground conditions. We say that C *holds* if and only if

1. C is of the form $\texttt{equal}(t_1, t_2)$ and $t_1 \approx t_2$,
2. C is of the form $\texttt{leq}(t_1, t_2)$ where the ground terms t_1 and t_2 evaluate as expected to n_1 and $n_2 \in \mathbb{N}$ and $n_1 \leq n_2$,
3. C is of the form $\texttt{not}(c_0)$ and c_0 does not hold, and
4. C is of the form $c_1 \,\texttt{\&}\, c_2$ and both c_1 and c_2 hold.

Let $\mathbf{H}$ be the set of clauses defined in the section `ClausesSection`. For a state S, *the closure of S with respect to a set of clauses H*, denoted $\lceil S \rceil^H$, is the smallest set containing S and satisfying the following property:

$$\forall (A \leftarrow A_1, \cdots, A_n) \in H, \forall \sigma. \bigcup_{1 \leq i \leq n} A_i\sigma \subseteq \lceil S \rceil^H \implies A\sigma \in \lceil S \rceil^H$$

where σ is any substitution function mapping the set of variables in $var(A) \cup \bigcup_{1 \leq i \leq n} var(A_i)$ to the set of ground terms.

The transition relation $\rightarrow$ is defined as follows:

- for all S, $S \rightarrow S$ (which models stuttering), and
- for all S, $S \rightarrow S'$, with S' as follows, if and only if there exists a rule

$$PF.NF\&PC\&NC \ {=}[V]{\Rightarrow}\ R$$

in the section `RulesSection` (where PF and NF are sets of positive state facts and negated state facts respectively, PC is a conjunction of positive atomic conditions, and NC is a conjunction of negative atomic conditions) and a substitution $\sigma : \{v_1, \ldots, v_n\} \to T_\Sigma$, where $v_1, \ldots, v_n$ are the variables that occur in PF or in PC, such that

1. $PF\sigma \subseteq \lceil S \rceil^{\mathbf{H}}$,
2. $PC\sigma$ holds,
3. $NF\sigma\sigma' \cap \lceil S \rceil^{\mathbf{H}} = \emptyset$ for all substitutions σ' such that $NF\sigma\sigma'$ is ground,
4. $NC\sigma\sigma'$ holds for all substitutions σ' such that $NC\sigma\sigma'$ is ground, and
5. $S' = (S \setminus PF\sigma) \cup R\sigma\sigma''$, where σ'' is any substitution such that for all $v \in V$, $v\sigma''$ does not occur in S or in any algebraic equation declared in the prelude specification.
   Note that state facts (also known as "explicit facts" in ASLan++), as well as `iknows` facts, that occur only in R are introduced, while state facts that occur only in PF are retracted. Policy facts (also known as "implicit facts" in ASLan++) cannot occur in PF or R and thus are neither introduced nor retracted by the execution of a transition rule. Moreover, `iknows` facts, which may occur in conditions and/or in R, are never retracted.

We remark that the closure of states, w.r.t. clauses, is computed "implicitly" here, in contrast to the "explicit" closure semantics which was given in Deliverable D2.1 [1]. The difference between the semantics is best shown via an example. Let us consider a state $s = \{a\}$, and the clause $c \leftarrow b$, with a, b and c being ground facts. Note that $\lceil s \rceil = \{a\}$, w.r.t. the clause above. Now, given the rewrite rule $a \Rightarrow b$, s is rewritten to $s_1 = \{b\}$ with $\lceil s_1 \rceil = \{b, c\}$ in the implicit closure semantics, and s is rewritten to $s_1' = \{b, c\}$ in the explicit closure semantics. The difference between these semantics becomes evident now by considering the following rewrite rule: $b \Rightarrow a$. Then, s_1 is rewritten to $s_2 = \{a\}$ with $\lceil s_2 \rceil = \{a\}$ in the implicit closure semantics, while s_1' is rewritten to $s_2' = \{a, c\}$. Note that in the implicit closure semantics, c is removed from the closure of s_2 because the "cause" for c, namely b, is not contained in s_2. In explicit closure semantics, however, c is present in s_2', even after b is removed. This feature of the implicit closure semantics enables us to model policy revocations in a natural way in ASLan.

A.3.3 Security goals

A simple way to describe safety properties of a transition system is by defining a subset of so-called "bad" states or *attack* states. The attack states specification in ASLan is syntactically similar to a rule, only that there is no right-hand side. The declaration of an attack state A amounts to adding a rule $A \Rightarrow A.attack$ for a nullary fact symbol *attack* and defining every state that contains *attack* to be an attack state.

