Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures

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

www.avantssar.eu

Deliverable D2.3
ASLan final version with dynamic service and policy composition

Abstract

This deliverable describes ASLan++, the AVANTSSAR specification language for security-sensitive service-oriented architectures, their associated security policies, and their trust and security properties. In particular, ASLan++ allows for the formal specification of dynamic service and policy composition. The semantics of ASLan++ is defined formally by translation to ASLan, the low-level input language for the back-ends of the AVANTSSAR Platform.

Deliverable details

Deliverable version: *v1.0* Classification: *public*
Date of delivery: *02.07.2010* Due on: *30.06.2010*
Editors: *UNIVR, ETH Zurich, UGDIST, IBM, IEAT, SAP, SIEMENS (principal editors).*
INRIA, UPS-IRIT, OpenTrust (secondary editors) Total pages: *110*

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

Contents

1 Introduction **5**

2 ASLan **8**
2.1 Motivation . . . 8
2.2 ASLan Syntax . . . 9
2.3 ASLan Semantics . . . 14
2.3.1 Typing . . . 16
2.3.2 Micro and Macro Steps . . . 18

3 ASLan++ introduction and syntax **20**
3.1 Introductory example . . . 20
3.2 Specifications . . . 22
3.3 Entities . . . 23
3.4 Facts and Clauses . . . 25
3.5 Declarations . . . 26
3.6 Terms . . . 27
3.7 Channels . . . 27
3.7.1 Channel syntax . . . 28
3.7.2 Channel models . . . 29
3.8 Goals . . . 32
3.8.1 Invariants . . . 32
3.8.2 Assertions . . . 33
3.8.3 Channel goals . . . 33
3.8.4 Secrecy goals . . . 35
3.9 Statements . . . 37
3.10 Guards . . . 37
3.11 Grammar in EBNF . . . 38
3.12 ASLan++ prelude . . . 41

4 ASLan++ semantics **45**
4.1 Preprocessing . . . 45
4.2 Translation of Entities . . . 46
4.3 Translation of Types and Symbols . . . 49
4.4 Translation of Horn Clauses . . . 50
4.5 Translation of Equations . . . 51
4.6 Representing the Control Flow . . . 51
4.7 Translation of Statements . . . 52
4.7.1 Grouping . . . 53
4.7.2 Variable assignment . . . 53

4.7.3 Generation of fresh values . . . 55
4.7.4 Entity instantiation . . . 56
4.7.5 Symbolic entity instantiation . . . 57
4.7.6 Message Transmission . . . 59
4.7.7 Fact introduction . . . 62
4.7.8 Fact retraction . . . 62
4.7.9 Branch . . . 63
4.7.10 Loop . . . 67
4.7.11 Select . . . 68
4.7.12 Assert . . . 70
4.7.13 Channel Goals . . . 71
4.7.14 Secrecy Goals . . . 74
4.8 Translation of Guards . . . 75
4.9 Translation of Terms . . . 76
4.10 Translation of the Body Section . . . 78
4.11 Translation of the Goals Section . . . 78
4.12 Assignment of numbers to step label terms . . . 78
4.13 Step Compression . . . 79
4.14 Optimizations . . . 82
4.14.1 Elimination of empty transitions and redundant guards 83
4.14.2 Merging of transitions . . . 85

5 ASLan++ example 87
5.1 Car Registration . . . 87

6 Conclusion 108

References 109

List of Tables

1 Channel notations in ASLan++ 30
2 LTL operators for specifying goals 33
3 Channel goal symbols in ASLan++ 34
4 Events used in the translation of channel goals and rules used for deriving the protocol ID from the name of channel goals . 73
5 Substitutions done by the `adaptGuard` function 76
6 Translation of goals 79

1 Introduction

This deliverable describes ASLan++ and ASLan, the AVANTSSAR specification languages. ASLan++ is a formal language for specifying security-sensitive service-oriented architectures, the 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.

Apart from consolidating ASLan++ and ASLan, in this deliverable we introduce major extensions to the previous versions of these languages in three directions.

- We define the notion of dynamic policies, and modify the semantics of ASLan (which in turn affects ASLan++) in order to support revocation of policies in a natural way.
- We introduce and support various channel abstractions in ASLan++. Abstract channels can be used for expressing both the goals that a service should achieve and the assumptions upon which a service relies.
- A number of optimization techniques are introduced for translating ASLan++ to ASLan. These techniques have been automated in the translator.

Background on ASLan. The first version of ASLan (called ASLan v.1) was described in deliverable D2.1 [3] and then it was extended (and called ASLan v.1.1) in deliverable D2.2 [6]. 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 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 multiset 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 [3], 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 [5], 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.

Background on ASLan++. The first version of ASLan++ (called ASLan v.2) was introduced in deliverable D2.2 [6]. In this deliverable, we build upon this previous version in particular by adding extensive support for using channels abstractions, and also automating a number of optimization techniques.

ASLan++ is conceptually more high-level than ASLan and its semantics is defined by translation to ASLan. 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 of formal protocol/service specification languages.

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

- Control flow constructs (e.g. `if` and `while`) allow for good readability and conciseness of the specifications, and make the specification job easier for modelers who are already familiar with programming languages.

- Modularity is supported by the use of entities. Each entity is specified separately and can then be instantiated 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.
- Furthermore, ASLan++ provides a simple way to specify communication and service compositionality by using an intuitive notation for channels that may be used both as service assumptions and as service goals.

The AVANTSSAR platform supports ASLan++ via a translator.

Structure of the document. This deliverable is self-contained in terms of ASLan and ASLan++. In § 2, we give a complete description of ASLan. § 3 is an introduction to ASLan++, and its common usage. There we also give the syntax of ASLan++. § 3.12 contains the definitions of several "predefined" ASLan++ features. 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 translator tool that is integrated into the AVANTSSAR Platform. § 5 demonstrates ASLan++ in use: the car registration procedure, a scenario described in Deliverable D5.1 [5] and elaborated in D5.2 [7], is specified in ASLan++. We conclude the deliverable in § 6.

2 ASLan

This section 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 Section 2.1, we motivate why, and describe how, ASLan extends the specification language Intermediate Format (IF) of the AVISPA Tool [9]. In Section 2.2, we give the syntax of ASLan, and we then describe the semantics of the language in Section 2.3.

2.1 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 multiset 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 [5], 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) [4]. ASLan++ natively supports various channels models, as described in § 3.7.

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.

2.2 ASLan Syntax

We first present the entire ASLan language in BNF with the usual conventions. The symbol for comments is %. 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 § 3.12.

```
Prelude ::=  TypeSymbolsSection
             SignatureSection
             TypesSection
             EquationsSection
             IntruderSection

ASLanFile  ::=  SignatureSection
                TypesSection
                InitsSection
                HornClausesSection
                RulesSection
                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*
GoalsSection        ::= "section goals:" Goal*
IntruderSection     ::= "section intruder:"
                        (HornClause | Rule)*
HornClausesSection ::= ("section hornClauses:" HornClause+)?

HornClause ::=  "hc" ConstId ("(" Vars ")")?
                ":=" ("forall" Vars ".")? Fact ":-" Fact ("," Fact)*
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 "]"

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 )*
```

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 2.3.1 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 $\mathtt{M_1} : t_1, \ldots, \mathtt{M_n} : t_n$ 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 $op(\mathtt{M_1}, \ldots, \mathtt{M_n})$.

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 Horn clauses (`HornClausesSection`) describe the service as a transition system, and the section goals (`GoalsSection`) describes the security goals or the attack states.

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.

For the declaration of rules, we also use the following syntactic sugar. We assume that the `network` predicate (`network(M)` means that the message `M` has been sent through the network and is accessible to the intruder) is persistent, in the sense that if a `network` fact holds in a state, then it holds in the successor states (i.e. once sent, messages are always available to the intruder). To simplify the rules, however, we do not write the `network` facts that already appear in the left-hand side of the rules.

Also, 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 HornClauses (`HornClausesSection`) A finite set of Horn clauses is given in this section. These describe, for instance, the authorization logic.

Section Goals (`GoalsSection`). Security goals can be defined as attack states or by means of LTL formulas.

Section Intruder (`IntruderSection`). The rules and Horn 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.

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.

For a rule declaration, the variables in the variable list must contain exactly those variables that occur in the LHS of the rule. 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 rewrite rule with an empty LHS.

For a Horn clause declaration, the variable list must contain exactly those variables that occur in the clause. Variables on the LHS of the "`:-`" (i.e. not appearing in the RHS), if any, must be declared in the prefix `forall X.`, and they are interpreted as *universally quantified* over their type set.

We assume the set of predicates defined in `SignatureSection` can be partitioned into *state* predicates and *policy* predicates, such that the following two conditions hold:

- no policy predicate appears in the RHS of a rewrite rule
- no state predicate appears in the LHS of a Horn clause

In particular, the predicate `iknows`, used to model the intruder knowledge, is considered a policy predicate. For this reason, the intruder deduction model must be specified in terms of a Horn theory, e.g. the usual Dolev-Yao attacker can be formalized with Horn clauses like $\texttt{iknows(X)} :- \texttt{iknows}(\{\texttt{X}\}_\texttt{K}), \texttt{iknows(K)}$, and so on. Additionally, we use a Horn clause `iknows(X) :- network(X)` to make the transmitted terms

(modelled by `network` facts In general, we require that `iknows` facts are never retracted, hence any state fact that directly or indirectly affects the `iknows` facts must be persistent.

2.3 ASLan Semantics

All terms in ASLan 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.

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 $\rightarrow \subseteq S \times S$ is the transition relation. We assume that 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 $\mathbf{H}$ be the set of Horn clauses defined in the section `HornClausesSection`. Let C be a conjunction of possibly negated atomic conditions. If C is ground then we say that C *holds* if and only if *(i)* C is of the form $\texttt{equal}(t_1, t_2)$ and $t_1 \approx t_2$, *(ii)* C is of the form $\texttt{leq}(t_1, t_2)$ where the ground terms t_1 and t_2 evaluate as expected to yield $n_1, n_2 \in \mathbb{N}$ and $n_1 \leq n_2$ holds, *(iii)* C is of the form $\texttt{not}(C)$ and C does not hold, and *(iv)* C is of the form $\texttt{and}(c_1, c_2)$ and both c_1 and c_2 hold.

For a state S, the closure of S with respect to a set of Horn clauses H, denoted $\lceil S \rceil^H$, is the smallest set that contains S and satisfies 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 facts. The transition relation $\rightarrow$ is defined as follows: $S \rightarrow S'$ if and only if there exists a rule

$$PF.NF \ C \ =[V]\Rightarrow \ R$$

in the section `RulesSection` (where PF and NF are sets of facts and negated facts respectively and C is a conjunction of possibly negated 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 only in PF, such that

1. $PF\sigma \subseteq \lceil S \rceil^{\mathbf{H}}$,
2. $NF\sigma\sigma' \cap \lceil S \rceil^{\mathbf{H}} = \emptyset$ for all substitutions σ' such that $NF\sigma\sigma'$ is ground.
3. $C\sigma\sigma'$ holds for all substitutions σ' such that $C\sigma\sigma'$ is ground.
4. $S' = S \setminus PF\sigma \cup R\sigma\sigma''$, where σ'' is any substitution such that $v\sigma''$ does not occur in S or in any algebraic equation for all $v \in V$.

Remark that the closure of states, w.r.t. Horn clauses, is computed "implicitly" here, in contrast to the "explicit" closure semantics which was given in Deliverable D2.1 [3]. The difference between the semantics is best shown via an example. Let us consider a state $s = \{a\}$, and the Horn clause $c \leftarrow b$, with a, b and c being ground facts. Note that $\lceil s \rceil = \{a\}$, w.r.t. the Horn 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 $s'_2 = \{a, c\}$. We 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 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 $i = 0, 1, \ldots$. 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 M, H be a set of Horn 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.

