Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures

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

www.avantssar.eu

Deliverable D2.2
ASLan v.2 with static service and policy composition

Abstract

This deliverable describes ASLan v.2, the second version of the ASLan language for specifying security-sensitive service-oriented architectures, the associated security policies, and their trust and security properties. In particular, ASLan v.2 allows for the formal specification of static service and policy composition.

Deliverable details

Deliverable version: *v1.0* Classification: *public*
Date of delivery: *08.07.2009* Due on: *30.06.2009*
Editors: *UNIVR, ETH Zurich, UGDIST, IBM, IEAT, SAP, SIEMENS (principal editors).*
INRIA, UPS-IRIT, OpenTrust (secondary editors) Total pages: *92*

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

List of Tables

1 Introduction

This deliverable describes ASLan v.2, the second version of the ASLan language. ASLan is a formal language for specifying security-sensitive service-oriented architectures, the associated security policies, and their trust and security properties. The syntax and semantics of ASLan v.1, the first version of the language, was described in deliverable D2.1 [3] by extending the Intermediate Format IF [7], a specification language that several of the partners of the AVANTSSAR consortium developed in the context of the AVISPA project. 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 v.1 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 v.1 extends IF with:

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 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 v.2 is conceptually more high-level than ASLan v.1 and is defined by translation to ASLAN v.1.1, which is an extension of ASLan v.1 with some useful features that make it suitable as a target of the translation, and which is the actual input language to the validation tools that comprise the AVANTSSAR Platform. The platform will thus support both ASLan v.1.1 and ASLan v.2, with updates and extensions to either of them whenever necessary. In particular, we have been developing ASLan v.2 to achieve the following design goals:

- The language should be expressive enough to model a wide range of service-oriented architectures while at the same time allow for succinct specifications.

- The language should allow for the (architecture-level) 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.

In order to formally model static service and policy composition, ASLan v.2 introduces a number of improvements with respect to v.1, such as:

- Control flow constructs (e.g. `while` and `if`) allow for better readability and conciseness of the specifications, and make the specification 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 certain authorization requests as well as the various trust relationships between entities.
- Furthermore, ASLan v.2 provides easier ways to specify communication and service compositionality by using a suitable, intuitive notation for channels that are used both as service assumptions and as service goals.

Structure of the document In § 2, we describe ASLan v.1.1, which extends ASLan v.1 of D2.1 [3] so as to be suitable as a target language for the translation of ASLan v.2.

In § 3, we give a gentle introduction to ASLan v.2 describing its syntax, while in § 4 we give a procedure for translating ASLan v.2 specifications into ASLan v.1.1 specifications. This procedure therefore defines the semantics of ASLan v.2 in terms of ASLan v.1.1, and will serve as a basis for the implementation of the translation software to be integrated in the AVANTSSAR Platform.

In § 5, we then assess ASLan v.2 against a selection of problem cases taken from Deliverable D5.1 [5]; see also Deliverable D2.1 [3], in which we described the requirements for ASLan that we extracted from the case studies and the corresponding features of ASLan.

We conclude in § 6 by briefly summing up and highlighting the main features that will be included in the next versions of ASLan in future deliverables. In particular, ASLan v.3, the final version of the language to be

delivered in D2.3, will include dynamic service and policy composition, so that the language and the whole AVANTSSAR Platform will be applicable for the full-fledged specification and analysis of the case studies.

2 ASLan v.1.1: an extension of ASLan v.1

As explained above, ASLan v.2 is conceptually more high-level than ASLan v.1 and is defined by translation to an extension of ASLan v.1, called ASLan v.1.1, which is the actual input language to the validation tools that comprise the AVANTSSAR Platform. The platform will thus support both languages, with updates and extensions to either of them whenever necessary.

In this section, we describe ASLan v.1.1.

2.1 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, the typing can be helpful for the tools to reduce the size of the transition system by excluding ill-typed actions of the intruder. While this in general means a restriction that can exclude attacks, for many cases typing can be justified [9]. Moreover, the user may of course also choose to deliberately abstract from type-flaw attacks (even though the user cannot justify it).[1]

In the first version of ASLan (of deliverable D2.1 [3]) we had simply defined that variables can only be instantiated with terms of the respective types. While this is intuitively clear, the formal definition of a type system can be subtle and this definition is thus crucial for the precise interpretation. Therefore, we have decided to formally define the type system as part of this update.

- 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 subtyping, a constant may be part of other types as well). The disjointness of the $\mathcal{C}_.$ 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 $\mathcal{C}_{\texttt{agent}} \cap \mathcal{C}_{\texttt{msg}} = \emptyset$.

[1] We also allow the option that 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 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 $\mathcal{L}_{\tau_1} \subseteq \mathcal{L}_{\tau_2}$. (Note 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 `msg` $\times$ `msg` $\rightarrow$ `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 every $f(\tau_1, \ldots, \tau_n) : \tau$.

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 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.