Another way to specify in ASLan the trust and security properties is by using LTL goals. To this end, we first consider the following definitions. A *path* π is a sequence of states

$S_0 S_1 \ldots$ such that $S_i \to S_{i+1}$ for $i \in \mathbb{N}$. We define $\pi(i) = S_i$ for all $i \in \mathbb{N}$. If $S_0 \subseteq \mathcal{I}$, then we say that the path is *initialized*. Let π be an initialized path of the execution model M, H be a set of clauses, and $\sigma : \mathcal{V} \to T_\Sigma$. An LTL formula ϕ *is valid on* π *under* σ *w.r.t.* H, in symbols $\pi \models^H_\sigma \phi$, if and only if $(\pi, 0) \models^H_\sigma \phi$, where $(\pi, i) \models^H_\sigma \phi$ is inductively defined as follows. We suppress the superscript H when it is clear from the context.

$$\begin{array}{lll}
(\pi,i) \models^H_\sigma \phi & \text{iff} & \phi\sigma \in \lceil \pi(i) \rceil^H \text{ for } \phi \text{ a fact} \\
(\pi,i) \models_\sigma \texttt{equal}(t_1,t_2) & \text{iff} & t_1\sigma \approx t_2\sigma \\
(\pi,i) \models_\sigma \texttt{not}(\phi) & \text{iff} & (\pi,i) \not\models_\sigma \phi \\
(\pi,i) \models_\sigma \texttt{or}(\phi,\psi) & \text{iff} & (\pi,i) \models_\sigma \phi \text{ or } (\pi,i) \models_\sigma \psi \\
(\pi,i) \models_\sigma \texttt{X}(\phi) & \text{iff} & (\pi,i+1) \models_\sigma \phi \\
(\pi,i) \models_\sigma \texttt{Y}(\phi) & \text{iff} & i > 0 \text{ and } (\pi,i-1) \models_\sigma \phi \\
(\pi,i) \models_\sigma \texttt{F}(\phi) & \text{iff} & \text{there exists } j \geq i \text{ such that } (\pi,j) \models_\sigma \phi \\
(\pi,i) \models_\sigma \texttt{O}(\phi) & \text{iff} & \text{there exists } j, \text{ with } 0 \leq j \leq i, \text{ such that } (\pi,j) \models_\sigma \phi \\
(\pi,i) \models_\sigma \texttt{U}(\phi,\psi) & \text{iff} & \text{there exists } j \geq i \text{ such that } (\pi,j) \models_\sigma \psi \text{ and } (\pi,k) \models_\sigma \phi \text{ for} \\
 & & \text{all } k \text{ such that } i \leq k < j \\
(\pi,i) \models_\sigma \texttt{S}(\phi,\psi) & \text{iff} & \text{there exists } j, \text{ with } 0 \leq j \leq i, \text{ such that } (\pi,j) \models_\sigma \psi \text{ and} \\
 & & (\pi,k) \models_\sigma \phi \text{ for all } k \text{ such that } j < k \leq i \\
(\pi,i) \models_\sigma \texttt{exists}(x,\phi) & \text{iff} & \text{there exists } t \in T_\Sigma \text{ such that } (\pi,i) \models_{\sigma[t/x]} \phi
\end{array}$$

The semantics of the remaining connectives readily follows from the following equivalences: $\texttt{and}(\phi,\psi) \equiv \texttt{not}(\texttt{or}(\texttt{not}(\phi),\texttt{not}(\psi)))$, $\texttt{implies}(\phi,\psi) \equiv \texttt{or}(\texttt{not}(\phi),\psi)$, $\texttt{G}(\phi) \equiv \texttt{not}(\texttt{F}(\texttt{not}(\phi)))$, $\texttt{R}(\phi,\psi) \equiv \texttt{not}(\texttt{U}(\texttt{not}(\phi),\texttt{not}(\psi)))$, $\texttt{H}(\phi) \equiv \texttt{not}(\texttt{O}(\texttt{not}(\phi)))$, $\texttt{forall}(x,\phi) \equiv \texttt{not}(\texttt{exists}(x,\texttt{not}(\phi)))$.

Given an ASLan specification, let $\phi_1, \ldots, \phi_n$ be the set of all LTL goal formulas and $\psi_1, \ldots, \psi_m$ be the set of LTL formulas occurring in the constraints section. Let Ψ be $\texttt{and}(\psi_1, \texttt{and}(\ldots, \psi_m)\ldots)$ and Φ be $\texttt{and}(\texttt{G}(\texttt{not}(\mathit{attack})), \texttt{and}(\phi_1, \texttt{and}(\ldots, \phi_m)\ldots))$. The ASLan specification is *valid* if and only if Φ is valid on π under σ w.r.t. H under the assumption Ψ, – in symbols: $\pi \models^H_\sigma \texttt{implies}(\Psi, \Phi)$ – for all interpretations $\sigma : \mathcal{V} \to T_\Sigma$, and all initialized paths π, where H is the set of clauses defined in the specification. Note that for the validity of a specification it makes no difference whether the constraints Ψ are interpreted (as written above) as constraining all goals and attack states, or if they are interpreted as constraining system execution (i.e. the model itself). The aim is actually to do the latter, which allows for a more efficient implementation because traces not conforming to the constraints do not need to be produced and checked for violations of Φ.

A.3.4 Typing

Types in our specification language serve two main purposes. First, like in programming languages, types can be very helpful to avoid mistakes in the specification. Second, typing can help the back-ends to reduce the size of the transition system being analyzed by excluding the ill-typed actions of the intruder. While this in general means a restriction that can exclude attacks, the correctness of a typed model implies the correctness of its untyped version [12]. . Moreover, the user may also choose to deliberately neglect type-flaw attacks.