$(\pi, i) \models^H_\sigma \phi$	iff	$\phi\sigma \in \lceil \pi(i) \rceil^H$ for ϕ a fact
$(\pi, i) \models_\sigma \texttt{equal}(t_1, t_2)$	iff	$t_1\sigma \approx t_2\sigma$
$(\pi, i) \models_\sigma \texttt{not}(\phi)$	iff	$(\pi, i) \not\models_\sigma \phi$
$(\pi, i) \models_\sigma \texttt{or}(\phi, \psi)$	iff	$(\pi, i) \models_\sigma \phi$ or $(\pi, i) \models_\sigma \psi$
$(\pi, i) \models_\sigma \texttt{X}(\phi)$	iff	$(\pi, i + 1) \models_\sigma \phi$
$(\pi, i) \models_\sigma \texttt{Y}(\phi)$	iff	$i > 0$ and $(\pi, i - 1) \models_\sigma \phi$
$(\pi, i) \models_\sigma \texttt{F}(\phi)$	iff	there exists $j \geq i$ such that $(\pi, j) \models_\sigma \phi$
$(\pi, i) \models_\sigma \texttt{O}(\phi)$	iff	there exists j, with $0 \leq j \leq i$, such that $(\pi, j) \models_\sigma \phi$
$(\pi, i) \models_\sigma \texttt{U}(\phi, \psi)$	iff	there exists $j \geq i$ such that $(\pi, j) \models_\sigma \psi$ and $(\pi, k) \models_\sigma \phi$ for all k such that $i \leq k < j$
$(\pi, i) \models_\sigma \texttt{S}(\phi, \psi)$	iff	there exists j, with $0 \leq j \leq i$, such that $(\pi, j) \models_\sigma \psi$ and $(\pi, k) \models_\sigma \phi$ for all k such that $j < k \leq i$
$(\pi, i) \models_\sigma \texttt{exists}(x, \phi)$	iff	there exists $t \in T_\Sigma$ such that $(\pi, i) \models_{\sigma[t/x]} \phi$

The semantics of the remaining connectives readily follows from the following equivalences: $\texttt{and}(\phi, \psi) \equiv \texttt{not}(\texttt{or}(\texttt{not}(\phi), \texttt{not}(\psi)))$, $\texttt{implies}(\phi, \psi) \equiv \texttt{or}(\texttt{not}(\phi), \psi)$, $\texttt{G}(\phi) \equiv \texttt{not}(\texttt{F}(\texttt{not}(\phi)))$, $\texttt{R}(\phi, \psi) \equiv \texttt{not}(\texttt{U}(\texttt{not}(\phi), \texttt{not}(\psi)))$, $\texttt{H}(\phi) \equiv \texttt{not}(\texttt{O}(\texttt{not}(\phi)))$, $\texttt{forall}(x, \phi) \equiv \texttt{not}(\texttt{exists}(x, \texttt{not}(\phi)))$.

Let $\phi_1, \ldots, \phi_n$ be the set of formulas occurring in the LTL goals of an ASLan specification. The ASLan specification is *valid* if and only if $\pi \models^H_\sigma \texttt{G}(\texttt{not}(\mathit{attack}))$ and $\pi \models^H_\sigma \phi_i$ for all $i = 1, \ldots, n$, all interpretations $\sigma : \mathcal{V} \to T_\Sigma$, and all initialized paths π, where H is the set of Horn clauses defined in the specification.

2.3.1 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 backends 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 [11]. 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. $add(\alpha\ \texttt{set}, \alpha) : (\alpha\ \texttt{set})$. 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.

2.3.2 Micro and Macro Steps

For the translation from ASLan++ to ASLan, we consider a concept of micro-steps and macro-steps, 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 § 4.13.

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 $\rightarrow$ of ASLan as defined above. We define the new *macro-step transition relation* $\twoheadrightarrow$ based on the micro steps as follows: $S_1 \twoheadrightarrow S_n$ iff there exist $S_2, \ldots, S_{n-1}$ such that

- $S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_{n-1} \rightarrow S_n$,
- 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 $\rightarrow$ 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.

3 ASLan++ introduction 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

As an example of how an ASLan++ specification looks like, take the well-known Needham-Schroeder Public Key protocol [18] 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.

```
specification NSPK channel_model CCM

entity Environment {

  entity Session (A, B: agent) {

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

```
  symbols
    Na, Nb: message;

  body {
    Na := fresh();
    secrecy_goal secret_Na: Actor,B: Na;
    Actor  -> B: {Na.Actor}_pk(B);

    B  -> Actor: {Na.?Nb}_pk(Actor);
  channel_goal Alice_authenticates_Bob_on_Na:
    B *-> Actor: Na;
    secrecy_goal secret_Nb: B,Actor: Nb;

    Actor  -> B: {Nb}_pk(B);
  channel_goal Bob_authenticates_Alice_on_Nb:
    Actor *-> B: Nb;
  }
}

entity Bob (Actor: agent) {

  symbols
    A: agent;
    Na, Nb: message;

  body {
    ?A -> Actor: {?Na.?A}_pk(Actor);     % Bob learns A here!
   %secrecy_goal secret_Na: A,Actor: Na;

    Nb := fresh();
    secrecy_goal secret_Nb: A,Actor: Nb;
    Actor  -> A: {Na.Nb}_pk(A);
  channel_goal Alice_authenticates_Bob_on_Na:
    Actor *-> A: Na;

    A  -> Actor: {Nb}_pk(Actor);
  channel_goal Bob_authenticates_Alice_on_Nb:
    A *-> Actor: Nb;
  }
}
```

```
    body {
      new Alice(A,B);
      new Bob(B);
    }
  }

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

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 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 dec-

larations. There is a "virtual" module available called `Prelude`, defined in § 3.12, 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 § 3.11, start with a "`%`" symbol and extend until the end of the line.

3.3 Entities

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 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 § 3.9 for details.)

If an entity has a formal parameter `Actor` (which must be of type `agent` or a subtype of it), it is instantiated only if the actual value of the parameter `Actor` is the name of an honest agent. In this case, the agent faithfully plays the role defined by the entity. The value of `Actor`, i.e. the name of the agent playing this role, is important for defining the security properties of the entity.

The assumed attacker, who by definition is dishonest, is known as *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", i.e. he may have several names that he controls, in the sense that he has, for instance, the necessary long-term keys to actually work under a particular name. 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 all collaborate. This worst case of a collaboration of all dishonest agents may be simply modeled by one intruder who acts under different identities. To that end, we use the predicate `dishonest` that holds true of every dishonest agent `A` (from the initial state on). We can also allow for rules that model the compromise of an agent `A` by giving the intruder all knowledge of `A` and adding the fact `dishonest(A)`.

We will use this also for pseudonyms freshly created by the intruder for pseudonymous channels, which are discussed in more detail below; see also [16, 17].[1]

It is important to note that the predicate `dishonest` is not built-in, although it is used in the predefined channel goals (see below). It is in the hand of the modeler how to use it, and we just give a few recommendations how it should normally be handled. The most basic use is that only the intruder is dishonest. One may, however, declare any number of dishonest agents and at any time (to model, for instance, that an intruder may be able to attack some machine at some point, as remarked above). 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, if S is a secret between a set of agents and the intruder successfully compromises one of these agents, one would usually take a provision in the secrecy goal that the respective agents have not yet been compromised.

Also, it is very recommendable to ensure, for each entity that may also be "played" by a dishonest party, that 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 is subsumed by what the intruder can do. (Of course, the intruder does not necessarily stick to the prescribed steps of this entity as an honest agent would.) If this property is ensured, one may safely exclude that any entity is instantiated with a dishonest player, i.e. such `dishonest(Actor)` never holds for any entity at any time. This is safe, because all transitions that the dishonest Actor can perform are covered by the intruder's ability, and we therefore spare the tools the transitions associated with executing "honest" steps of the entity description for a dishonest agent.

[1] For instance, to ensure that the intruder can generate 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 model, where pseudonyms are simply public keys (as, e.g., in PBK). Creating a new pseudonym thus means generating a key pair $(\psi, \mathsf{inv}(\psi))$.

Finally, to conclude this subsection, note that entity execution is "compressed", which means that the statements in its body are executed atomically, except for certain "breakpoints". At these breakpoints, compressed execution is broken up, such that other entities and the intruder may interleave. By default, these breakpoints are immediately before the reception of messages. Optionally, an alternative set of breakpoints may be specified just before the the entity's body. This set of breakpoints is inherited to sub-entities unless they declare their own set of breakpoints.

3.4 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 a predicate holds or not is expressed by the existence the respective predicate term in the "fact space". Facts may be combined with the usual logical operators in LTL formulas and used in conditions (e.g. in an `if` statement).

By default a fact does not hold, but it may be explicitly introduced (simply by writing it as an ASLan++ statement) and retracted. For instance, the constant `true` is introduced automatically, while the constant `false` should never be introduced.

Facts may also be manipulated by so-called *clauses*[2]. These are rules for "spontaneously" producing facts whenever the conditions on their right-hand side are fulfilled. Both the head (on the LHS) and the body (on the RHS) may refer to local or inherited variables of the entity where the clause is defined, making it applicable only when the entity instances "owning" these variables are present. All other variables in the clause (i.e. free in the scope of the owner entity) are treated as universally quantified. Free variables *only* appearing in the LHS must be explicitly universally quantified using the keyword `forall`. Any other free variables must be listed as formal parameters right after the name of the clause, which is useful for documenting derivations that involve the clause, as demonstrated at the end of § 5.1. For instance, assume `Z` is a variable defined in an entity. Then the following Horn clause may be declared in the entity:

```
hc_name(X): forall Y. fact2(X,Y) :- fact1a(X) & fact1b(Z);
```

[2]We use this name as a shorthand for *definite Horn clauses*.