2.2 Universally Quantified Horn Clauses

As a second extension to the ASLan v.1 presented in D2.1 [3], we describe here a slight generalization of Horn clauses. So far, we have allowed the specification of Horn clauses where the variables occurring on the left-hand side of the clause are a proper subset of the right-hand side variables. This excludes Horn clauses like the following:

$$trustedOn(A, X) :- trusted(B).delegates(B, A)$$

Here, X is a "free" variable of the left-hand side which is forbidden by the ASLan v.1 of D2.1. Such a situation is, however, quite common, especially for rules with an empty right-hand side; intuitively, one would specify that the left-hand side fact holds for every value of the free variable, i.e. interpret it as being universally quantified:

$$\forall X.\, trustedOn(A, X) :- trusted(B).delegates(B, A)$$

Note that one may read this rule in two slightly different (but equivalent) ways, depending on the scope of the quantification: namely either as an infinite set of rules or as one rule yielding an infinite set of facts. The former representation gives a straightforward way to describe the semantics, while the latter one is the key for a simple implementation.

More in detail, the semantics of this extension is defined by a translation of universally quantified Horn clauses into infinite sets of quantifier-free Horn clauses. Consider the Horn clause

$$\forall X_1, \ldots, X_n.\, L :- R$$

where $vars(L) \subseteq vars(R) \cup \{X_1, \ldots, X_n\}$.[2] We also consider that each X_i has a unique type τ_i. The type system defines in a straightforward way a set of all terms that have a particular type. Let us denote this set as $\mathcal{L}_\tau$ for type τ. Then, the meaning of the above rule is the following set of quantifier-free Horn clauses:

$$\{L[X_1 \mapsto t_1, \ldots, X_n \mapsto t_n] :- R \mid t_1 \in \mathcal{L}_{\tau_1} \wedge \ldots \wedge t_n \in \mathcal{L}_{\tau_n} \wedge \bigcup_{i=1}^{n} vars(t_i) = \emptyset\}$$

[2]The tools should give a warning if $\{X_1, \ldots, X_n\} \cap vars(R) \neq \emptyset$ as in this case the modeler potentially has made a mistake.

We then interpret the specification as defined in deliverable D2.1 with the resulting quantifier-free Horn clauses.

2.3 Micro and Macro Steps

For the translation from ASLan v.2 to v.1.1, 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 § 4.7.

To support this concept in ASLan v.1.1, 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 in D2.1. 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 v.2 gentle introduction and syntax

We have defined the syntax of ASLan v.2 to achieve the following design goals:

- The language should be expressive enough to model a wide range of service-oriented architectures while at the same time allow for succinct specifications.
- The language should allow for the (architecture-level) 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 Entities

The top level constructs are *entities*, which are similar to classes in Java or roles in HLPSL[3]. 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 (typed) parameters and contain sub-entities with nested scoping, such that variables and other items declared in outer entities are inherited to the current one, where inner declarations hide any outer declarations with the same name. 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). If an entity contains a parameter named `Actor`, it must be of type `agent`. When a role is instantiated (executed), the agent given as actual value of the parameter `Actor` defines the agent that executes the role, which is important for reasoning about the knowledge of an agent and the security properties needed or provided by the role.

An entity may import other entities contained in separate files (with the filename being the entity name with the extension `.aslan` appended to it), which typically contain collections of declarations known as *modules*.

[3]See [6] for a detailed description of the high-level protocol specification language HLPSL of the AVISPA Tool for security protocol analysis.

An entity may be declared to be "compressed", which means that the execution of the statements in its body is essentially atomic. By default, compressed execution is broken up (such that other entities and the intruder may interfere) just before the reception of messages. Optionally, one may specify an alternative set of "breaking events".

3.2 Declarations

Within an entity, declarations of types, variables, constants, functions, macros, Horn clauses and equalities may be freely mixed. Note, however, that constants, functions and equalities do always have global scope, i.e. like declaring them in the root entity. All other declarations are effective in the scope of the entity including sub-entities.

Types may have subtypes, e.g. `agent < msg`, and parameters. The only types that have parameters are tuples (e.g. `agent * msg`) and sets (e.g. `msg set`).

A communication-specific feature are compound types, which are a shorthand for specifying structural constraints on (mostly message) terms. For example, a variable declaration

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

is expanded to a set of declarations

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

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

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

The names of constants and functions must be unique, i.e. not overloaded.

Like in C, *macros* are used to shorten and simplify lengthy 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.