A typed specification can be analyzed in an untyped model, i.e., the specification contains the types to check that the specification is well-formed, but the validation is performed ignoring the types.

For these reasons, we formally define the type system of an ASLan as follows:

- If a model contains a set of basic types $\beta_1, \ldots, \beta_n$ such as `agent` and `nonce`, we assume that there exists an infinite number of constants of each such type and write $\mathcal{C}_{\beta_i}$ for each such set.

- We assume that $\mathcal{C}_{\beta_1} \cap \mathcal{C}_{\beta_2} = \emptyset$ for $\beta_1 \neq \beta_2$. Note that $\mathcal{C}_\beta$ is only part of all values that have type β (this latter set is defined below as $\mathcal{L}_\beta$) but rather the set of constants whose *primary* type is β (but, due to sub-typing, a constant may be part of other types as well). The disjointness of the $\mathcal{C}_\cdot$ ensures that each constant has a unique primary type. For example, the "main" message type `msg` is a basic type in the sense of this definition, and the type `agent` is a subtype of it, even though $\mathcal{C}_{\texttt{agent}} \cap \mathcal{C}_{\texttt{msg}} = \emptyset$.

- For every declaration $c : \beta$ that constant c has type β in a given ASLan specification, we assume that $c \in \mathcal{C}_\beta$.

- The only non-basic types in our setting are sets and tuples. We do not allow constants of non-basic types such as `msg set`.

- We now inductively define the set $\mathcal{L}_\tau$ of terms that have type τ, i.e. each $\mathcal{L}_\tau$ is the least set that satisfies the following properties:
  - First, for all basic types $\mathcal{C}_\beta \subseteq \mathcal{L}_\beta$.
  - Then, for a function f that has type $\tau_1 \times \ldots \tau_n \rightarrow \tau$ and any terms $t_1 \in \mathcal{L}_{\tau_1}, \ldots, t_n \in \mathcal{L}_{\tau_n}$, we have $f(t_1, \ldots, t_n) \in \mathcal{L}_\tau$.
  - If $\tau_1 < \tau_2$ (sub-typing), then $\mathcal{L}_{\tau_1} \subseteq \mathcal{L}_{\tau_2}$. (This is extending $\mathcal{L}_{\tau_2}$.)
  - If $S \subseteq \mathcal{L}_\tau$ and S is finite, then $S \in \mathcal{L}_{\tau\ \mathsf{set}}$.
  - Finally, $\mathcal{L}_{\tau_1,\ldots,\tau_n} = \mathcal{L}_{\tau_1} \times \ldots \times \mathcal{L}_{\tau_n}$.

- Each variable of type τ can be assigned to any term of the set $\mathcal{L}_\tau$, and we rule out all other interpretations.

- We require that $\mathcal{L}_\tau$ is closed under $\approx$, i.e. the algebraic properties do not imply the equality of terms of different types.

- As a consequence, for instance, exclusive or, denoted $\oplus$, should be a function of type $\texttt{msg} \times \texttt{msg} \rightarrow \texttt{msg}$, and not a polymorphic one of type $\alpha \times \alpha \rightarrow \alpha$. The reason is the type of the neutral element e: it must be of type `msg`.

- Polymorphic types are disallowed for constants or variables, and only allowed for functions, e.g. $\mathit{contains}(\alpha\ \texttt{set}, \alpha) : \texttt{fact}$. This affects the definition of all $\mathcal{L}_\tau$ where τ is an instance of the return type: $f(t_1, \ldots, t_n) \in \mathcal{L}_\tau$ for every $t_1 \in \mathcal{L}_{\tau_1}, \ldots, t_n \in \mathcal{L}_{\tau_n}$ for all $f(\tau_1, \ldots, \tau_n) : \tau$.

Finally, we allow for a special kind of typing that is merely a syntactical abbreviation: *compound types*. This concept has been first introduced in HLPSL/IF [9] and allows one to specify the format of message terms as a type. For instance, we may declare

`M : crypt(public_key,pair(agent,msg))`.

This constrains the set of values that can be substituted for `M` to those of the appropriate message format. More generally, the declaration `X: f(t1, ..., tn)` is just an abbreviation for `X1:t1, ..., Xn:tn` for new variables `Xi` and replacing all occurrences of `X` in its syntactical scope with `f(X1,...,Xn)`. This replacement must, of course, be repeated recursively if any of the `ti` is itself a compound type.

A.3.5 Micro and Macro Steps

For the translation from ASLan++ to ASLan, we consider a concept of micro-steps and macro-steps[46], where the micro-steps represent the intermediate states of an honest agent in a longer computation that we like to abstract from, as explained in § 4.14.

To support this concept in ASLan, we introduce a new fact *state*! that is similar to the normal state fact but will be used for local states of honest agents within a compressed (i.e. micro-stepping) section. We require transition rules to have at most one *state*! fact on either side, and that the initial state does have a *state*! fact.