3.5 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) even if they are declared in a sub-entity of the root entity. 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. `agent < message`, and parameters. The only types that have (implicit) type parameters are tuples (e.g. `agent * message`) and sets (e.g. `message 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. 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).
```

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 are by default public, which means they are known and usable by the intruder. To avoid this default behavior, the keyword `private` must be given before the symbol name when declaring the function or constant. Similarly, user-defined functions are by default invertible, 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.

3.6 Terms

The set of terms includes variables (e.g. `A`), constants (e.g. `b2`), function applications (or to be more precise, function symbols applied to first-order terms, e.g. `hash(Msg)`), tuples (e.g. `(A,b2)` and set literals (e.g. `{A,B,C}`). We denote the concatenation of messages `M1` and `M2` as `M1.M2`. This operation associates to the right, i.e., `M1.M2.M3` = `M1.(M2.M3)`. 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. [3]

To enhance readability, any function application of the form `f(X,Y,Z,...)` may alternatively be written "in object-oriented style" as `X->f(Y,Z,...)`.

The precedence of operators in guards and formulas is, in descending order: '`(`' ... '`)`', '`->`', '`!`', '`=`', '`&`', '`|`', '`=>`'.

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*.

3.7 Channels

To express security properties of (potentially pseudonymous) channels, we recall and partially extend the notation we introduced in Deliverable D2.2 ("ASLan v.2 with static service and policy composition" [6]) and Deliverables D3.3 ("Attacker models" [4]). We use an intuitive notation from [14], where a secure end-point of a channel is marked with a bullet, with the following informal meaning:

[3] 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).

- `A *->  B: M` represents an *authentic channel*.
- `A  ->* B: M` represents a *confidential channel*.
- `A *->* B: M` represents a *secure channel*,
  i.e. a channel that is both authentic and confidential.

Note that in [4, 6], we used abstract bullet symbols while here we prefer to stress the concrete syntax and write, e.g., `A *-> B: M` instead of $A \bullet\!\!\rightarrow B : M$.

We use a similar notation to represent another kind of channel:

- `A => B: M` represents a *resilient channel*.

This notation may be combined with the previous ones, e.g. `A *=>* B: M` represents a channel that is both secure and resilient. As stated in [2]: "A communication channel is *resilient* if it 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." With reference to the formalism presented in § 4.2 of D3.3, as specified in [1], this amounts to requiring that every message sent over the channel will be eventually delivered to the intended recipient.

There is also a further variant of the channel notation using a double-headed arrow like "`->>`" representing a *fresh channel*. *Freshness* of channels means replay protection: each sent message can be received only once. This notation may be used for channels that include at least authenticity, e.g. `A *->>* B: M` represents a channel that is both secure and fresh.

While [14] uses the bullet notation to reason about the existence of channels, we use it to specify message transmission in services in two ways.

- First, we 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 § 3.8.3.

3.7.1 Channel syntax

In general, i.e. for both channels as assumptions and channels as goals, we also allow agents to be alternatively identified by pseudonyms rather than by their real names. We denote this as `[a]_[p]` where `a` is the real name and `p` is the pseudonym. By default, every agent that acts pseudonymously will create a fresh pseudonym upon instantiation; we write simply `[a]` if `a` acts under this default pseudonym. Note that from the point of view of a

particular agent, the real name of the peer is not visible when communicating over a pseudonymous channel, i.e. we cannot use the `[a]` notation for other agents than the agent playing the current rule (denoted by `Actor`), but will instead use only their pseudonyms (in place of their real agent names).

Table 1 shows the notations that ASLan++ supports: facts in "OO-style" notation as well as the annotated channel notation. Here, `ActP` stands for `Actor` or any pseudonym of it, i.e. `[Actor]` or `[Actor]_[P]` for any `P`.

For sends and receives, if the first actual parameter is `Actor`, like in `send(Actor,B,M)` or `Actor->receive(A,M)`, it may be omitted such that only `send(B,M)` or `receive(A,M)`, respectively, is written.

3.7.2 Channel models

As described in detail in [4], we have formalized three different models of communication and definitions of channels as assumptions:

- The *Cryptographic Channel Model CCM*: here, channels are realized by certain ways of encrypting/signing messages and transmitting them.
- 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*: this model is similar to the ICM, but using explicit send and receive events that are constrained by an LTL formula over the traces.

For each ASLan++ specification, the channel model is globally specified, right at the beginning after the specification name.

We have shown in [4,16] 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 [4, 6], while here we just summarize the main differences.

CCM and ICM The CCM and the ICM focus on individual message transfers and consider the following notions of confidentiality and authenticity:

"OO-style" notation	Annotated channels notation
`ActP->send(B,M)`	`ActP -> B: M`
`ActP->receive(A,M)`	`A -> ActP: M`
`ActP->send(B,M) over authCh`	`ActP *-> B: M`
`ActP->receive(A,M) over authCh`	`A *-> ActP: M`
`ActP->send(B,M) over confCh`	`ActP ->* B: M`
`ActP->receive(A,M) over confCh`	`A ->* ActP: M`
`ActP->send(B,M) over secCh`	`ActP *->* B: M`
`ActP->receive(A,M) over secCh`	`A *->* ActP: M`
`ActP->send(B,M) over fresh_authCh`	`ActP *->> B: M`
`ActP->receive(A,M) over fresh_authCh`	`A *->> ActP: M`
`ActP->send(B,M) over fresh_secCh`	`ActP *->>* B: M`
`ActP->receive(A,M) over fresh_secCh`	`A *->>* ActP: M`
`ActP->send(B,M) over linkCh`	`ActP -L-> B: M`
`ActP->receive(A,M) over linkCh`	`A -L-> ActP: M`
`ActP->send(B,M) over link_authCh`	`ActP *-L-> B: M`
`ActP->receive(A,M) over link_authCh`	`A *-L-> ActP: M`
`ActP->send(B,M) over link_confCh`	`ActP -L->* B: M`
`ActP->receive(A,M) over link_confCh`	`A -L->* ActP: M`
`ActP->send(B,M) over link_secCh`	`ActP *-L->* B: M`
`ActP->receive(A,M) over link_secCh`	`A *-L->* ActP: M`
`ActP->send(B,M) over resCh`	`ActP => B: M`
`ActP->receive(A,M) over resCh`	`A => ActP: M`
`ActP->send(B,M) over res_authCh`	`ActP *=> B: M`
`ActP->receive(A,M) over res_authCh`	`A *=> ActP: M`
`ActP->send(B,M) over res_confCh`	`ActP =>* B: M`
`ActP->receive(A,M) over res_confCh`	`A =>* ActP: M`
`ActP->send(B,M) over res_secCh`	`ActP *=>* B: M`
`ActP->receive(A,M) over res_secCh`	`A *=>* ActP: M`
`ActP->send(B,M) over res_linkCh`	`ActP =L=> B: M`
`ActP->receive(A,M) over res_linkCh`	`A =L=> ActP: M`
`ActP->send(B,M) over res_link_authCh`	`ActP *=L=> B: M`
`ActP->receive(A,M) over res_link_authCh`	`A *=L=> ActP: M`
`ActP->send(B,M) over res_link_confCh`	`ActP =L=>* B: M`
`ActP->receive(A,M) over res_link_confCh`	`A =L=>* ActP: M`
`ActP->send(B,M) over res_link_secCh`	`ActP *=L=>* B: M`
`ActP->receive(A,M) over res_link_secCh`	`A *=L=>* ActP: M`

Table 1: Channel notations in ASLan++

- *Confidential* in the CCM/ICM means that A can be sure that only B can read the content of the message M. An intruder cannot learn any message contents on these channels, but he can store and replay the

messages later.

- *Authentic* in the CCM/ICM means that B can be sure that A has sent the message M. Moreover, A intended the message for B — this additional property, known as *directedness*, has been motivated in [4, §4.1.4].

Further, for CCM and ICM channels that are authentic or secure, one may use the channel attribute *freshness*, indicated by a double-headed arrow like `->>`, meaning replay protection.

All these channel kinds 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 [4, §4.1.4]).

ACM In the ACM, the notion of a channel is per "session" between two parties. The ACM considers the following notions of confidentiality and authenticity:

- In the ACM, a channel provides *confidentiality* if its output is exclusively accessible to a given receiver. According to this definition an intruder cannot learn anything from the messages in transit on these channels, and — differently from the ICM and CCM — he cannot even store and replay the messages later.

  If the recipient is identified by a pseudonym, the confidential channel provides *weak confidentiality* as defined in [4, §4.2.2], i.e. its output is exclusively accessible to a single, yet unknown, receiver.

- In the ACM, a channel provides *authenticity* if its input is exclusively accessible to a specified sender. According to this definition, the authentic channel provides sender authentication, without providing directedness.

  If the sender is identified by a pseudonym, the authentic channel provides *weak authenticity*, as defined in [4, §4.2.2], i.e. its input is exclusively accessible to a single, yet unknown, sender. This is the notion of *sender invariance* described in [4, §4.1.4] as well.

For authentic or confidential channels (and thus also for secure channels), the freshness property is implied by the ACM.

As described in [4, §4.2.2], in case we want to model a run of SSL/TLS in which principal B has a valid certificate but principal A does not, it is necessary to consider a pair of channels $A2B$ and $B2A$, with the additional

requirement that the principal sending messages on $A2B$ is the same principal that receives messages from $B2A$. These channels are indicated by the arrow "`-L->`" rather than "`->`" and "`=L=>`" rather than "`=>`".

Extensions It is important to note that we have designed ASLan++ so to allow the modeler to choose the particular channel models 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]), channel identifiers, 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.

3.8 Goals

Validation goals have a name and (in one way or the other) give rise to an LTL formula that is checked by the validation backends 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 (usually at the current state of execution), the usual first-order logic operators (namely, propositional connectives and quantifiers), and (future and past) temporal operators. The LTL operators for ASLan++ are listed in Table 2. See § 2.3 and § 4.11 for formal definitions of the operators.

In ASLan++ there are different kinds of goals, which we describe below.

3.8.1 Invariants

Goals that should hold during the whole life of an entity instance should be stated in the `goals` section of the entity declaration. In particular, goals expected to hold globally during the overall system execution should be given in the outermost entity. For this type of goals, any LTL formula can be given. Free variables, i.e. those not declared in the given entity nor inherited from any outer entity, are implicitly universally quantified.

Operator	ASLan++ connective	Explanation
$\neg$	`!f`	negation
$=$	`f_1 = f_2`	equality
$\wedge$	`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
Yesterday	`Y(f)`	in the previous state
Finally	`<>(f)`	eventually
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 2: LTL operators for specifying goals

3.8.2 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.

3.8.3 Channel goals

A channel goal has the form

```
channel_goal Name: Sender Channel Receiver: Payload
```

similar to the syntax of message transmission: `Sender` and `Receiver` can be real names or the pseudonym notation, and `Channel` is a symbol for the kind of channel used. For instance, the goal of using TLS without client authentication to send a suitable payload message is `[Client] *->* Server: Payload`. A channel goal pertains to message transmission immediately preceding the goal.

Symbol	Explanation
`->*`	confidentiality (not including freshness)
`->>*`	confidentiality plus freshness
`*->`	authenticity and directedness
`*->>`	authenticity and directedness plus freshness
`*->*`	authenticity, directedness, and confidentiality
`*->>*`	the above plus freshness

Table 3: Channel goal symbols in ASLan++

When goals are specified using the channel notation, they refer to individual message transfer even when the ACM is chosen for channel assumptions. In other words, the arrow symbols are interpreted as for channels as assumptions in the ICM/CCM. This implies the intuitive meaning for each type of arrow as given in Table 3. More details can be found in [4, §4.1.4] and [16,17].

As an example consider the following specification excerpt:

```
entity Alice (Actor, B: agent) {
  symbols
    Payload: message;
  ...
  body {
    ...
    Actor -> B: ...Payload...  % send Payload to B,
                               % somehow fully secured
  channel_goal secure_Alice_Payload_Bob:
    Actor *->* B: Payload;
    ...
  }
}

entity Bob (A, Actor: agent) {
  symbols
    Payload: message;
  ...
  body {
    ...
    A -> Actor: ...?Payload...  % receive Payload from A,
                                % somehow fully secured
  channel_goal secure_Alice_Payload_Bob:
    A *->* Actor: Payload;
```

```
    ...
  }
}
```

A channel goal is given as follows:

- The goal is stated — using the same goal name — both in the entity representing the sender role and the entity representing the receiver role. In the above example, the sender is `Alice` and the receiver is `Bob`.
- On the sender side, the goal is stated just after the respective send statement, with the agent name before the arrow being `Actor` (or a pseudonym for it).
- On the receiver side, the goal is stated just after the respective receive operation, with the agent name after the arrow being `Actor` (or a pseudonym for it).
- 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 3 and § 3.7.
- The payload part of the goal, on both the sender and receiver sides, should refer to the protected value sent or received, respectively. The ASLan++ terms denoting the payload in the two entities involved may look different (e.g. due to different variable names), but their actual values should of course be equal.

3.8.4 Secrecy goals

A secrecy goal has the form

```
secrecy_goal Name: Agents: Term
```

with the interpretation that the value of the given term must be known only to the agents given. This goal should be stated — using the same goal name — in all entities that may know the secret value, as soon as the value is known to the respective entity instance, for example:

```
entity Alice (Actor, B: agent) {
  symbols
    Nonce: message;
  ...
  body {
```

```
    ...
    Nonce := fresh();
    secrecy_goal secret_Nonce: Actor,B: Nonce;
    Actor -> B: ...Nonce...; % send Nonce to B,
                             % somehow confidentiality protected
    ...
  }
}

entity Bob (A, Actor: agent) {
  symbols
    N_A: message;
  ...
  body {
    ...
    A -> Actor: ...?N_A...;  % receive Nonce from A,
                             % somehow confidentiality protected
    secrecy_goal secret_Nonce: A,Actor: N_A;
    ...
  }
}
```

The terms given for the same secrecy goal in the 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.

Note that secret transmission of a value between two parties can also be stated as a channel goal, but the secrecy goal is more general: it may be used to state the the value is shared between more than two parties.

It is even possible to dynamically change the set of agents that may know the value. For example, if we have

```
secrecy_goal myNa: A,B: Na;
```

we can dynamically add `C` to the set by introducing the fact

```
secrecy_myNa_set(IID)->contains(C);
```

or dynamically remove B from the set by stating

```
retract secrecy_myNa_set(IID)->contains(B);
```

where the `myNa` part is the name of the secrecy goal introduced before.

3.9 Statements

Statements may be the usual assignments, branches and loops, but also nondeterministic selections, assertions, channel goals, secrecy goals, the generation of fresh values and of new entity instances, the transmission of messages (i.e. send and receive operations), and the generation 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.

Assertions induce goals to be checked at the current position in the control flow.

Entity generation may instantiate only direct sub-entities, such that static and dynamic scoping coincide.

Symbolic entity generations, introduced by the keyword `any`, are a convenient shorthand for loosely instantiating an entity (which is typically a session), in the following sense: the agents executing the instance, as indicated by the given list of variables, are arbitrarily selected from the domain of agents. An optional guard, which may refer to the variables listed, may constrain the selection.

3.10 Guards

In guards (i.e. conditions in `if`, `while`, `select`, and symbolic entity generation statements) variable names may be preceded by "`?`" to specify that during pattern matching each "`?`"-variable obtains a suitable value such that the overall condition is satisfied. For each variable name occurring in a guard, either all occurrences must be preceded by "`?`" or appear without a "`?`".

At pattern positions where the value to be matched is irrelevant, one may write a singleton "`?`", without giving a variable name.

When a guard is evaluated, variables are handled as follows.

- A variable name without "`?`" is evaluated according to the current value of the variable.
- A "`?`" not followed by a variable name matches any value.
- For a variable name with "`?`", it is checked if there is a value that can be consistently replaced for all occurrences of this variable name such that the condition holds.

If the guard cannot be satisfied according to these rules, no variable assignment occurs. Otherwise, each "`?`"-variable is assigned to a value such that the condition holds. If there is more than one solution then each of them is a possible successor state. Note that an ASLan++ specification may give rise to many different possible behaviors of the described system and the verification covers all of these behaviors.

To avoid interference with negation and implicit quantification, there are further syntactic restrictions on "`?`"-variables (but not on singleton "`?`"s). They apply after constructing the disjunctive normal form (DNF) of the guard (which among others expands implications of the form "`P => Q`" as "`!P | Q`").

- "`?`"-variables must not occur as arguments of negated facts.
- In the condition of an `if` or `while` statement, the sets of "`?`"-variables occurring as arguments of facts must be pairwise disjoint, i.e. no "`?`"-variable may appear in more than one conjunct of the DNF. This rules out the occurrence of variables like `?X` in multiple facts, e.g. `if (fact1(?X) & fact2(?X,Y))` because their negation in the "else" branch would require a quantifier, namely

  ```
  !(exists 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);
  ```

3.11 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.