Horn clauses define Prolog-style rules for "spontaneously" producing facts whenever the conditions on the right-hand side are fulfilled. Both the LHS and RHS may refer to local or inherited variables of the entity the clause is defined in, making it applicable only when the entity instances "owning" these variables are present. All other variables in the Horn 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 Horn clause, which is useful for documenting derivations that involve the Horn 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. f(X,Y) :- g(X) & g(Z)
```

Equalities are used to describe algebraic properties like the associativity of concatenation. 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.

3.3 Terms

Terms may include variables, constants, function applications (but not functions themselves), and specific type constructors for tuples and set literals. We denote the concatenation of messages `M1` and `M2` as `M1.M2`.

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

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

3.4 Channels

For (potentially pseudonymous) secure channels, we briefly recall here the notation we introduced in Deliverable D3.3 (Attacker models) [4] and which is now integrated into ASLan. We use an intuitive notation from [12], where a secure end-point of a channel is marked by a bullet with the following informal meaning:

- $A \rightarrow\bullet\, B : M$ represents a *confidential channel*.
- $A \bullet\!\rightarrow B : M$ represents an *authentic channel*.
- $A \bullet\!\rightarrow\bullet\, B : M$ represents a *secure channel*, i.e. a channel that is both authentic and confidential.

These channel kinds are defined formally in D3.3 [4], where we also consider other kinds; see also [1,14]. Here, we introduce a similar notation to represent another kind of channel:

- $A \Rightarrow B : M$ represents a *resilient channel*.

This notation can be combined with the previous ones. For instance, $A \bullet\!\Rightarrow\!\bullet\ B : M$ 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. Thus, the condition that a channel *ch* is resilient can be formalized by the following LTL formula:

$$\mathit{resilient}(ch) := \mathbf{G}\ \forall(\mathtt{sent}(a', a, b, m, ch) \Rightarrow \mathbf{F}\ \mathtt{rcvd}(b, a, m, ch))$$

While [12] 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. A channel goal has the form

```
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 TLS without client authentication (adding a suitable payload message) is `[Client] *->* Server: Payload`.

As an example consider the following specification:

```
entity Session (A, B: agent) {
  symbols Payload: payload;
  entity Alice(Actor, B: agent) {
    ...
    Actor.send(B,...Payload...)
  }
  entity Bob(Actor, A: agent) {
    symbols Payload: payload;
    ...
    Actor.receive(A,...?Payload...)
  }
  body {
    new Alice(A,B);
```

```
    new Bob  (B,A);
  }
  goals
    authentic_transmission: A *-> B: Payload;
}
```

A channel goal is part of the goals section of an entity (in the example: `Session`) and must fulfill the following syntactic restrictions:

- There are (at least) two sub-entities, namely one that represents the sender role and one that represents the receiver role. In the above example, the sender is `Alice` and the receiver is `Bob`.
- For each of these sub-entities, exactly one instance is created with the real names of both the sender and receiver, as given in the goal, occurring as actual parameters of the instance, such that the sender's name is the actor of the sender entity and the receiver's name is the actor of the receiver entity. For instance, in the above example where the goal is `A` $\bullet\!\!\rightarrow$ `B: Payload`, one sender entity instance is created where its actor is `A` and the second parameter is `B`, and one receiver instance is created where its actor is `B` and the second argument is `A`.
- The entity has a `Payload` variable of the special type `payload` which is a placeholder for an actual message that shall be transmitted over the channel. This variable is not initialized by the sender entity and just received (and never modified) by the receiver entity.

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 `Agent`), but will instead use only their pseudonyms (in place of their real agent names).

Table 1 shows the notations (and writing styles) that ASLan v.2 supports: facts in OO-style notation as well as the annotated channel notation. Here, we write `*` for the bullet annotations of [4], and `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.

Table 1: Notations supported by ASLan v.2

OO 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 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`

3.5 Statements

Statements may be the usual assignments, loops and branches, but also assertions that induce goals to be checked at the current position in the control flow, the generation of fresh values and entities, the transmission of messages, the generation or retraction of facts, a non-deterministic selection among whichever guards are fulfilled (which blocks as long as no guard is fulfilled), and compressed sections that are executed atomically.

Entity generation may instantiate only direct sub-entities, such that static and dynamic scoping coincide. Symbolic sessions are a convenient shorthand to generate an unspecified number of entity instances each of which defines a service session; the agents playing the roles 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.

3.6 Names

In the syntax of ASLan v.2 that we will define shortly, we will adopt the following conventions. 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 or underscore "`_`" or quote "`'`".

In guards, variable names may be preceded by "`?`" to denote that these variables are assigned by pattern matching. In case there are multiple occurrences of the same variable `V` preceded by "`?`" in the same guard, the guard will only be fulfilled if all the relevant patters agree on the value to be assigned to `V`. All other occurrences of `V` without a "`?`" in front will use the new value assigned. For situations where the pattern-matched value is arbitrary and does not need to be stored, one may simply write "`?`" instead of a variable.

Comments in ASLan, as well as in the grammar given below, start with a "`\%`" symbol and extend until the end of the line.