The *micro-step transition relation* is now the "standard" transition relation → of ASLan as defined above. We define the new *macro-step transition relation* ↠ based on the micro steps as follows: $S_1 \twoheadrightarrow S_n$ iff there exist $S_2, \ldots, S_{n-1}$ such that

- $S_1 \to S_2 \to \ldots \to S_{n-1} \to S_n$,
- S_i contains exactly one *state*! fact for $1 < i < n$,
- the *state*! fact is not the same in two consecutive S_i (so there is progress in the process represented by the *state*!), and
- S_1 and S_n do not contain a *state*! fact.

All LTL-goals are now interpreted with respect to the macro-step transition relation, i.e. the formulae are "blind" for the micro steps.

Note that our definition does not allow for "partial" macro-steps: suppose that by → we can get into a state with a `state!` fact where no transition rule allows further progress of this intermediate state (i.e. a process is "stuck" within micro-steps), then there simply is no corresponding macro step according to our definition.

[46] These concepts have not yet been implemented in the back-ends.

B Connector and Model Checker Options

This appendix gives a brief overview on how to fine-tune the ASLan++ connector and the model checking back-ends by using various options.

The ASLAN++ connector and the model checkers are available in both an online and offline version. In the *offline* version you supply the options in the command line. This method used in this chapter, when describing the available options of the various tools. In the *online* version you may set the required options in the `Set Options` form field. In case of the model checkers you may also include the options in a model checker *directive* at the very beginning of the ASLan++ model definition. Here as a kind of comment each model checker can be assigned options that are used, whenever the model is analyzed. For instance `% @clatse(--hc blr)` sets options for CL-AtSe, whenever that model checker analyzes this model. With `satmc` the model checker SATMC can be addressed, for OFMC we must use `ofmc`.

For more details regarding the operation of the ASLan++ connector and the model checkers, please refer to Deliverable 4.2 [6]. The AVANTSSAR Tools are available for download on the AVANTSSAR home page at http://www.avantssar.eu/ - Menu Item `Avantssar Platform`. There you'll also find directions to online versions of the AVANTSSAR Tools.

B.1 ASLan++ Connector

This section sketches how to use the ASLan++ connector and the various options available to fine-tune the behavior of the ASLan++ connector, as well as the strategies to use special options for the connector with the targeted model checker.

```
Usage: java -jar aslanpp-connector.jar [INPUT_FILE] [-]
[-E (--preprocess)] [-ar (--analysis-result) ANALYSIS_RESULT]
[-gas (--goals-as-attack-states)]
[-ltl (--goals-as-ltl-formulas)]
[-gv (--graphviz) DIR]
[-h (--help)] [-hc (--horn-clauses-level) [ALL | NPI | NONE]]
[-o (--output) OUTPUT_FILE]
[-opt (--optimization-level) [NONE | NOPS | LUMP]]
[-orch (--orchestration-client) CLIENT_ENTITY_NAME]
[-pp (--pretty-print)] [-so (--strip-output)] [-v (--version)]
```

Available options:

INPUT_FILE : Input file with ASLan++ specification that should be translated to ASLan, or with the ASLan specification for which the analysis result should be translated to ASLan++.

- : Read input from stdin instead of from a file.

-E (--preprocess) : Perform only preprocessing of the ASLan++ specification. Macros are expanded and a detailed representation of the specification is returned. Hidden symbols, added automatically during the translation, are also shown, so the specification that is returned cannot be used again as a valid ASLan++ specification. This options is intended only for debugging purposes.

-ar (--analysis-result) ANALYSIS_RESULT : Translate back to ASLan++ the analysis result given by a backend on an ASLan specification that was generated by translation from ASLan++. A file must be provided that holds the analysis result given by the validation platform.

-gas (--goals-as-attack-states) : Request rendering ASLan++ goals as ASLan attack states whenever possible. This is the default. When an ASLan++ goal cannot be rendered as an ASLan attack state (for example if it uses non-trivial LTL operators) then it will be rendered as an ASLan goal and a warning will be issued.

-ltl (--goals-as-ltl-formulas) : Force rendering ASLan++ goals as ASLan LTL goals. Backends typically have trouble with this.

-gv (--graphviz) DIR : Output Graphviz `.dot` files with the transition rules of each entity. The files are written to the specified directory. If the '`dot`' program is available in the `PATH` then also output `.png` files with the transitions rules. Otherwise you can convert the `.dot` files into images manually. See `http://www.graphviz.org/` for details.

-h (--help) : Show help.

-hc (--horn-clauses-level) (ALL | NPI | NONE) : Makes it possible to leave out from the generated ASLan output some or all of the Horn clauses. Can be one of: `ALL` (all Horn clauses are included in the output), `NPI` (the Horn clauses related to public and invertible functions are left out of the translation) or `NONE` (all Horn clauses are left out of the translation). The default value is `ALL`.

-o (--output) OUTPUT_FILE : Send output to the specified file instead of stdout.

-opt (--optimization-level) (NONE | NOPS | LUMP) : Sets a certain optimization level. Can be one of: `LUMP` (this is the highest level of optimization, which means that the generated ASLan transitions are lumped together as much as possible), `NOPS` (this is an intermediate level of optimization, which means that empty transitions and transitions that can never be taken are eliminated) or `NONE` (this disables any optimization). The default value is `LUMP`.