```
Main        ::= "specification" Name
                "channel_model" ("CCM" | "ICM" | "ACM") EntityDecl
Module      ::= EntityDecl
EntityDecl  ::= "entity" UpperName Params? "{" Imports?
                 Decls* EntityDecl* Body? GoalDecls? "}"
```

```
UpperLetter ::= ["A" .. "Z"]
LowerLetter ::= ["a" .. "z"]
Digit       ::= ["0" .. "9"]
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 "->")? LowerName ("(" Types ")")? % general compound
              | Type ("." Type)+                        % concatenation compound
              | "(" Type ("," Type)+ ")"                % tuple compound
              | "(" Type ")"
Types       ::= (Type #",")+

SymbolDecls ::= "symbols" (SymbolDecl ";")+
SymbolDecl  ::= Vars ":" Type
              | "private"? (LowerName #",")+ ":" Type        % constants
              | "private"? "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 (":-" (Term #"&")+)?

EquationDecls ::= "equations" (EquationDecl ";")+  %% only allowed in the root entity
EquationDecl  ::= Term "=" Term

Term        ::= LowerName
              | Var
              | "?"Var  %% only allowed within guards
              | "?"     %% only allowed within guards
              | "[" 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 ")"
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
              | Guard ("&" | "|" | "=>") Guard
              | "(" Guard ")"
GuardNoRcv  ::= Guard %% any Guard but no transmissions allowed here

Stmt        ::= Assign
              | FreshGen
              | EntityGen
              | SymbEntityGen   % symbolic session
              | Transmission ";" ChannelGoal*
              | IntroduceFact
              | RetractFact
              | Branch
              | Loop
              | Select
              | Assert
              | SecrecyGoal
              | "{" Stmt* "}"

Assign      ::= Var ":=" Term ";"
FreshGen    ::= Var ":=" "fresh()" ";"
EntityGen   ::= "new" Entity ";"
SymbEntityGen ::= "any" (Var+ ".")? Entity ("where" Guard)? ";"
Transmission  ::= (Term "->")? ("send" | "receive") "(" Terms ")" ("over" Term)?
          % where "Actor ->" is optional
              | Term ChannelArrow Term ":" Term ";"  % annotated channels
          %% set literals are not allowed in transmissions
ChannelArrow  ::= "*"? ("->" | "->>" | "-L->" |
                        "=>" | "=>>" | "=L=>" ) "*"?

Entity      ::= UpperName ("(" Terms ")")?
IntroduceFact ::= FuncApp ";"
```

```
RetractFact ::= "retract" FuncApp ";"
Branch      ::= "if" "(" GuardNoRcv ")" Stmt ("else" Stmt)? ";"
Loop        ::= "while" "(" GuardNoRcv ")" Stmt
Select      ::= "select" "{" ("on" "(" Guard ")" ":" ChannelGoal* Stmt)+ "}"
                                    % Guard may include receive
Assert      ::= "assert" GoalDecl ";"
ChannelGoal ::= "channel_goal" Name ":" Term ChannelArrow Term ":" Term ";"
SecrecyGoal ::= "secrecy_goal" Name ":" (Term #",")+ ":" Term

GoalDecls   ::= "goals" (GoalDecl ";")+
GoalDecl    ::= Name ":" Formula
Formula     ::= FuncApp
              | "!" Formula
              | LTLOp1 "(" Formula ")"
              | LTLOp2 "(" Formula "," Formula ")"
              | 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"
```

3.12 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(message): fact` to specify that the intruder knows a particular value (of type message or a subtype of it), 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;
    nat < message;
  symbols
    i: agent;
    dishonest(agent): fact;
    iknows(message): fact;
    true, false: fact;
    0,1,2,3, ...: nat;
    succ(nat)   : nat;
    "."  (message, message): message; % non-associative concatenation

  % subtypes of message and related symbols
  types
    agent         < message;
    symmetric_key < message;
    public_key    < message;
    private_key   < message;
  symbols
    noninvertible hash   (message): message;
    noninvertible scrypt (symmetric_key, message): message;
    noninvertible crypt  (public_key, message): message;
    noninvertible sign   (private_key, message): message;
```

```
    private       inv(public_key): private_key;
    noninvertible pk(agent): public_key;
    noninvertible defaultPseudonym(agent,nat): agent;
  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"
  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(K,M));
} % 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 term `pk(A)` is used to denote the canonical public key of an agent `A`. 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.

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 tool.

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), Horn clauses (§ 4.4), equations (§ 4.5) and security goals (§ 4.11).

Some preliminaries for the translation of code sections are described in § 4.6 (for control flow) and in § 4.13 (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 § 4.12).

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. § 4.14 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 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 `agent`) and will represent the agent who is playing the role of the entity to which the state fact is associated.
- The second parameter will be of type `nat` and will represent the instance ID.
- The third parameter will be of type `nat` and will represent the step label.

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.

Assuming the restriction that entities can instantiate only direct sub-entities, we implement this binding as follows:

- We utilize a binary predicate: *descendant(A,D)* stating that the entity instance D has been instantiated directly or indirectly by A, i.e. is a sub-node of A in the tree of dynamically created entity instances.
- At instantiation of an entity, let *iid* and *pid* be the IDs of the new instance and its parent, respectively. We add to the state the (persistent) fact *descendant(pid,iid)*.

- *descendant* predicate is completed by taking it transitive closure, which may be implemented via the Horn clause

$$descendant(A, D) :- descendant(A, E) \ \& \ descendant(E, D)$$

- When we add the state fact of the owner of a variable to the state, we actually add both its state fact (with ID *OID*) and the predicate *descendant(OID,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 § 3.12. 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.

Note also that for each declaration of a constant `c` of type `agent` we need to add to the initial state the fact `isAgent(c)`, denoting that `c` belongs to the domain of agents (needed for *symbolic entity instantiation*, see § 4.7.5).

Symbols can be declared using the modifiers `private` and `noninvertible`.

- If a symbol (which may be a constant) $f(\tau_1, \ldots, \tau_n) : \tau$ is not declared as private, it is considered public. Its semantics can be defined by the following Horn 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. Its semantics can be defined by the following rules:

```
hc f_invertible_i(M1,..,Mn):
  iknows(Mi) :-  iknows(f(M1,...,Mn))
```

for each $i \in \{1, \ldots, n\}$ (again M_i of type τ_i). 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 Horn Clauses

Although each Horn clause is declared inside an entity, it should be applicable globally if it does not refer to local variables (including parameters) of the entity or any enclosing ones. Otherwise, the Horn clause will be applicable locally, being inherently bound to the variables it refers to, which might belong to the entity as well as to its ancestors. Translating the Horn clauses is therefore adjusted so to preserve this binding.

For each of the entities whose variables are referred to by the given Horn clause, we add to the conditions of the clause their state fact. This means that if in an entity `E` we have a Horn clause

```
h: A :- A_1 & .. & A_n;
```

such that if `V` is the intersection between the variables in the scope of `E` and the variables in `A,A_1,..,A_n`, and `{o_1,..,o_m}` is the set of owners of the variables in `V`, we add to the translation the clause

```
h: A :- A_1 & .. & A_n & state_{o_1}(..) & .. & state_{o_m}(..)
```

so that each variable in the Horn clause that exists in the scope of E is bound to the value the variable currently has in its owning entity instance.

Example 2 Take the entity declaration for the `BidM` above, extended with a Horn 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);
}
```

The only *free* variable in the clause is `X`, while `Actor` and `BP` are bound to the values of the parameters.

If this clause were transcribed to an ASLan specification as is,

```
trusts(Actor, BP, X) :- saidTo(BP, Actor, X);
```

then `Actor` and `BP` would be considered as universally quantified and the clause would change its meaning entirely (anyone would trust any entity who says something). Adding instead the state fact to the conditions of the clause

```
trusts(Actor, BP, X) :- saidTo(BP, Actor, X) &
                        state_BidM(Actor, IID, SL, BP, M);
```

`Actor` and `BP` are bound by the state fact

```
state_BidM(Actor, IID, SL, BP, M)
```

to belong to an instance of the entity. □

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 § 4.12 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`.

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 an ancestor of `BidM`, e.g. `Env`, the resulting rewrite rule would be

```
state_BidM(Actor,IID,sl,BP,M).
  state_Env(..,OID,..,E,..).
  descendant(OID,IID)
  =>
state_BidM(Actor,IID,succ(sl),BP,M).
  state_Env(..,OID,..,crypt(pk(BP),m),..).
  descendant(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)
}
```

Recall that, as we remarked in Subsection 3.3 above, it is very recommendable to ensure, for each entity that may also be "played" by a dishonest party, that with the sufficient knowledge of long-term 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. (Of course, the intruder does not necessarily stick to the prescribed steps of this entity as an honest agent would.) If this property is ensured, one may safely exclude that any entity is instantiated with a dishonest player, i.e. such `dishonest(Actor)` never holds for any entity at any time.

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).
  descendant(OID,IID)
```

that will create the fact for the `BidM` instance, with step label `sl_0`. □

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

  create, if not present, the following rules
      true =[exists A]=> true.isAgent(A)
      true =[exists A]=> true.isAgent(A).dishonest(A)
  in "section rules" in the translation
```

```
  g_1..g_k = positiveGuards(Guard)

  apply adaptGuard to g_1..g_k

  IID = fresh variable

  declare IID of type nat 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.isAgent(V_j)
      RHS = RHS.isAgent(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 the partially specified instantiation of an entity. Its purpose is modelling a whole class of instances, i.e. for all possible substitutions of non-ground agent names `A_1,..,A_m` with

ground agent constants taken from the domain.

This is done by first "generating" the domain of the agents, whose membership is represented by the predicate `isAgent()`. The domain includes the user-defined agent constants (for which the corresponding `isAgent()` predicates are added to the initial state in § 4.3), and is augmented to an unbounded set of agents by the rules

```
true =[exists A]=> true.isAgent(A)
true =[exists A]=> true.isAgent(A).dishonest(A)
```

that generate fresh constants for which the predicate `isAgent` holds. The predicate `dishonest` is needed later on for verification purposes. The constant fact `true` is used because the LHS of a transition rule cannot be empty.

Then, the actual symbolic instantiation is translated in a rule generating a new instance of the entity. The instantiation will contain the non-ground agents `A_1,..,A_m`, but these will be bound to concrete agents by the predicates `isAgent(A_1)..isAgent(A_m)` in the *LHS* of the rule (and also renewed in the *RHS*). In addition to this, the rule will also enforce that the chosen agents for `A_1,..,A_m` satisfy `Guard`, whose treatment we will introduce in detail in § 4.7.9.

4.7.6 Message Transmission

```
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 it 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, 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 certain ways of encrypting messages and transmitting them over secure channels.

- 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*: similar to the ICM, but using explicit send and receive events that are constrained by an LTL formula over the traces.

We have shown in [4,16] 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 namely 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 that 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 the modeling of channels like resilient channels (i.e. every message is eventually received) expressed as LTL constraints on the sending and receiving.

Translation of channel goals is discussed in § 4.7.13. Here, we focus on the translation of channels when used as an assumption, which depends on the choice of the communication model as we discussed in detail in D3.3. 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 communication model chosen, let us illustrate the

translation(s) at hand of a concrete example (we proceed analogously for the other cases).