3.7 Extended Backus-Naur Form

The context-free aspects of the ASLan v.2 grammar are defined using an extended BNF, 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.

```
EntityDecl  ::= ComprDecl? "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*

Var         ::= UpperName
Vars        ::= (Var #",")+
Params      ::= "(" ((Vars ":" Type) #",")+ ")"

ComprDecl   ::= "compressed" ( "{" ( LowerName #"," )+ "}" )?

Imports     ::= "import" (UpperName #",")+

Decls       ::= TypeDecls | SymbolDecls | MacroDecls | ClauseDecls
              | EqualityDecls

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)+ ")"                % tuple compound
              | Type ("." Type)+                        % concatenation
              | "(" Type ")"
Types       ::= (Type #",")+

SymbolDecls ::= "symbols" (SymbolDecl ";")+
SymbolDecl  ::= Vars ":" Type
              | (LowerName #",")+ ":" Type       % constants, only in root
              | 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 #"&")+

EqualityDecls ::= "equalities" (EqualityDecl ";")+  % only allowed in root
EqualityDecl  ::= Term "=" Term

Term        ::= LowerName
              | Var
              | "?"Var  %% only allowed within guards
              | "?"     %% only allowed within guards
              | FuncApp
              | "[" Term ("]" | ("]_" Term))  % pseudonym
              | Term ("." Term)+              % message concatenation
              | "(" Term "," Terms ")"        % tuple
              | "{" Terms? "}"  % set literal, only in variable assignment
              | "(" Term ")"
Terms       ::= (Term #",")+
FuncApp     ::= (Term "->")? LowerName ("(" Terms ")")? % not transmission
Body        ::= "body" Stmt

Guard       ::= FuncApp
              | (Term "->")? "receive" "(" Terms ")" ("over" Term)?
        % where "Actor ->" is optional
              | Term "*"? ("->" | "=>") "*"? Term ":" Term  % annotated channels
              | "!" Guard
              | Term "=" Term
              | Guard "&" Guard
              | Guard "|" Guard
              | Guard "=>" Guard
              | "(" Guard ")"
GuardNoRcv  ::= Guard %% any Guard but not containing receive
```

```
Stmt        ::= Assign
              | FreshGen
              | EntityGen
              | SymbEntityGen   % symbolic session
              | (Term "->")? ("send"|"receive") "(" Terms ")" ("over" Term)?
                % where "Actor ->" is optional
                %% sets cannot be used in ‘‘send’’ and ‘‘recv’’ facts
              | Term "*"? ("->" | "=>") "*"? Term ":" Term  % annotated channels
                %% sets cannot be used in annotated channel facts
              | IntroduceFact
              | RetractFact
              | Branch
              | Loop
              | Select
              | Assert
              | "{" (Stmt ";")* "}"
              | "compressed" Stmt

Assign      ::= Var ":=" Term
FreshGen    ::= Var ":=" "fresh()"
EntityGen   ::= "new" Entity
SymbEntityGen ::= "several" (Vars ".")? Entity ("where" GuardNoRcv)?
Entity      ::= UpperName ("(" Terms ")")?
IntroduceFact ::= FuncApp
RetractFact ::= "retract" FuncApp
Branch      ::= "if" "(" GuardNoRcv ")" Stmt ("else" Stmt)? "fi"
Loop        ::= "while" "(" GuardNoRcv ")" Stmt
Select      ::= "select" "{" ("on" Guard ":" Stmt)+ "}"
% Guard may include receive
Assert      ::= "assert" LowerName ":" ("exists" Var+ ".")? GuardNoRcv


GoalDecls   ::= "goals" (GoalDecl ";")+
GoalDecl    ::= LowerName ":" Formula
Formula     ::= FuncApp
              | Term "*"? ("->" | "=>") "*"? Term ":" Term  % annotated channels
              | "!" Formula
              | LTLOp1 "(" Formula ")"
              | LTLOp2 "(" Formula "," Formula ")"
              | Term "=" Term
              | Formula "&" Formula
              | Formula "|" Formula
              | Formula "=>" Formula
              | "forall" Var+ "." Formula
              | "exists" Var+ "." Formula
              | "(" Formula ")"
% neXt | Yesterday | Finally | Once | Globally | Historically
LTLOp1      ::= "X" | "Y" | "F" | "O" | "G" | "H"
% Until | Release | Since
```

```
LTLOp2      ::= "U" | "R" | "S"
```

3.8 Prelude

The syntax of both ASLan v.1 and v.2 gives the user great freedom in declaring new symbols such as functions and facts (cf. the prelude in D2.1). Their meaning may be specified using equations, transition rules, Horn clauses, and/or LTL constraints. In fact, it is an important feature of the ASLan semantics that we do not need built-in symbols with a pre-defined meaning but can express the meaning of all symbols in ASLan itself. For instance, we use in many examples the fact symbol `iknows(msg): fact` that represents that the intruder knows a particular message. We may further use the symbols `crypt(msg,msg):msg` and `scrypt(msg,msg):msg` for asymmetric and symmetric encryption, respectively. ASLan allows us to define the deductions of the intruder in the style of Dolev-Yao by a set of Horn clauses, e.g.

```
iknows(M) :- iknows(crypt(K,M)).iknows(inv(K))
iknows(crypt(K,M)) :- iknows(K).iknows(M)
```

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 are defining a *standard prelude file* that contains standard definitions such as the above ones and that is imported by ASLan v.2 specifications. We do not discuss this in detail here, as the prelude is not considered a built-in part of ASLan, i.e. the syntax and semantics is completely independent of the precise definition of the prelude. The semantics section will silently assume the declaration of several standard symbols in the prelude, such as the type symbol `agent`.