-orch (--orchestration-client) CLIENT_ENTITY_NAME : Render the specified entity as an orchestration client. The name of the specified entity is changed to '`OrchestrationClient`' and some extra symbols and transitions are added to the resulting ASLan specification.

-pp (--pretty-print) : Pretty-print the ASLan++ specification. Comments are lost during pretty-printing. This option is meant mainly for syntax checking specifications and for getting a compact view of large specifications.

-so (--strip-output) : If this flag is enabled, then no comments and no line information will be generated in the ASLan specification. This is intended for debugging purposes.

-v (--version) : Display version information.

In order for imports to work, you must define an environment variable named `ASLANPATH`. It should contain all directories where the translator should search for files to import. It works similarly with the `CLASSPATH` variable used by Java.

For example if your imported `.aslan++` files are located in the `'../common'` directory, you can set the `ASLANPATH` variable by invoking java with the `-D` argument, like this:

```
  java -DASLANPATH=../common -jar aslanpp-connector.jar ...
```

The current directory is automatically included in the `ASLANPATH`, so there is no need to manually add it.

B.2 SATMC

This section sketches how to use the model checker SATMC and the various options available to fine-tune the behavior of SATMC, as well as the various specific characteristics and idiosyncrasies of this backend.

You can execute SATMC by issuing at the shell prompt:

```
Usage:
satmc [OPTION]
satmc FILE [OPTIONS]
satmc [OPTIONS] -f FILE
```

`FILE` is the ASLan file (comprehensive of the path) you want to analyze.

With SATMC you can use the following options.

-h, --help : display this help and exit.

-v, --version : output version information and exit.

-n MAX, --max=MAX : `MAX` is the maximum depth of the search space up to which SATMC will explore; it can be set to -1 meaning infinite, but in this case the procedure is not guaranteed to terminate; by default it is set to 80.

--enc=ENC : ENC is the selected SAT reduction encoding technique; it can be set to either ’`gp`’ (default value) or ’`ltl-gp`’. The `ltl-gp` encoding supports the specification of security properties and constraints in linear temporal logic (LTL).

--mutex=MUTEX : `MUTEX` is an integer indicating the level of the mutex relations to be used during the SAT-reduction, i.e.: '0': mutex off (absRef on) and '1': mutex on (absRef off). Basically, if `MUTEX` is set to 0, then the abstraction/refinement strategy provided by SATMC is enabled; otherwise the abstraction/ refinement strategy is disabled and the static mutexes are generated; the default value is 0.

-o OUTPUT_FILE : the output is printed in the file `OUTPUT_FILE`, instead of the standard output.

--output_dir=ODIR : `ODIR` is the directory in which SATMC will store temporary files and put the detailed results of the analyzes in case the AVANTSSAR output is enabled; by default these files will be put in the same directory in which FILE is contained.

--of=OF : OF is a string indicating the output format selected: 'aslan': standard output format from the AVANTSSAR project, (default value) or 'if': standard output format from the AVISPA project.

--sc=SC : `SC` is a boolean parameter for enabling or disabling the step-compression optimization; by default it is set to true. Note: in case the Abstract Channel Model (ACM) is used, it must be set to false.

-e, --exec : to enable or disable the check on executability of actions/rules without any intruder. It is very useful to debug the input specification; by default it is set to false.

--prelude=PRELUDE : PRELUDE is the prelude file (comprehensive of the path); by default it is assumed that the prelude file is in the same directory of FILE with the name 'prelude.if' or 'prelude.aslan'.

--no_opt_redirect : disables the optimization for redirect. By default is enabled an optimization that improves the performance when in specification there is a redirection of messages.

--opt_imp : enables an optimization that find faster an attack (if any). With this option enabled SATMC searches an attack on a simplified model in which the intruder is able to do fewer actions. (!) NOTE that using this optimization attacks can be LOST. By default it is not enabled.

--empiric_stop_cond=X : X is a integer that represents the percentage regards the step of graphplan leveled off to set the empiric stop condition (we say that a planning graph has levelled-off when two adjacent fact levels are identical: this means that all the necessary terms have been computed). E.g if X is set to 100 the search of attack stops when the graphplan levels off, by default is set to 200. (!) NOTE that decreasing this value under 150 attacks can be LOST.

--no_empiric_stop_cond : this option disables the `empiric_stop_cond`, by default if an attack is not found until an empirical limit step the protocol analysis return SAFE with a warning.

Reserved keywords While using the tool, consider also that specifications must not contain new definition of default functions, nor misuse reserved keywords that are managed in a special way by the tool.

Keywords reserved for internal usage by SATMC or at which particular meaning has been already assigned, and thus cannot be freely used by users, are:

- The set of all terminal symbols of the ASLan and ASLan++ syntax (step, section, intruder, equal, leq, not, state, goal...). For a complete list see § 3.12 and § A.2.
- The set of all identifiers and symbols defined in the ASLan and ASLan++ prelude.
- `action`, `apply`, `pair`, `witness`, `wrequest`, `request`, `PreludeK`, `PreludeM`, `PreludeM1`, `PreludeM2`, `PreludeM3`, `Xvar`, `s` (`s : nat -> nat`), `has` (`has : agent * message -> fact`).