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 secure channel from `A` and sending a message `M2` under pseudonym `P` to `B` also on a secure channel:

```
   A     *->* Actor: M1
[Actor]_P *->*   B : M2

L =[V]=> R
```

The final translation into ASLan in the three models is as follows:

- CCM:

  ```
  iknows(crypt(ck(Actor),sign(inv(ak(A)),stag.Actor.M1))).L
  =[V]=>
  R.network(crypt(ck(B),crypt(inv(P),stag.B.M2)))
  ```

  Here, `ak` and `ck` are the authentication key and confidentiality key solely used for defining channels.

  ```
  noninvertible ak(agent): public_key; % authentication  key
  noninvertible ck(agent): public_key; % confidentiality key
  ```

- ICM:

  ```
  secCh(A,Actor,M1).L
  =[V]=>
  R.secCh(P,B,M2)
  ```

- ACM:

  ```
  rcvd(Actor,A,M1,ch(A,Actor,secure)).L
  =[V]=>
  R.sent(Actor,P,B,M2,ch(P,B,secure))
  ```

  Here, `ch(...)` is a function to create a channel identified from the given identities and the type of channel.

Note that each model may introduce a number of symbols, facts, and rules necessary to express the different channels as defined in D3.3; for instance, the CCM introduces a function `ck`, the ICM a fact `secCh`, and ACM facts `sent` and `rcvd`, amongst many others.

4.7.7 Fact introduction

```
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)`.

An element removal statement of the form `remove(Set,X)` is handled as a retraction `contains(Set,X)`; see below.

4.7.8 Fact retraction

```
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)
}
```

Note that if a fact to be retracted is not present, execution is blocked (until the fact is introduced).

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 facts in the clause.

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 `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 § 4.11.

4.7.13 Channel Goals

As we described above, we can use the bullet annotations to specify goals of a service using the different kinds of channels. Intuitively, this means that the service should ensure the authentic, confidential, or secure transmission of the respective message. These definitions are close to standard authentication and secrecy goals of security protocols, e.g. [9, 12, 15]. 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.

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 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 IF and Casper [13]). More specifically, we consider the following kinds of fact symbols:[4]

```
witness(agent,agent,protocol_id,message): fact
request(agent,agent,protocol_id,message,nat): fact
secret(message,protocol_id,set(agent)): fact
```

where the `protocol_id` type is used to hold an identifier for a particular goal (such as an ID of the service type).

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 § 3.7) 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 goals can be used only immediately after a transmission statement, or immediately after a guard when the guard contains a receive 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 4 summarizes the events that are used for each kind of channel, together with the rule for deriving the value for `protocol_id`.

For ease of description, consider the example given in § 3.8.3. Given a channel goal where the `Actor` appears on the left of the arrow:

```
  Actor -> B: ...Payload...  % send Payload to B,
                             % somehow fully secured
channel_goal secure_Alice_Payload_Bob:
  Actor *->* B: Payload;
```

[4]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 [16].

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 4: Events used in the translation of channel goals and rules used for deriving the protocol ID from the name of channel goals

the translator will use the channel goal for inserting suitable events right after the transmission statement:

```
Actor -> B: ...Payload...;
witness(Actor,B,auth_secure_Alice_Payload_Bob,Payload);
secret(Payload,secr_secure_Alice_Payload_Bob,{Actor,B});
```

When translating the transmission statement into ASLan, the inserted auxiliary events are put into the same transition rule with the transmission event. The `secret` event is further translated as described in § 4.7.14.

For channel goals where the `Actor` appears on the right of the arrow:

```
  A -> Actor: ...?Payload...  % receive Payload from A,
                              % somehow fully secured
channel_goal secure_Alice_Payload_Bob:
  A *->* Actor: Payload;
```

the translator will insert the needed events right after the transmission event:

```
A -> Actor: ...?Payload...;
request(Actor,A,auth_secure_Alice_Payload_Bob,Payload,IID);
secret(Payload,secr_secure_Alice_Payload_Bob,{A,Actor});
```

Again the inserted auxiliary events are translated into the same transition rule as the transmission event itself. And, like in the previous case, the `secret` event is further translated as described in § 4.7.14.

When using a pseudonym `P`, the sender or receiver name will simply be replaced by that in the goal, e.g. `witness(P,B,auth_Alice_Payload_Bob,Payload)`.

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 semantics of the authentication goal is

```
forall A,B,P,M,IID. [] (request(B,A,P,M,IID) =>
  (<-> (witness(A,B,P,M))
   || (dishonest(A) & iknows(M))))
```

This is a standard authentication goal (non-injective agreement [12]), which includes directedness, with the additional condition that, if the sender is dishonest, then the intruder must know this message. This requirement is necessary for the channel compositionality results of [16] (for which, otherwise, the channels would be too weak to compose).

The semantics 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 no agent `B` executing an entity instance `IID` should accept the same value `M` in a different session (identified by `IID'`) from the same honest communication partner `A` on the same channel identified by `A`, `B`, and `P`: that is, he has never previously requested the same value on the same channel.

4.7.14 Secrecy Goals

As mentioned in § 3.8.4, secrecy goals are a way of expressing the fact that the value of a given term must be known only to a certain set of agents.

Secrecy goals are translated using the *secret* predicate available in ASLan, with the signature `secret(message, protocol_id, set(agent))`. The protocol ID is derived from the name of the secrecy goal, while the message and the set of agents are taken as they are specified in the secrecy goal.

Each secrecy goal will generate on the ASLan level a transition rule of its own.

For example consider the following secrecy goal

```
secrecy_goal secret_token : alice, bob, cooper : token;
```

This means that the three agents `alice`, `bob` and `cooper` should be the only ones who know about the term `token`.

The generated transition rule will be:

```
state_...(Actor, IID, ...)
=>
```

```
contains(alice , secret_token_set(IID)).
contains(bob   , secret_token_set(IID)).
contains(cooper, secret_token_set(IID)).
secret(token, secret_token, secret_token_set(IID)).
state_...(Actor, IID, ...)
```

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 entity 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:

```
secret_token_set : nat -> set(agent)
```

The membership of each agent to the set is stated using the `contains` predicate.

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 should be part of the set `As` of agents.

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 5. 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 5. 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)`

Table 5: 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 = [T_1]`, i.e. `T` is the default *pseudonym* for a term `T_1` (which must be of type `agent`). Then `T` will be replaced by the function

  `defaultPseudonym(T_1',IID)`

  where `T_1'` is the recursive translation of `T_1` and `IID` is the instance ID associated to the `state` fact of `T_1`.

- `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`.

- `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.

- `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)`.

  Remark that set literals can only be used in assignments; in particular, they cannot be used as arguments of `send` or `receive`.

- `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).

Note that, in the case of the root entity, it will also be necessary to add its state fact with step label `sl_0` in `section inits` in the translation, in order for the execution to start.

Also 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. § 4.12 describes how this is done.

4.11 Translation of the Goals Section

In the case of goals, in a similar way as for guards, we need to translate the LTL operators to ASLan LTL syntax (given in § 2.3), according to Table 6. 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).

4.12 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 replace 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.

Operator	ASLan++ connective	ASLan predicate
$\neg$	`!f`	`not(f)`
$=$	`f_1 = f_2`	`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 6: Translation of goals

4.13 Step Compression

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).network(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.

The desired granularity for the model is induced by the choice of some *breakpoint* actions, optionally specified for each entity with the declaration `breakpoints {bp_1,...,bp_n}`. By default, the only breakpoint is the action `receive`. This set of breakpoints is inherited to sub-entities unless they declare their own set of breakpoints.

If needed, a modeler can specify additional 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.

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 § 2.3.2): 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 is now 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).network(N)
```

According to the ASLan semantics, 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 this simple case of a sequence of atomic actions, the following *compressed rule* would be equivalent:

```
state_responder(Actor, IID, 1, dummy_agent, dummy_message).iknows(A)
  =[exists N]=>
state_responder(Actor, IID, 4, A, N).network(N)
```

Whenever feasible, the translator will produce such compressed rules (as 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 regard it as a potential optimization step (that must be semantics-preserving, as in the above example) of the translator from ASLan++ to ASLan. See also § 4.14.

4.14 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.

4.14.1 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).network(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).network(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

$$PF.NF.state_{SL}\ C\ =[V]\Rightarrow\ R.state_{SL'}$$

See § 2.2 for syntax of 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

$$state_{SL} \Rightarrow state_{SL'}$$

Empty transitions are introduced by the translation procedure whenever a branch is exited.

An empty transition $state_{SL'} \Rightarrow state_{SL''}$ can be eliminated when it is the only transition that takes the entity from $state_{SL'}$ to $state_{SL''}$ and one of the following holds:

- there exists one or more other transitions that take the entity into state $state_{SL'}$, so that we can write

$$\begin{aligned} PF.NF.state_{SL}\ C\ &=[V]\Rightarrow\ R.state_{SL'} \\ state_{SL'}\ &\Rightarrow\ state_{SL''} \end{aligned}$$

- there exists one or more other transitions that take the entity out of state $state_{SL''}$, so that we can write

$$\begin{aligned} state_{SL'}\ &\Rightarrow\ state_{SL''} \\ PF.NF.state_{SL''}\ C\ &=[V]\Rightarrow\ R.state_{SL'''} \end{aligned}$$

In this case the existing chains of two transitions of the above form can be replaced by single transitions of the form

$$PF.NF.state_{SL}\ C\ =[V]\Rightarrow\ R.state_{SL''}$$

or

$$PF.NF.state_{SL'}\ C\ =[V]\Rightarrow\ R.state_{SL'''}$$

Redundant guards. A transition rule is considered to be a redundant guard if $PF = \emptyset$, $NF = \emptyset$, $C \subset \{true, false\}$, $V = \emptyset$ 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

$$state_{SL}.\texttt{true} \Rightarrow state_{SL'}.\texttt{true}$$

or

$$state_{SL}.\texttt{not}(\texttt{true}) \Rightarrow state_{SL'}$$

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 $state_{SL}.\texttt{not}(\texttt{true}) \Rightarrow state_{SL'}$ can be eliminated if there exists no other transition taking the entity out of $state_{SL}$. 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 $state_{SL}.\texttt{true} \Rightarrow state_{SL'}.\texttt{true}$ are equivalent to empty transitions and can be eliminated under the same conditions as empty transitions.

4.14.2 Merging of transitions

Following the notion of step compression explained in § 4.13, a second level of optimization is introduced. Instead of generating the transition rules with the *state*! predicate and leaving the handling of atomicity to the verifier tools, these transition rules can be 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*! predicate.

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)
=>
state_Alice(E_A_Actor, E_A_IID, 3, B, Req, Resp).network(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).network(Resp).some_fact(Req, Resp)
```

5 ASLan++ example

In this section, we assess ASLan++ against a problem case taken from Deliverable D5.1 [5] and elaborated in D5.2 [7], namely the Car Registration Scenario.

5.1 Car Registration

The car registration scenario has been described in Deliverable D5.1 (Problem Cases and their Trust and Security Requirements [5]) and formalized in a simplified version in Deliverable D2.1 [3, §4.3] and Deliverable D2.2 [6, §5.1]. Here, we present this scenario in the same simplified version but expressed in the ASLan++, to demonstrate the latest changes in the language.

When comparing the model given in [3, §4.3] specified in ASLan with the one given below specified in ASLan++, one will notice an enormous gain in readability and conciseness, in particular when comparing the unstructured set of transition rules with the structured blocks of (object-oriented programming language style) statements. Such a comparison also shows that the specification of policies is now modular: rather than being defined globally, they are now defined as locally as possible and combined statically. This not only enhances readability, but also maintainability of the model.

The following module details the abstract DKAL-like policy communication model employed in our car registration model, as introduced in [3, §4.3].