4 ASLan v.2 semantics

In this section, we give a procedure for translating ASLan v.2 specifications into ASLan v.1.1 specifications. This procedure therefore defines the semantics of ASLan v.2 in terms of ASLan v.1.1, and will serve as a basis for the implementation of the translation software.

We provide a high-level overview of the translation procedure, which consists of a number of steps that each focus on different aspects of the mapping from the feature-rich, process-based ASLan v.2 into the rewrite-based ASLan v.1.1.

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), equalities (§ 4.5) and security goals (§ 4.13).

Some preliminaries for the translation of code sections are described in in § 4.6 (for control flow) and in § 4.7 (for compression).

Then, the code sections are translated in two sequential phases:

- translation of statements, guards and transmission events (described respectively in § 4.8, § 4.9 and § 4.12), where the rewrite rules for the ASLan v.1.1 specification are generated (although in a temporary form allowing for ASLan v.2 terms), and
- translation of terms (§ 4.10), applied to the rules generated in the first phase, converting ASLan v.2 terms into ASLan v.1.1 ones.

Finally, § 4.11 describes the translation of the encapsulating body section.

4.1 Preprocessing

As a first step, we consider a group of preliminary operations on the input specification, which do not generate ASLan v.1.1 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 allow for an arbitrary nesting of sub-entities, local declarations of elements (e.g. symbols, sub-entities, types, goals, ...) override any homonymous elements of the same sort that were defined in ancestor entities. Because ASLan v.1.1 only supports a flat name space, each overridden element is renamed uniquely (for instance, by prepending the name of the entity it is defined in) and thus is known, for all further translation steps and at the ASLan v.1.1 level, under the new disambiguated name.

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 v.1.1 `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.

Example 1 Consider the following entity declaration for the *Bid Manager* in the *Public Bidding* model

```
entity BidM(Actor, BP: agent) {

  symbols
    M: msg; % a variable
}
```

then the following state predicate is created

```
state_BidM: agent * agent * msg * IDType * SLType -> fact
```

in `section signature` in the translation.

The arguments of the predicate are two agent names (for parameters `Actor` and `BP`), a message (for variable `M`), a unique *instance* `ID` (of type `IDType`) created at instantiation, and finally a step label of type `SLType` (which we leave unspecified as different encodings for different types might be employed).

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, bp, ui, iid, sl_0)
```

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 `ui`, for *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 subentities, we implement this binding as:

- We utilize two binary predicates: *child(A,D)* stating that D is a child entity instance of A, and its reflexive-transitive closure *descendant(A,D)* stating that D is A itself of a (direct or indirect) descendant of A.
- *descendant(A,D)* may be implemented via the two Horn clauses

$$\forall A.\ descendant(A, A)$$

and

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

- 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 *child(pid,iid)*.
- 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 v.2 has the built-in types `fact`, `msg`, and `agent` which is a subtype of `msg`, and the parameterized types `A * B` and `A set`.

There are also a set of built-in constants and functions:

- `i` of type `agent` denotes the intruder,

- `defaultPseudonym` of type `(agent,iid): agent` is the default pseudonym of a given agent (and a given instance ID), 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,
- `contains` of type `(A set,A): fact` denotes set membership, and
- `iknows` of type `(msg): fact`.

After having (in the preprocessing phase) renamed all homonymous elements so to have globally unique names, mapping of types and symbols declaration can be done in a straightforward way, mainly requiring only syntactical adjustments.

In particular, declarations of new types must be reflected in the ASLan v.1.1 file in `section typeSymbols`. All types, including compound types and parameterized types, are passed unchanged to the ASLan v.1.1 level. 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 v.1.1 file. Parameter declarations for the entity are handled like variable declarations. Note also that for each declaration of a constant `c` of type `agent` we need add to the initial state the fact `agent(c)`, denoting that `c` belongs to the domain of agents (needed for *symbolic entity instantiation*, see § 4.8.5).

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 more 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 v.1.1 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 (everybody would trust anybody who says something on anything). Adding instead the state fact to the conditions of the clause

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

`Actor` and `BP` are bound by the state fact `state_BidM(Actor, BP, M, IID, SL)` to belong to an instance of the entity. □

4.5 Translation of Equalities

The syntax for equalities in ASLan v.2 closely resembles the syntax for the equalities in ASLan v.1.1, 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 § 4.10) 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`, already declared in `section types` of the translation, 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`.

4.7 Step Compression

The translation we define in this document produces for each entity transition rules expressing progress at statement-level granularity. For instance, the three lines

```
Actor->receive(?A, A);
N := fresh();
Actor->send(A, N);
```

would get translated into three individual transitions:

```
state_responder(Actor, ui, ui, SL,IID).iknows(A)
  =>
state_responder(Actor, A, ui, succ(SL),IID)

state_responder(Actor, A, ui, succ(SL),IID)
  =[exists N]=>
state_responder(Actor, A, N, succ(succ(SL)),IID)

state_responder(Actor, A, N, succ(succ(SL)),IID)
  =>
state_responder(Actor, A, N, succ(succ(succ(SL))),IID).iknows(N)
```

After each such progress step, other processes or the intruder may make progress. This gives rise to the following issues:

- Race conditions may appear in case of shared knowledge (e.g. all instances of a service running on the same server, accessing its database). If we do not want to explicitly specify mutual exclusion algorithms

within the model to prevent race conditions, we thus need to be able to specify that certain sequences of transitions are *atomic*. This is of course an abstraction step that the specifier may choose to use, at the risk of losing certain behaviors of the model.

- On the practical side, even for relatively small processes, this fine-grained model produces a large set of interleavings even for a few processes. Specifications can thus easily get infeasible for the validation tools. Note that in many cases these interleavings are irrelevant because they concern internal computations of a service and the relative ordering with other internal computations is not really relevant. On the other hand, it is almost impossible to determine which interleavings could give rise to new attacks on the tool side.

For this reason, we have introduced the construct `compressed {s}` for statements `s`. The idea is to define that all steps of `s` are performed as a single transition. This is similar to *atomic* blocks in other languages for distributed systems: nothing else can happen before the atomic or compressed section is finished. In contrast to atomic definitions, however, we do not regard this as individual transitions (with a restriction on the allowed traces). Instead, we *compress* the respective actions into one transition, e.g. in the above example:

```
state_responder(Actor, ui, ui, SL,IID).iknows(A)
  =[exists N]=>
state_responder(Actor, A, N, succ(SL),IID).iknows(N)
```

We expect that merely for efficiency, such compressions are necessary for many sections of statements, which can lead to cumbersome, hard-to-read specifications. Therefore, we also allow for a different, more global way of specifying compression: we allow the specifier to write `compressed {f} entity` for an entire entity specification with an optional argument `{f}` which is a set of fact symbols. If the optional set is not given, the default is the singleton set `{receive}`. The meaning is that all transitions within the entity (but not sub-entities) are compressed up to, but not including, any introduction of a fact whose fact symbol is included in the set `{f}`. This corresponds to declaring that everything following such an event `e` from `{f}` up to the next one is considered as an internal computation and shall be an immediate consequence of `e`. 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 that we have introduced in our update of ASLan v.1 (see § 2.3): 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 v.2 to v.1.1, 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, ui, ui, SL,IID).iknows(A)
  =>
state!_responder(Actor, A, ui, succ(SL),IID)

state!_responder(Actor, A, ui, succ(SL),IID)
  =[exists N]=>
state!_responder(Actor, A, N, succ(succ(SL)),IID)

state!_responder(Actor, A, N, succ(succ(SL)),IID)
  =>
state_responder(Actor, A, N, succ(succ(succ(SL))),IID).iknows(N)
```

According to the ASLan v.1.1 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, ui, ui, SL,IID).iknows(A)
  =[exists N]=>
state_responder(Actor, A, N, succ(succ(succ(SL))),IID).iknows(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 v.2 to v.1.1.

4.8 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 out-

put. 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)
     - retract     : 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)
     - compressed  : Compressed(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`.

4.8.1 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.

4.8.2 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(publicKey(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,BP,M,IID,sl)
  =>
state_BidM(Actor,BP,crypt(publicKey(BP),m),IID,succ(sl))
```

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,BP,M,IID,sl).state_Env(..,E,..)
  =>
state_BidM(Actor,BP,M,IID,succ(sl)).
  state_Env(..,crypt(publicKey(BP),m),..)
```

□

4.8.3 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 Term

  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 `msg` 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,BP,M,IID,sl)
  =[exists M_1]=>
state_BidM(Actor,BP,M_1,IID,succ(sl))
```

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. □

4.8.4 Entity instantiation

```
EntityGen(stmt, rest, sl, return_sl) {

  stmt = "new" entity "(" t_1 "," .. "," t_n ")"
```

```
  IID = fresh variable

  declare IID of type IDType 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 "ui"

  add
      LHS =[exists IID ]=> RHS
  to "section rules" in the translation

  ParseCode(rest, succ(sl), return_sl)
}
```

Conceptually, instantiating an entity corresponds to creating a new state fact for the instance, running from this point on in parallel to all other entity instances in the system.

To this end, we create a rule that adds to the current state the state fact for the new instance, with a fresh instance ID to make it distinct from any other already in the system and step label `sl_0` from which execution of its body can start. Its parameters `p_1,..,p_n` are assigned the terms passed, namely `t_1,..,t_n`, while its internal variables `v_1,..,v_m` are assigned the value `ui` standing for *uninitialized*.

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(sle,..)
  =[exists IID]=>
state_Env(succ(sle),..).state_BidM(bm,bp,ui,IID,sl_0)
```