B.3 OFMC

This section sketches how to use the model checker OFMC and the various options available to fine-tune the behavior of OFMC, as well as the various specific characteristics and idiosyncrasies of this backend.

By launching OFMC with `--help` the following text is displayed:

```
Options:
  --numSess <INT>  specify the number of sessions (for an AnB spec)
  --ot IF          do not check, but produce only AVISPA IF
  --ot Isa         generate a fixedpoint proof for Isabelle-OFMC
  --of if          standard output format from the Avispa project
  --of aslan       Standard output format from the Avantssar
                   project
  --fp             run only in fixedpoint module
  --classic        run only in classic mode
  -o <outputfile>  saves the result of the analysis in a file
  --exec, -e       checks the executability of each rule (do not
                   search for attacks)
  --help, -h       display this help
  --version        display the version

Options for classic mode:
  --theory <TheoryFile>        use a custom algebraic theory
  --sessco                     performs an executability check and
                               session compilation
  --untyped, -u                ignores all type-declarations
  --depth <DEPTH>, -n <DEPTH>  specify maximum search depth/depth
                               first search
  --trace <PATH>, -t <PATH>    (PATH is white-space separated list
                               of ints) specify a path in the
                               search tree to visit by the indices
                               of the successors to follow.
```

By default OFMC makes use of both `classic` and `fixedpoint` modules (see [6] for more information on OFMC components and how they work) and takes into account type declaration while considering the specification. But since the `fixedpoint` module can only check models without composed types, in many cases an error occurs if no options have been set. As the error message says, the more common way to circumvent the problem is to use the `classic` module only (without the `fixedpoint` module) by using the option `--classic`.

A good practice while using OFMC is to use also **`--untyped`** option, because if there are composed types in the specification it is possible to obtain false `NO_ATTACK_FOUND` results (see [6] for more details).

For what concerns **`--depth`** and **`--trace`** options, please note that:

- **`--depth`** sets a limit to the maximum search depth on the tree of reachable states. This obviously implies a limitation on the obtained result, that is obtaining `NO_ATTACK_FOUND` by setting depth to n does not mean that no attack will occur while checking the specification with an higher depth i.e. $n + 1$

- **`--trace`** can be used to select a specific trace in the reachable states tree. Note that the tree may not be balanced, i.e. nodes probably have not a common degree. This means that **`--trace`** `0 3 2 1` may return a partial trace (selecting the first branch starting from the root, then the fourth one, the third one and finally the second available) but the partial trace **`--trace`** `0 3 2 2` may not exist causing OFMC to return an error.

Reserved keywords While using the tool, consider also that specifications must not contain new definition of default functions, nor misuse reserved keywords that are managed in a special way by the tool.

Keywords reserved for internal usage by OFMC or at which particular meaning has been already assigned, and thus cannot be freely used by users, are:

- The set of all terminal symbols of the ASLan and ASLan++ syntax. For a complete list see § 3.12 and § A.2.

- The set of all identifiers and symbols defined in the ASLan++ prelude. For a complete list see § 3.13.

- `apply`, `pair`, `give`, `witness`, `switness`, `wwitness`, `wrequest`, `request`, `srequest`, `error`

B.4 Cl-AtSe

This section sketches how to use the model checker CL-AtSe and the various options available to fine-tune its behavior, as well as the various specific characteristics and idiosyncrasies of this backend.

Properties and Limitations. The Cl-AtSe model checker analyzes Avantssar's ASLan v1 files w.r.t. a bounded number of "sessions". The tool supports associative ("`--assoc`") or non-associative ("`--free`") pairing symbol, as well as the algebraic properties of xor and exponential. There is no need to include such properties in the specification, these are hard-coded in the tool. Also, both the typed ("`--typed`") and untyped ("`--untyped`") models are supported. The tool is backward-compatible with the Avispa project. It has currently two main limitations w.r.t ASLan: horn clauses with an intruder knowledge in the LHS are not supported, except those coding composition or decomposition of functions; and LTL goals are limited to those that the tool can automatically translate to attack states. Besides that, the support is complete.

Horn Clauses. For horn clauses, it is important to know what strategy the tool employ: in Cl-AtSe, this is a backward strategy. This means that horn clauses are used when and only when some fact is required to be build (by e.g. the LHS of a transition). Other tools may employ a forward strategy, where the set of known facts is saturated by horn clauses between each transition. Whatever the strategy used, there are always systems of horn clauses that won't terminate, and thus, the modeler should make sure that the horn clauses in his specification terminate w.r.t the strategy used by the tool used for the analysis.