```
% Lightweight DKAL %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
entity DKAL {

  import Prelude

  types
   info;

  symbols

% DKAL-like policy predicates and functions:
% A piece of information (represented as an element of
% type info) is known to an agent without involving
% communication at the meta level but possibly at the
% system level.  Typically, this kind of knowledge is
% acquired upon reception of a message over the network.
      knows0 (agent, info): fact;
% A piece of information (represented as an element of
% type info) is known to an agent possibly involving
% communication at the meta level
      knows  (agent, info): fact;
```

```
% Agents may communicate parts of their knowledge at the
% meta level to other principals (this is a different kind
% of communication with respect to communication at the
% system level). It is worth noticing that the meta level
% communication is secure and targeted; e.g., a1->saysTo(a2,x)
% means that agent a1 says a piece of information x to
% agent a2 and the intruder will have no access to x.
% Notice that the piece of information x in a1->saysTo(a2,x)
% is part of the knowledge of agent a1 that might have been
% obtained from some other agent a3.
      saysTo  (agent, agent, info): fact;
% Instead, if we write a1->saysTo0(a2,x), then the piece of
% information x is part of the "internal" knowledge of
% agent a1, i.e. it is not obtained from some other principal
% but only by some computation, e.g. reception of a message
% over the network at the system level.
      saysTo0 (agent, agent, info): fact;

% The function symbols said and said0 allows one to
% characterize how a piece of information has been obtained
% by an agent.  For example, if an agent a1 saysTo an
% agent a2 a piece of information x, then a2 knows
% that a1 said the piece of information x.
       said  (agent, info): info;
% The difference between said and said0 is that the latter
% reflects that the piece of information has been acquired
% by a principal without resorting to communication at the
% meta level.
       said0 (agent, info): info;

% The function symbols tdOn and tdOn0 encode trust
% relationships between agents concerning some piece of
% information.
       tdOn  (agent, info): info;
       tdOn0 (agent, info): info;
% The difference between tdOn and tdOn0 is that the former
% allows agents to delegate trust while the latter does not.
% For example, the piece of information a tdOn x expresses
% not only trust in the agent a about some piece of
% information x but also a permission for agent a to
% delegate the trust about x.

  macros % two useful abbreviations

     A->trusts (B,M) = A->knows(B->tdOn (M));
     A->trusts0(B,M) = A->knows(B->tdOn0(M));

  clauses
```

```
% The following Horn clauses characterize (an
% over-approximation of) the DKAL language.

% Internal knowledge is knowledge
    knowledge0inf(P,AnyThing):
      P->knows (AnyThing) :-
      P->knows0(AnyThing);

% An agent knows whatever is said to him and he/she also
% knows whether the piece of knowledge being communicated
% is based on the internal knowledge of the speaker
% (says2know0) or not (says2knowinf).
    says2know0(P,Q,AnyThing):
      P->knows  (Q->said0(AnyThing)) :-
      Q->saysTo0(P,AnyThing);
    says2knowinf(P,Q,AnyThing):
      P->knows (Q->said(AnyThing)) :-
      Q->saysTo(P,AnyThing);

% An agent P knows a piece of information AnyThing
% whenever P knows that another agent Q said AnyThing
% and P knows that the agent Q is trusted on saying AnyThing
    trustapp0(P,Q,AnyThing):
      P->knows(AnyThing) :-
      P->trusts0(Q,AnyThing) &
      P->knows(Q->said0(AnyThing));
    trustappinf(P,Q,AnyThing):
      P->knows(AnyThing) :-
      P->trusts(Q,AnyThing) &
      P->knows(Q->said(AnyThing));

} % end entity DKAL
```

Next, we give the model of the car registration itself. After specifying the policy-related constants and function names, as well as a macro used as a shorthand for a lengthy term, it contains entity declarations for each role in the scenario. In contrast to the version in [3], all policy handling with Horn clauses, as well as the statements defining the workflow of each role in the scenario, is specified locally to the entities, which provides a stronger modularity. Moreover, the workflow can be described more concisely and in a more readable way using structured OO language-like statements rather than the low-level transitions of ASLan.

```
%% Specification of the Car Registration Office in D5.1
% initial writeup: David von Oheimb and AVANTSSAR team
% adapted for use with orchestration by Marius Minea
% further extended by Gabriel Erzse
```

```
specification CarRegistrationProtocol
channel_model CCM

entity Environment {

  import DKAL ;

  types
    doc      < message;
    decision < message;
    role     < message;

  symbols

    % The empty form, provided by the central repository to
    % anyone upon request.
    emptyDoc : doc;

    % Input predicate (for employees): we abstracted away how
    % an employee decides whether a request is correct or not.
    isok(doc) : fact;

    % Operation of filling an empty document by a citizen.
    fillDoc(agent, doc) : doc;

    % Flags saying that a request has been refused or accepted.
    refused, accepted: decision;
```

Messages passed between agents. Every message will have a first argument of type nat that should be a fresh value for every request and will uniquely identify the request. The responses will refer back to the fresh values sent with the requests.

```
    requestEmptyDoc(nat, agent):message;
    provideEmptyDoc(nat, doc):message;
    submitDoc(nat, agent, message):message;
    docReceived(nat, nat):message;
    requestDocForProcessing(nat, agent):message;
    postponeDoc(nat, agent, message, nat):message;
    provideDocForProcessing(nat, message, nat):message;
    notifyProcessedDoc(nat, agent, decision, agent, nat):message;
    notifyCitizen(nat, message):message;
    noDocAvailable(nat):message;
    storeDocToCentralRep(nat, agent, message):message;
    authorized(nat):message;
    notAuthorized(nat):message;
```

```
% Possible roles of employees.
head, employee: role;

% Meta level construct corresponding to action
% certificates.
canStoreDoc (agent): info;

% Meta level construct corresponding to role certificates.
hasRole (agent, role): info;

% Primitive to create a role certificate by the
% certification authority, i.e.  an employee can either
% be a (simple)  employee or the head of a
% car registration office.
rolecert (agent, role): doc;

% The agents in the system.
regOffice : agent; % Registration Office.
centrRep : agent;  % Central Repository.
theCA: agent;      % Certification Authority.
melinda : agent;   % Head of the Registration Office.
peter : agent;     % Employee of the Registration Office
                   % trusted with storing documents into
                   % the Central Repository.
john : agent;      % Employee of the Registration Office
                   % not trusted  with storing documents
                   %into  the Central Repository.
mike : agent;      % Citizen.
```

Facts for signaling completion of steps in the workflow. They are used for specifying security goals.

```
citizenSubmittedDoc(agent, doc):fact;
citizenReceivedRegNo(agent, doc, nat):fact;
citizenReceivedDecision(agent, nat, decision):fact;
employeeAskedForWork(agent, agent):fact;
employeeAcceptedRequest(agent, agent, nat):fact;
employeePostponedRequest(agent, agent, nat):fact;
employeeRefusedRequest(agent, agent, nat):fact;
centralRepProvidedEmptyDoc(agent, agent):fact;
centralRepAcceptedStoreRequest(agent, agent, agent):fact;
centralRepRejectedStoreRequest(agent, agent, agent):fact;
regOfficeReceivedRequest(agent, agent, nat):fact;
regOfficeSentRequestForProcessing(agent, agent, nat):fact;
regOfficeHasNoPendingRequest(agent, agent):fact;
regOfficeStoredPostponedRequest(agent, agent, nat):fact;
regOfficeNotifiedCitizen(agent, agent, nat):fact;
regOfficeRejectedAccess(agent, agent):fact;
```

```
macros
  signedDoc(Actor,Doc)={Doc}_inv(pk(Actor));
  signedRoleCert(CA,Empl,Role)=
              {rolecert(Empl,Role)}_inv(pk(CA));
```

In the following are shown the different entities of the case study using the main features of the language, i.e. encryption shorthands, secrecy goals, channel goals, asserts, etc.

```
% Citizen
%
% Citizen sends a request to Central Repository for
% obtaining the empty form. Then he fills the form, signs
% it and submits it to the Registration Office.
% The Registration Office assigns a unique registration
% number to the document and informs the
% citizen about this registration number.
% After the document is processed, the Registration
% Office informs the Citizen about the decision, referring
% back to the initial registration number.
entity Citizen(Actor, CentrRep, RegOffice : agent)
{
  symbols
    Doc : doc;
    Token, RegNo : nat;
    Decision : decision;

  body
  {
    % The citizen requests and receives the empty form.
    % This communication is done over regular channels.
    Token := fresh();
    Actor -> CentrRep : requestEmptyDoc(Token, Actor);
    CentrRep -> Actor : provideEmptyDoc(Token, ?Doc);

    % The citizen fills the empty form with its data.
    Doc := fillDoc(Actor, Doc);

    % The citizen submits the filled and signed form.
    % Communication takes place over a confidential
    % channel,  because we don't want the filled document
    % to fall into wrong hands.
    Token := fresh();
    Actor ->* RegOffice :
        submitDoc(Token, Actor, signedDoc(Actor, Doc));
    channel_goal citizen_token_secrecy :
        Actor ->* RegOffice : Token;
    citizenSubmittedDoc(Actor, Doc);
```

```
    % The citizen receives the registration number for
    % the submitted doc. Communication is done over secure
    % channels.
    RegOffice *->* Actor : docReceived(Token, ?RegNo);
    citizenReceivedRegNo(Actor, Doc, RegNo);

    % The citizen receives the decision for his request.
    % Communication is done over secure channels.
    RegOffice *->* Actor :
        notifyCitizen(?RegNo,
              signedDoc(RegOffice, ?Decision));
    citizenReceivedDecision(Actor, RegNo, Decision);
}

goals
    % A citizen who submitted a document should eventually
    % receive from the Registration Office a
    % registration number.
    citizen_eventually_receives_regno :
      forall Cit Doc .
        [] (citizenSubmittedDoc(Cit, Doc) =>
              exists RegNo .
                <> (citizenReceivedRegNo(Cit, Doc, RegNo)));

    % A citizen who receives a registration number should
    % have submitted a documment.
    citizen_receives_regno_only_after_submit :
      forall Cit Doc RegNo .
        citizenReceivedRegNo(Cit, Doc, RegNo) =>
              citizenSubmittedDoc(Cit, Doc);

    % A citizen who received a registration number from
    % the Registration Office should eventually receive a
    % decision about his submitted document.
    citizen_eventually_receives_decision :
      forall Cit Doc RegNo .
        [] (citizenReceivedRegNo(Cit, Doc, RegNo) =>
              exists Decision .
                <> (citizenReceivedDecision(Cit,
                      RegNo, Decision)));

    % A citizen who receives a decision should have
    % received a registration number before.
    citizen_receives_decision_only_after_regno :
      forall Cit RegNo Decision .
        citizenReceivedDecision(Cit, RegNo, Decision) =>
              exists Doc . citizenReceivedRegNo(Cit,
                      Doc, RegNo);
```

```
    % The registration number received by each citizen
    % is unique.
    each_regno_is_unique :
      forall Cit1 Cit2 Doc1 Doc2 RegNo1 RegNo2 .
        !(citizenReceivedRegNo(Cit1, Doc1, RegNo1) &
          citizenReceivedRegNo(Cit2, Doc2, RegNo2) &
          !Cit1=Cit2 &
          RegNo1=RegNo2);
}


% Certification Authority
%
% It has the role of distributing certificates to employees
% of Registration Offices.
% Certificates are distributed over confidential channels,
% because in general the internal organization of
% Registration Offices should be protected.
entity CA(Actor: agent, HeadEmpls, RegularEmpls : agent set)
{
  symbols
    HeadEmpl, RegularEmpl : agent;
    RegularEmplsTemp : agent set;

  body {
    % Each Head employee receives a certificate about
    % itself.
    while (HeadEmpls->contains(?HeadEmpl)) {
      HeadEmpls->remove(HeadEmpl);
      Actor ->* HeadEmpl :
          signedRoleCert(Actor, HeadEmpl, head);
     }

    % Each Regular employee receives a certificate
    % about itself.
    while (RegularEmpls->contains(?RegularEmpl)) {
      RegularEmpls->remove(RegularEmpl);
      Actor ->* RegularEmpl :
          signedRoleCert(Actor, RegularEmpl, employee);
    }
  }
}


% Registration Office Head
%
% Note that we cannot express (but actually do not need
% here) the fact that Head is a special form of Employee.
% The role of the Head is to distribute certificates to
```

```
% trusted employees, so that they can store documents
% in the central repository.
entity Head(Actor: agent, TrustedEmpls,
      UntrustedEmpls: agent set)
{
  symbols
    Empl : agent;
    TrustedEmplsTemp : agent set;
    Cert : message;

  clauses
    % The head of the car registration office, e.g. Melinda,
    % decides that all her older employees, e.g. Peter,
    % have enough experience to store accepted requests
    % in the central repository.
    % There are also employees who don't have enough
    % experience, so they can't be trusted to store data
    % in the central repository.
    trustedEmplsCanStoreDoc(Actor,TrustedEmpls) :
      forall Empl.
      Actor->knows0(Empl->canStoreDoc) :-
        contains(TrustedEmpls, Empl);
    trustedEmplsCanStoreDoc2(Actor,TrustedEmpls) :
      forall Empl.
      Actor->knows0(Empl->canStoreDoc) :-
        contains(TrustedEmplsTemp, Empl);