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

4.8.5 Symbolic entity instantiation

```
SymbEntityGen(stmt, rest, sl, return_sl) {

  stmt = "several" A_1 .. A_m "."
           entity "(" t_1 "," .. "," t_n ")" "where" Guard

  create, if not present, the following rules
      =[exists A]=> agent(A)
      =[exists A]=> agent(A).honest(A)
  to "section rules" in the translation

  g_1..g_n = positiveGuards(Guard)

  apply adaptGuard to g_1..g_k

  IID = fresh variable

  declare IID of type IDType 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 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 "ui"

    for (j from 1 to m) {
      LHS = LHS.agent(V_j)
      RHS = RHS.agent(V_j)
    }

    add
        LHS.g_i =[exists IID ]=> RHS.renewPositiveFactsIn(g_i)
    to "section rules" in the translation
  }

  LHS1 = state fact for this entity, with step label sl
  RHS1 = state fact for this entity, with step label succ(sl)

  add
      LHS1 "=>" RHS1
  to "section rules" in the translation

  ParseCode(rest, succ(sl), return_sl)
}
```

A *symbolic* entity instantiation consists in the instantiation of an unbounded but finite number of instances of an entity. Its purpose is modelling a whole class of instances, i.e. for all possible substitutions of non-ground agents `A_1,..,A_m` with ground agents taken from the domain.

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

```
=[exists A]=> agent(A)
=[exists A]=> agent(A).honest(A)
```

that generate fresh constants for which the predicate `agent` holds.[4]

Then, the actual symbolic instantiation is translated in two rules, one looping on the current point of execution and generating a new instance of the entity at every step, the other just leaving this loop. Each instantiation will contain the non-ground agents `A_1,..,A_m`, but these will be bound to

[4]The predicate `honest` is needed later on for verification purposes.

concrete agents by the predicates `agent(A_1)..agent(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 § 4.8.9.

4.8.6 Transmission statements

```
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)
  fi
}
```

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 as if it appeared in `select { on R: {}}` (see § 4.8.11 for details).

A `send` statement is translated at first like a fact introduction (see § 4.8.7 for details).

Note that transmission events will be further translated as defined in § 4.12.

4.8.7 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)`.

4.8.8 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).

4.8.9 Branch

```
Branch(stmt, rest, sl, return_sl) {

  stmt = "if" Guard "then" LeftStmt ("else" RightStmt)? "fi"

  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 v.1.1 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)
fi
```

We will then create one rule leading to the first branch if `p(Actor)` is satisfied

```
state_BidM(Actor,BP,M,IID,sl).p(Actor)
  =>
state_BidM(Actor,BP,M,IID,branch(sl, 0)).p(Actor)
```

and another leading to the second branch if instead `p(Actor)` is not satisfied

```
state_BidM(Actor,BP,M,IID,sl).not(p(Actor))
  =>
state_BidM(Actor,BP,M,IID,branch(sl, 1))
```

Parsing the two branches separately will give rise to the following rules

```
state_BidM(Actor,BP,M,IID,branch(sl, 0))
  =>
state_BidM(Actor,BP,M,IID,succ(branch(sl, 0))).q(Actor)
state_BidM(Actor,BP,M,IID,succ(branch(sl, 0)))
  =>
state_BidM(Actor,BP,M,IID,succ(sl))

state_BidM(Actor,BP,M,IID,branch(sl, 1))
  =>
state_BidM(Actor,BP,M,IID,succ(branch(sl, 1))).q'(Actor)
state_BidM(Actor,BP,M,IID,succ(branch(sl, 1)))
  =>
state_BidM(Actor,BP,M,IID,succ(sl))
```

□

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)
fi
```

Here the DNF obtained is

$$(p \wedge not(s)) \vee (not(r) \wedge not(s))$$

Therefore the obtained rules will be

```
state_BidM(Actor,BP,M,IID,sl).p.not(s)
  =>
state_BidM(Actor,BP,M,IID,branch(sl, 0)).q(Actor).p
state_BidM(Actor,BP,M,IID,sl).not(r).not(s)
  =>
state_BidM(Actor,BP,M,IID,branch(sl, 0)).q(Actor)
```

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,BP,M,IID,sl).not(p).r
  =>
state_BidM(Actor,BP,M,IID,branch(sl, 1)).q'(Actor).r
state_BidM(Actor,BP,M,IID,sl).s
  =>
state_BidM(Actor,BP,M,IID,branch(sl, 1)).q'(Actor).s
```

□

Note, finally, that, before any actual translation of the statement is put in place, the function `adaptGuard`, which we will present in detail in § 4.9, is applied to each clause of the guard's DNF to convert all logical connectives into ASLan v.1.1 equivalent predicates. This approach will be applied to all statements using guards.