Bounding Execution. A second very important point to be aware of is the option "`--nb`" which controls the bound on the number of sessions used during the analysis. Except for e.g. OFMC in non-classical mode, this bound is a need to ensure termination anyway. The definition of sessions is controlled by the "`--lvl`" option, but, whatever the definition of session used, the "`--nb`" option defines the maximum number of times each transition can be run in a trace. The default value is '`1`', and thus, it is up to the modeler to correctly adjust this value depending on his specification. For example, if the specification does not use loops, then the default '`1`' value should be fine; if the specification uses loops but the maximum number of iteration is known in advance (e.g. loops used to avoid writing the same transitions many times), then a value sufficiently large must be provided to ensure that the specification can be run completely; and if the specification uses loops in an unpredictable or infinite way, then the problem is fundamentally undecidable and thus, the modeler should interactively increase the bound until an attack is found or the problem gets too big.

Level of Rebuilding. As pointed out above, Cl-AtSe tries by default to extract coherent sessions from the ASLan v1 specification. A session for Cl-AtSe is a (longest) sequence or tree of transitions that is played by one (and only one) agent and where each transition appears only once. The kind of session used by Cl-AtSe is controlled by the option

“--lvl” (’level’ of rebuilding). A ’0’ value is the safe mode where one transition is equal to one session. The values ’1’ and ’2’ (default) are two rebuilding methods and should be equivalent. Identifying sessions within an ASLan v1 file may fail due to the structure of the specification: the syntax does not guarantee that simple sessions can be rebuilt, and moreover, sometimes building sessions is equivalent to enumerating all execution traces. In such situation, the tool will fallback to the default “--lvl 0” option and write a mesage like “Forced use of the --lvl 0 option.”. A successful session rebuilding often means better analysis performances.

Output Formats. Many levels of verbosity, or even output syntaxes, are available with Cl-AtSe. Those kind of outputs are controlled by the “--of” option, which accepts many values: “ASLan” for the standard ASLan output format, as defined in the Avantssar’s deliverables; “if”, “if+” and “if++” are tree levels of verbosity of an Avispa-like output format, which gives more and more information on the way the specification was understood by the tool; “if-” gives a condensed view of an attack found, if any; and “brief” gives an even more elementary view of the result.

Advanced Horn Clause Use. For more advanced users, Cl-AtSe allows you some control on the strategy used for horn clauses. Even if the strategy remains backward only, you can control the kind of recursivity used to help the tool to analyze your specification more efficiently. That is, three variants of the horn clauses strategies are available:

1. Backward with Top-Down recursivity : h.c. are used to produce the facts required by a transition, an attack state, or the RHS of an other h.c., in a Depth-first mode. That is, first finish the deductions of the first fact in the list before doing deductions for the next one. Facts are taken in the order they were written in the ASLan file. Activated by option "--hc btd" (for BackwardTopDown)

2. Backward with Top-Down recursivity, but with modified ordering (default). This is the same as above, but the tool choose himself the ordering of facts to use, i.e. the tool choose which fact must be deduced first. This will reorder the lists of facts so that the more "interesting" facts are tried first. This is an heuristic, but in the worst case still everything is tested. If you want to control the ordering yourself, please use the other option above. This one can be activated with "--hc btd+", and it is the default.

3. Backward with Breadth-first recursivity. Assume you have a set of facts to test. In this mode the tool with make one elementary deduction for each fact, storing the subsequent new facts to produce, and iterate only after all facts have been processed. The advantage is that the ordering is not important anymore in this mode, since you move "layer by layer". The bad consequence however, is that if you are unlucky today, one of your facts may explodes into a huge set of facts before the deductions on an other fact could add the right constraints. However, such conditions are quite particular, and in that case the modeler should choose his fact ordering carefully with "--hc btd" or

thrust the tool with the default "--hc btd+". This option is activated with the option "--hc blr" (for BackwardLeftRight).