    % The head of the car registration office, once he/she
    % has decided to grant the capability of storing
    % processed requests in the central repository to some
    % of his/her employees, he/she is willing to share this
    % information with the respective employees.
    grantStoreRight(Actor,Empl) :
      forall Empl.
        Actor->saysTo0(Empl,Empl->canStoreDoc) :-
          Actor->knows0(Empl->canStoreDoc);

  body {
    % The Head does not do anything until he/she receives
    % the needed certificate from the CA.
    theCA ->* Actor : ?Cert;

    % Make sure the received certificate is signed
    % by the CA.
    assert head_certificate_ok :
       Cert = signedRoleCert(theCA, Actor, head);

    % The received certificate should be a secret between
    % the CA and the Head, for now. Later it will be
```

```
      % shared with trusted employees.
      secrecy_goal head_certificate_secret :
         theCA, Actor : Cert;

      % The Head sends its certificate to all trusted
      % employees.
      while (TrustedEmpls->contains(?Empl)) {
        TrustedEmpls->remove(Empl);
        TrustedEmplsTemp->add(Empl);
        Actor ->* Empl : ?Cert;
        % Share the secrecy of the certificate with the
        % employee.
        head_certificate_secret_set(IID)->add(Empl);
      }

      % Restore the set of trusted employees.
      while (TrustedEmplsTemp->contains(?Empl)) {
        TrustedEmplsTemp->remove(Empl);
        TrustedEmpls->add(Empl);
      }
    }
  }

  % Regular Employee of Registration Office
  %
  % Continuously asks for work from the Registration Office
  % and solves car registration requests sent
  % by the Reg Office.
  % Also accepts certificates about employees
  % (not only himself/herself), in which case he/she
  % makes the knowledge derived from the certificate
  % available at the meta level.
  entity Employee(Actor, RegOffice, CentrRep : agent)
  {
    symbols
      Role : role;
      Token, RegNo : nat;
      Cit, Empl1: agent;
      Doc : doc;
     SignedRequest: message;

    clauses
      % An employee getting a certificate from the
      % certification authority about some agent in the car
      % registration office is willing to tell this to
      % the central repository.
      cert1(Actor,Cert,CentrRep) :
        Actor->saysTo(CentrRep,theCA->said0(Cert)) :-
          Actor->knows(theCA->said0(Cert));
```

```
    % An employee getting a certificate from the head of the
    % car registration office about some agent
    % that is entitled to store documents
    % in the central repository is willing
    % to tell this to central repository.
    cert2(Actor,Head,Cert,CentrRep):
      Actor->saysTo(CentrRep,Head->said0(Cert)) :-
        Actor->knows(Head->said0(Cert)) &
        Actor->knows(theCA->said0(Head->hasRole(head)));

  body {
    while(true) {
      select {
        % The employee asserts at the meta level
        % that he/she has received a certificate about
        % the role of a (possibly different) agent Empl1.
        on (? ->* Actor : signedRoleCert(theCA,
              ?Empl1,?Role)) : {
          Actor->knows0(theCA->said0(Empl1->hasRole(Role)));
        }

        % Employee can always ask for work
        % (a new registration request).
        on (true) : {
          Token := fresh();
          Actor *->* RegOffice :
              requestDocForProcessing(Token, Actor);
          employeeAskedForWork(Actor,RegOffice);

          select {
            % If the registration office says no
            % registration request is available, do nothing.
            on (RegOffice *->* Actor :
              noDocAvailable(Token)) : {}

            % If the registration office says not
            % authorized, then again do nothing.
            on (RegOffice *->* Actor :
              notAuthorized(Token)) : {}

            % If there is a registration request available,
            % process it.
            on (RegOffice *->* Actor :
              provideDocForProcessing(Token,
                      ?SignedRequest, ?RegNo)) :
            {
              % The registration request is accepted if
              % - it meets certain criteria (abstracted
```

```
%   away in this model);
% - is correctly signed by the sender.
if (isok(?Doc) & SignedRequest =
        signedDoc(?Cit, ?Doc)) {
  % The employee asks the central repository
  % to store the accepted registration request.
  Token := fresh();
  Actor *->* CentrRep :
        storeDocToCentralRep(Token, Actor,
           signedDoc(Actor,
                accepted.signedDoc(Cit, Doc)));

  select {
    % If the central repository authorized the
    % request, then notify the registration
    % office, which in turn will notify
    % the citizen.
    on (CentrRep *->* Actor :
         authorized(Token)) : {
      % The employee notified the registration
      % office about the decision.
      % The registration office will further
      % notify the citizen.
      employeeAcceptedRequest(Actor,
        Cit, RegNo);
      Token := fresh();
      Actor *->* RegOffice :
        notifyProcessedDoc(Token,
                Actor, accepted, Cit, RegNo);
    }

    % If the central repository rejected
    % the operation, then send the citizen's
    % registration document back to
    % the registration  office, and let some
    % other  employee handle it.
    on (CentrRep *->* Actor :
        notAuthorized(Token)) : {
      employeePostponedRequest(Actor,
        Cit, RegNo);
      Token := fresh();
      Actor *->* RegOffice :
        postponeDoc(Token, Actor,
                signedDoc(Cit, Doc), RegNo);
    }
  }
} else {
  % The employee refuses a request Doc by
  % a Citizen if the request is not correcly
```

```
                % signed or some other criteria
                % (abstracted away in this specification)
                % is not met.
                employeeRefusedRequest(Actor, Cit, RegNo);
                Token := fresh();
                Actor *->* RegOffice :
                  notifyProcessedDoc(Token,
                      Actor, refused, Cit, RegNo);
              }
            } % end on doc available
          } % end select reg office response
        } % end on (true)
      } % end select in while(true)
    } % end while(true)
  }
}

% Car Registration Office
%
% Holds a local database with car registration requests.
% Registration requests are received from citizens and
% stored into the local database.
% Employees of the registration office query for unsolved
% registration requests. The employees may send the unsolved
% requests back, with possible additional comments
% (the comments are abstracted away in this model).
% Solved requests will be  stored by authorized
% employees directly into the central repository.
entity RegistrationOffice(Actor: agent, Empls : agent set) {
  symbols
    Requestor, Cit : agent;
    SignedRequest : message;
    Doc : doc;
    Decision : decision;
    Token, RegNo : nat;
    DB : (message, nat) set;

  body {
    % Initially there is no request in the local database.
    DB := {};

    % The registration office runs indefinitely.
    while(true) {
      select {
        % Anyone can submit a filled and signed car
        % registration request. A unique registration number
        % is assigned to the request. The request,
        % together with the registration number, are stored
        % in  the local database for later processing.
```

```
% The registration number is sent back to the
% citizen.
on (?Requestor ->* Actor :
      submitDoc(?Token, Requestor, ?SignedRequest)):
  channel_goal citizen_token_secrecy :
    Requestor ->* Actor : Token;
{
  RegNo := fresh();
  add(DB, (SignedRequest,RegNo));
  regOfficeReceivedRequest(Actor, Requestor, RegNo);
  Actor *->* Requestor : docReceived(Token, RegNo);
}

% Employees can request a document for processing.
on (?Requestor *->* Actor :
      requestDocForProcessing(?Token, Requestor)) : {
  % If the requestor is indeed an employee of this
  % registration office, then his request will be
  % handled.
  if (Empls->contains(Requestor)) {
    % If there is a pending car registration request,
    % send it to the employee for processing.
    if (DB->contains((?SignedRequest,?RegNo))) {
      DB->remove((SignedRequest,RegNo));
      regOfficeSentRequestForProcessing(Actor,
              Requestor, RegNo);
      Actor *->* Requestor :
        provideDocForProcessing(Token,
              SignedRequest, RegNo);
    }
    % If there is no pending registration request,
    % just inform the employee about this.
    else {
      regOfficeHasNoPendingRequest(Actor, Requestor);
      Actor *->* Requestor : noDocAvailable(Token);
    }
  }
  % If the requestor is not an employee of this
  % registration office, then his request is rejected.
  else {
    regOfficeRejectedAccess(Actor, Requestor);
    Actor *->* Requestor : notAuthorized(Token);
  }
}

% Employees can send back a document for later
% processing.
on (?Requestor *->* Actor :
      postponeDoc(?Token, ?Requestor,
```

```
                    ?SignedRequest,?RegNo)) : {
          % If the requestor is a valid employee of this
          % registration office, then the request
          % will be processed.
          if (Empls->contains(Requestor)) {
            DB->add((SignedRequest, RegNo));
            regOfficeStoredPostponedRequest(Actor,
              Requestor, RegNo);
          }
          % If the requestor is not an employee of this
          % registration office, then the request
          % will be rejected.
          else {
            regOfficeRejectedAccess(Actor, Requestor);
            Actor *->* Requestor : notAuthorized(Token);
          }
        }

        % Employees can instruct the registration office
        % to notify customers about decisions taken.
        on (?Requestor *->* Actor :
              notifyProcessedDoc(?Token,
              ?Requestor, ?Decision, ?Cit, ?RegNo)) : {
          % If the requestor is a valid employee of this
          % registration office, then the request
          % will be processed.
          if (Empls->contains(Requestor)) {
            regOfficeNotifiedCitizen(Actor, Cit, RegNo);
            Actor *->* Cit :
                notifyCitizen(RegNo,
                      signedDoc(Actor, Decision));
          }
          % If the requestor is not an employee of this
          % registration  office, then the request
          % will be rejected.
          else {
            regOfficeRejectedAccess(Actor, Requestor);
            Actor *->* Requestor : notAuthorized(Token);
          }
        }
      } % end select
    } % end while
  }
}


% Central Repository
%
% Has the role of providing empty registration forms to
```

```
% anyone and storing documents sent by authorized employees
% of  registration offices.
entity CentralRepository(Actor: agent)
{
  clauses
    % The central repository trusts the certification
    % authority of the car registration office.
    centrRepTrustCA(Actor,AnyThing) :
      Actor->trusts0(theCA,AnyThing) :-
       Actor->knows(theCA->said0(AnyThing));

    % The central repository trusts anyone when
    % communicating a certificate emitted by the certification
    % authority of the car registration office.
    centrRepTrustAnyoneViaCA(Actor,AnyOne,AnyEmpl,AnyRole) :
      Actor->trusts(AnyOne,theCA->
              said0(AnyEmpl->hasRole(AnyRole))) :-
        Actor->knows(AnyOne->said(theCA->
              said0(AnyEmpl->hasRole(AnyRole))));

    % The central repository trusts anyone presenting a
    % certificate emitted by the head of a car registration
    % office if there exists a certificate of the fact that
    % the emitter is the head of the car registration office
    centrRepTrustAnyoneViaHead(Actor,Head,AnyOne,AnyThing) :
      Actor->trusts(AnyOne,Head->said0(AnyThing)) :-
        Actor->knows(theCA->said0(Head->hasRole(head))) &
        Actor->knows(AnyOne->said(Head->said0(AnyThing)));

    % The central repository trusts the head of a car
    % registration office when he/she emits a certificate
    % about the capability of storing processed requests,
    % once it has checked that there are certificates
    % proving that he/she is the head of a car registration
    % office and  that the subject of his/her certificate
    % is an employee
    centrRepTrustHead(Actor,Head,Empl):
      Actor->trusts0(Head,Empl->canStoreDoc) :-
      Actor->knows(theCA->said0(Head->hasRole(head))) &
      Actor->knows(theCA->said0(Empl->hasRole(employee)));

  symbols
    CentrDB : message set;
    Requestor, Cit : agent;
    SignedDoc : message;
    Decision : decision;
    Doc : doc;
    Token : nat;
```

```
body {
  while(true) {
    select {
      % Anybody can obtain the empty form.
      on (?Requestor -> Actor :
            requestEmptyDoc(?Token, Requestor)) : {
          Actor -> Requestor :
            provideEmptyDoc(Token, emptyDoc);
          centralRepProvidedEmptyDoc(Actor, Requestor);
      }

      % Upon reception of a request to store a document
      % in the database, check whether the employee asking
      % for this to be done has the right to do this.
      % If so, the request is added to the database.
      on (?Requestor *->* Actor :
            storeDocToCentralRep(?Token,
           ?Requestor, ?SignedDoc)) : {
        % If the employee has the right to store documents,
        % then his/her request is processed.
        if (Actor->knows(Requestor->canStoreDoc)) {
          % The document must be double signed in order
          % to be accepted.
          if (SignedDoc =
              signedDoc(?Requestor,
                    ?Decision.signedDoc(?Cit, ?Doc))) {
            add(CentrDB, SignedDoc);
            centralRepAcceptedStoreRequest(Actor,
                    Requestor, Cit);
            Actor *->* Requestor : authorized(Token);
          }
          % If the document is not correctly signed, then
          % the request is rejected.
          else {
            centralRepRejectedStoreRequest(Actor,
                    Requestor, Cit);
            Actor *->* Requestor : notAuthorized(Token);
          }
        }
        % If the employee does not have the right to
        % store documents, then the request is rejected.
        else {
          centralRepRejectedStoreRequest(Actor,
            Requestor, Cit);
          Actor *->* Requestor : notAuthorized(Token);
        }
      }
    } % end select
  } % end while
```

```
    }
  }
```

After described all the entities in the model, in the `body` of the `enviroment` the system is being initialized.

```
% Initialization of the overall system.
 body {
   % Replacement for access control policies.
   knows(centrRep, canStoreDoc(peter));

   % We abstract away the criteria for the correctness of
   % registration requests. We simply state that the
   % request of Mike is correct, while others' requests
   % are not corect.
   isok(fillDoc(mike, emptyDoc));

   % The certification authority receives two sets:
   % one with the Head employees and another one with
   % regular employees.
   new CA(theCA, {melinda}, {peter, john});

   % The central repository.
   new CentralRepository(centrRep);

   % The registration office.
   new RegistrationOffice(regOffice, {peter, john});

   % Melinda, the Head employee of the registration office.
   % It receives two sets of employees, one with older,
   % trusted employees, and another one with newer, not yet
   % trusted employees.
   new Head(melinda, {peter}, {john});

   % Peter, an old, experienced employee of the
   % registration office.
   new Employee(peter, regOffice, centrRep);

   % John, a newer, unexperienced employee of the
   % registration office.
   new Employee(john, regOffice, centrRep);

   % Mike, a citizen who submits a valid registration
   % request.
   new Citizen(mike, centrRep, regOffice);
 }
```

In the goal section we can describe security properties have to be checked during analysis. For instance, the Central Repository entity should never

know that `John` is authorized to store documents into it.

```
goals

    centrRepKnowsWrongAccessRights :
      !knows(centrRep, canStoreDoc(john));
```

The following are goals that express success scenarios. They are used only for making sure that the entities can work together properly in order to achieve a positive outcome. Note that because they are success scenarios, we must negate them if we want them to be found as attacks. Activate these goals one at a time, with all other goals commented out.

```
    % The success scenario when Mike registers his car
    % and his request is processed by Peter (the employee
    % who is authorized to store documents in the central
    % repository).
    Mike_registers_his_car_in_one_step:
      !exists CR Cit Doc RegNo RO Empl .
          centralRepProvidedEmptyDoc(CR, mike) &
          citizenSubmittedDoc(mike, Doc) &
          citizenReceivedRegNo(Cit, Doc, RegNo) &
          regOfficeReceivedRequest(RO, Cit, RegNo) &
          CR->knows(canStoreDoc(Empl)) &
          employeeAskedForWork(Empl, RO) &
          regOfficeSentRequestForProcessing(RO, Empl, RegNo) &
          centralRepAcceptedStoreRequest(CR, Empl, Cit) &
          employeeAcceptedRequest(Empl, Cit, RegNo) &
          regOfficeNotifiedCitizen(RO, Cit, RegNo) &
          citizenReceivedDecision(Cit, RegNo, accepted);

    % The success scenario when Mike registers his car, but
    % his request is processed first by John, who cannot store
    % documents in the central repository. John sends back the
    % request to the registration office, and then Peter takes
    % the request and solves it.
    Mike_registers_his_car_in_two_steps:
      !exists CR Cit Doc RegNo RO UEmpl TEmpl .
          centralRepProvidedEmptyDoc(CR, Cit) &
          citizenSubmittedDoc(Cit, Doc) &
          citizenReceivedRegNo(Cit, Doc, RegNo) &
          regOfficeReceivedRequest(RO, Cit, RegNo) &
          !CR->knows(canStoreDoc(UEmpl)) &
          employeeAskedForWork(UEmpl,RO) &
          regOfficeSentRequestForProcessing(RO, UEmpl, RegNo) &
          centralRepRejectedStoreRequest(CR, UEmpl, Cit) &
          employeePostponedRequest(UEmpl, Cit, RegNo) &
          regOfficeStoredPostponedRequest(RO, UEmpl, RegNo) &
```

```
            employeeAskedForWork(TEmpl, RO) &
            CR->knows(canStoreDoc(TEmpl)) &
            regOfficeSentRequestForProcessing(RO, TEmpl, RegNo) &
            centralRepAcceptedStoreRequest(CR, TEmpl, Cit) &
            employeeAcceptedRequest(TEmpl, Cit, RegNo) &
            regOfficeNotifiedCitizen(RO, Cit, RegNo) &
            citizenReceivedDecision(Cit, RegNo, accepted);

    % Success scenario when the Central Repository knows the
    % exact roles and access rights of each employee.
    Centr_Rep_Knows_Correct_Access_Rights :
       !exists CR M P J .
         knows(CR, hasRole(M, head)) &
         knows(CR, hasRole(P, employee)) &
         knows(CR, hasRole(J, employee)) &
         knows(CR, canStoreDoc(P)) &
        !knows(CR, canStoreDoc(J));
}
```

Note that the integrity goal of documents stored in the central repository, as stated in [3], is modeled above as an assertion with the name `stored_request_is_double_signed_and_sent_by_citizen` that is supposed to hold at specific points in the execution of the system.

Finally, a sample derivation of `centrRep->knows(peter->canStoreDoc)` is given to demonstrate the interplay of the Horn clauses and message transmissions defined in the above model. Note the names of Horn clauses with their actual parameters, which greatly help to trace the use of Horn clauses.

```
centrRep->knows(peter->canStoreDoc)
<= [ via trustapp0(centrRep,melinda,peter->canStoreDoc) ]
  centrRep->trusts0(melinda,peter->canStoreDoc)
  <= [ via centrRepTrustHead(melinda,peter) ]
    centrRep->knows(theCA->said0(melinda->hasRole(head)))
    <= [ via trustappinf(centrRep,peter,theCA->
     said0(melinda->hasRole(head))) ]
      centrRep->trusts(peter,theCA->said0(melinda->hasRole(head)))
      <= [ via centrRepTrustAnyoneViaCA(peter,melinda->hasRole(head)) ]
      centrRep->knows(peter->said(theCA->said0(melinda->hasRole(head))))
      <= [via says2knowinf(centrRep,peter,theCA->
       said0(melinda->hasRole(head)))]
        peter->saysTo(centrRep,theCA->said0(melinda->hasRole(head)))
        <= [ via Cert1(peter,melinda->hasRole(head),centrRep) ]
          peter->knows(theCA->said0(melinda->hasRole(head)))
  <= [ via knowledge0inf(peter,theCA->said0(melinda->hasRole(head))) ]
           peter->knows0(theCA->said0(melinda->hasRole(head)))
    <= [ via Employee code ]
      receive(signedRoleCert(melinda,head))
      <= [ via CA code ]
```

```
        send(signedRoleCert(melinda,head)) % body
    centrRep->knows(theCA->said0(peter->hasRole(employee))
    <= [ via trustappinf(centrRep,peter,theCA->
     said0(peter->hasRole(employee))) ]
      centrRep->trusts(peter,theCA->said0(peter->hasRole(employee)))
      <= [ via centrRepTrustAnyoneViaCA(peter,peter->hasRole(employee)) ]
      centrRep->knows(peter->said(theCA->said0(peter->hasRole(employee))))
      <=[via says2knowinf(centrRep,peter,theCA->
       said0(peter->hasRole(employee)))]
        peter->saysTo(centrRep,theCA->said0(peter->hasRole(employee)))
        <= [ via Cert1(peter,peter->hasRole(employee),centrRep) ]
          peter->knows(theCA->said0(peter->hasRole(employee)))
  <= [ via knowledge0inf(peter,theCA->said0(peter->hasRole(employee))) ]
            peter->knows0(theCA->said0(peter->hasRole(employee)))
    <= [ via Employee code ]
      receive(signedRoleCert(peter,employee))
      <= [ via CA code ]
        send(signedRoleCert(peter,employee)) % body
  centrRep->knows(melinda->said0(peter->canStoreDoc))
  <= [ via trustappinf(centrRep,peter,melinda->said0(peter->canStoreDoc)) ]
    centrRep->trusts(peter,melinda->said0(peter->canStoreDoc))
    <= [ via centrRepTrustAnyoneViaHead(peter,peter->canStoreDoc) ]
      centrRep->knows(theCA->said0(melinda->hasRole(head))) % see above
    centrRep->knows(peter->said(melinda->said0(peter->canStoreDoc)))
    <= [ via says2knowinf(centrRep,peter,melinda->said0(peter->canStoreDoc) ]
      peter->saysTo(centrRep,melinda->said0(peter->canStoreDoc))
      <= [ via Cert2(peter,melinda,peter->canStoreDoc,centrRep) ]
        peter->knows(melinda->said0(peter->canStoreDoc))
        <= [ via says2know0(peter,melinda,peter->canStoreDoc) ]
          melinda->saysTo0(peter,peter->canStoreDoc)
          <= [ via GenerateCert(melinda,peter,peter) ]
            melinda->knows0(peter->canStoreDoc) % initial fact
        peter->knows(theCA->said0(melinda->hasRole(head)))
<= [ via knowledge0inf(peter,theCA->said0(melinda->hasRole(head))) ]
          peter->knows0(theCA->said0(melinda->hasRole(head)))
  <= [ via Employee code ]
    receive(signed(rolecert(melinda,head))
    <= [ via CA code ]
      send(signedRoleCert(melinda,head)) % body
```

6 Conclusion

We have presented ASLan and ASLan++, the AVANTSSAR specification languages. The low-level language ASLan 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++ and ASLan allow 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 these languages is demonstrated via the formalized problem cases, reported in Deliverable D5.2 [7].

As discussed in the AVANTSSAR description of work (Annex I), the refined versions of ASLan and ASLan++, as presented in this deliverable, cover static and dynamic service and policy composition aspects. The specification languages and the entire AVANTSSAR Platform support full-fledged specification and analysis of the case studies and the industrial-strength applications that we have considered.

References

[1] A. Armando, R. Carbone, and L. Compagna. LTL Model Checking for Security Protocols. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF20)*, pages 385–396. IEEE Computer Society Press, 2007.

[2] N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 86–99, 1998. citeseer.nj.nec.com/asokan98asynchronous.html.

[3] AVANTSSAR. Deliverable 2.1: Requirements for modelling and ASLan v.1. Available at http://www.avantssar.eu, 2008.

[4] AVANTSSAR. Deliverable 3.3: Attacker models. Available at http://www.avantssar.eu, 2008.

[5] AVANTSSAR. Deliverable 5.1: Problem cases and their trust and security requirements. Available at http://www.avantssar.eu, 2008.

[6] AVANTSSAR. Deliverable 2.2: ASLan v.2 with static service and policy composition. Available at http://www.avantssar.eu, 2009.

[7] AVANTSSAR. Deliverable 5.2: Formalized problem cases. Available at http://www.avantssar.eu, 2010.

[8] AVISPA. Deliverable 2.1: The High-Level Protocol Specification Language. http://www.avispa-project.org, 2003.

[9] AVISPA. Deliverable 2.3: The Intermediate Format. http://www.avispa-project.org, 2003.

[10] C. Dilloway and G. Lowe. On the specification of secure channels. In *Proceedings of WITS '07*, 2007.

[11] 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.

[12] 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.

[13] 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/.

[14] U. M. Maurer and P. E. Schmid. A calculus for security bootstrapping in distributed systems. *Journal of Computer Security*, 4(1):55–80, 1996.

[15] S. Mödersheim. *Models and Methods for the Automated Analysis of Security Protocols*. PhD Thesis, ETH Zurich, 2007. ETH Dissertation No. 17013.

[16] S. Mödersheim and L. Viganò. Secure Pseudonymous Channels. In *Proceedings of Esorics'09*, LNCS 5789, pages 337–354. Springer-Verlag, 2009.

[17] 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.

[18] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), december 1978.