To conclude about horn clauses, please note also the existence of an option called “--not_hc”. Due to structural constraints inside the horn clauses functions in Cl-AtSe, sometimes false attacks might appear where negative tests inside horn clauses was validated while they should not. This is due to the fact that testing such negative tests might require to perform an inversion of a formal state, which is a very complex and costly operation. Thus, until now, this operation is done only when the option “--not_hc” is provided, instead of just refusing horn clauses using such tests. Note that no false feeling of security can appear here. This option might be activated by default in a near future.

```
Usage  :  Cl-AtSe [options] [-f file] [files]

  [files]  Name of the IF/ASLanv1 files to process.
           Note : HLPSL files allowed if hlpsl2if is available.
           Multiple files allowed. Default is standard input.
  -f file  Name of a specification file to process. Same as [files].
  -u       Do not take term types into account.
  -o       Write the attack to file.atk
  -v       Print more output informations (same as '--of if+').

Expert/Long options :

  --nb n    Maximum number of times that a transition can be run
            in a trace. The fefault is 3 for IF, and 1 for ASLan.
  --free    Uses a free pairing symbol (default for typed model).
  --assoc   Uses an associative pairing symbol (default for the
            untyped model).
  --typed   Use term types as defined in the spec. (default).
  --untyped Do not take term types into account.
  --short   Perform breath-first search (minimal attack).
  --of str  Select the output format and it's level of verbosity.
            The default value is 'if', and choices are :
     ASLan  Standard output format from the Avantssar project;
     if     Standard output format from the Avispa project;
     if+    Same as 'if' but also shows the protocol specification
            as it was understood;
     if++   Same as 'if+' but show things exactly as they were
            analysed (i.e. after simplification);
     if-    Shows a quick view of the attack trace, if any;
     brief  Shows an even more concise view of the attack trace and
            analysis result.
  --out     Write the attack to file.atk
  --dir d   (string) Output directory to use with -out
  --ns      Do not simplify the input file (complete trace).
  --opt     Try to replace hashing by tokens. (slower or faster
            depending on the protocol) (Default for IF).
  --noopt   Do not try to replace hashing by tokens. (slower or
            faster depending on the protocol) (Default for ASLan).
  --verbose Print more output informations (same as '--of if+').
  --noexec  Do not analyse, only write the specification as it is.
  --col n   Number of column in the output, default 80.
  --bench   Benchmark output format (very concise).
  --ASLan   Expect AVANTSSAR's ASLan in input, use ASLan for output.
  --if      Expect AVISPA's IF as input; Not default anymore.
  --lvl n   Level of protocol rebuilding (see explanation above).
  --store   When an attack is found, store the current state in
            a .state file.
  --split   Create a directory with the spec's name and fill it with
            more than 300 sub state files, so that many different
            instances of Cl-AtSe can analyse them in parallel.
  --hc str  Choses a strategy for Horn clauses. Default is btd+
     btd    Backward strategy with depth-first recursivity;
     blr    Backward strategy with breadth-first recursivity;
     btd+   Backward TopDown plus heuristical choice of the fact to
            descend into first.
  --not_hc  Extend tests for negtive constraints w.r.t Horn Clauses.

Debugging options :

  --max n   (int) Forced activation of -lvl 0 if this number of
            steps is reached (default 80).
  --vv      Print even more output and term informations (debug).
  --tab     Write the correspondence table (debug).
  --par     Write the parser output (debug).
  --heavy   Extend the search for unforgeable atoms.
  --debug   Write debuging informations for Read.ml (debug).
  --split   Use the --split option on hlpsl2if for hlpsl input file.
  --states  Shows all internal system states of the analysis.
  --trace   Use -trace "Short trace" to follow one particular trace.
  --reverse Inverse the order of protocol steps in the read module.

Miscellaneous options :

  --version Print the version number.
  --licence Print the Cl-AtSe's disclaimer (licence).
  -help  Display this list of options
  --help  Display this list of options
```

References

[1] AVANTSSAR. Deliverable 2.1: Requirements for modelling and ASLan v.1. www.avantssar.eu, 2008.

[2] AVANTSSAR. Deliverable 3.3: Attacker models. www.avantssar.eu, 2008.

[3] AVANTSSAR. Deliverable 5.1: Problem cases and their trust and security requirements. www.avantssar.eu, 2008.

[4] AVANTSSAR. Deliverable 2.2: ASLan v.2 with static service and policy composition. www.avantssar.eu, 2009.

[5] AVANTSSAR. Deliverable 2.3: ASLan final version with dynamic service and policy composition. www.avantssar.eu, 2010.

[6] AVANTSSAR. Deliverable 4.2: AVANTSSAR Validation Platform v.2. www.avantssar.eu, 2010.

[7] AVANTSSAR. Deliverable 5.3: AVANTSSAR Library of validated problem cases. www.avantssar.eu, 2010.

[8] AVISPA. Deliverable 2.1: The High-Level Protocol Specification Language. www.avispa-project.org, 2003.

[9] AVISPA. Deliverable 2.3: The Intermediate Format. www.avispa-project.org, 2003.

[10] C. Dilloway and G. Lowe. On the specification of secure channels. In *Proceedings of WITS '07*, 2007.

[11] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[12] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of The 13th Computer Security Foundations Workshop (CSFW'00)*. IEEE Computer Society Press, 2000.

[13] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *Proceedings of TACAS'96*, LNCS, pages 147–166. Springer-Verlag, 1996.

[14] G. 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.

[15] G. Lowe. Casper: a Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998. http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Casper/.

[16] U. M. Maurer and P. E. Schmid. A calculus for security bootstrapping in distributed systems. *Journal of Computer Security*, 4(1):55–80, 1996.

[17] S. Mödersheim. *Models and Methods for the Automated Analysis of Security Protocols*. PhD Thesis, ETH Zurich, 2007. ETH Dissertation No. 17013.

[18] S. Mödersheim and L. Viganò. Secure Pseudonymous Channels. In *Proceedings of Esorics'09*, LNCS 5789, pages 337–354. Springer-Verlag, 2009.

[19] S. Mödersheim and L. Viganò. The Open-source Fixed-point Model Checker for Symbolic Analysis of Security Protocols. In *Fosad 2007-2008-2009*, LNCS 5705, pages 166–194. Springer-Verlag, 2009.

[20] S. Mödersheim and L. Viganò. Channels as Assumptions, Channels as Goals, 2010. Submitted journal paper.

[21] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), december 1978.

[22] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol, Version 1.2. IETF RFC 5246, Aug. 2008.