4.8.10 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,BP,M,IID,sl).p
  =>
state_BidM(Actor,BP,M,IID,branch(sl, 0)).p
```

and another leading outside of the loop if instead `p` is not satisfied

```
state_BidM(Actor,BP,M,IID,sl).not(p)
  =>
state_BidM(Actor,BP,M,IID,succ(sl))
```

Parsing the inner statements of the loop, we will obtain the following rules

```
state_BidM(Actor,BP,M,IID,branch(sl, 0))
  =>
state_BidM(Actor,BP,M,IID,succ(branch(sl, 0))).q
state_BidM(Actor,BP,M,IID,succ(branch(sl, 0)))
  =>
state_BidM(Actor,BP,M,IID,sl)
```

□

4.8.11 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, 0).
            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 guards in a `select` statement typically contain receive conditions, which are translated with the `adaptGuard` function as described in § 4.9.

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,BP,M,IID,sl).p_1
  =>
state_BidM(Actor,BP,M,IID,branch(sl, 1)).p_1.q_1
state_BidM(Actor,BP,M,IID,branch(sl, 1))
  =>
state_BidM(Actor,BP,M,IID,succ(sl))
...
state_BidM(Actor,BP,M,IID,sl).p_n
  =>
state_BidM(Actor,BP,M,IID,branch(sl, n)).p_n.q_n
state_BidM(Actor,BP,M,IID,branch(sl, n))
  =>
state_BidM(Actor,BP,M,IID,succ(sl))
```

□

4.8.12 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 type agreeing with
          W_1,..,W_m plus the entity instance ID (IID)

  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)

  add
      LHS "=>" RHS
  to "section rules" in the translation

  add
      G(check(W_1,..,W_m,IID).S => exists V_1,..,V_n. Guard)
  to the "section goals" in the translation, where S is the
  state fact for this entity, with step label succ(s1).

  ParseCode(rest, 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 `goal 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 in the state where this predicate appears, the `Guard` holds. The `exists V_1,..,V_n. Guard` part in the above definition is further translated as given in § 4.13.

4.9 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 v.1.1 predicates, as specified in Table 2. Note that the | (disjunction) and `=>` (implication) operators are not included since there is no corresponding predicate in ASLan v.1.1, 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 2. Note that the translation of sub-terms (i.e. 'atoms' from the logical point of view) is specified in § 4.10.

4.10 Translation of Terms

In § 4.8, we presented a procedure for translating ASLan v.2 statements into ASLan v.1.1 transitions, but we did not apply any translation of terms. There are two reasons behind this:

Table 2: Substitutions done by the `adaptGuard` function

ASLan v.2 operator	ASLan v.1.1 predicate
`!G`	`not(G)`
`G_1 & G_2`	`G_1.G_2`
`T_1 = T_2`	`equal(T_1,T_2)`

- 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 v.2 terms can be easily replaced by equivalent terms in ASLan v.1.1 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 v.1.1 and ASLan v.2 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, containing the element terms `T_1,T_2,..,T_n`. Note that this case appears only in an assignment of a variable `V` of type `A set` where `A` is the element type shared by all `T_1,T_2,..,T_n`. We create then a fresh constant `sc` of the same type `A set`. Then we substitute `T` by `sc`, and add to the `RHS` of R the facts `contains(sc,T_1)`,..,`contains(sc,T_n)`. Ultimately we proceed recursively on each of `T_1,T_2,..,T_n`.
- `T = ?`, i.e. `T` pattern-matches anything, which is allowed only in guards, i.e. on the `LHS` of R. In this case, we just create a fresh variable name `V'` and replace `T` by `V'`.
- `T = ?V`, i.e. `T` represents an assignment of `V` by pattern-matching, which is allowed only in guards, i.e. on the `LHS` of R. We add the state fact of the owner of `V` to both the `LHS` and `RHS` of R where we replace the variable name `V` in the left state fact by a fresh variable name `V'`. For example, `if p(?V) then q(V) fi` 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 § 4.12.

4.11 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.

4.12 Translation of Transmission Events

As described in detail in D3.3, we have formalized different models of communication and definitions of channels as assumptions or goals, respectively. In particular, for channels as assumptions we have three models:

- 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 [4,14] 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.

We will discuss the translation of channels when used as goals in the next subsection. 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 v.1.1 in the three models is as follows:

- CCM:

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

- 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.

4.13 Translation of Goals

In the case of goals, in a similar way as for guards, we need to translate the LTL operators in ASLan v.1.1 predicates, according to Table 3.

After doing so, the terms contained in the goal formulae must be converted, by means of a slight variant of the procedure introduced in § 4.10 (which we omit).

4.13.1 Channels as 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. [7, 10, 13].

In order to formulate the goals in a service-independent way, we use, in the translation, a set of *auxiliary events* (modeled as ASLan v.1.1 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 [11]). More specifically, we consider the following kinds of fact symbols:

Table 3: Translation of goal operators

Operator	ASLan v.2 connective	ASLan v.1.1 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(f)`
Once	`O(f)`	`O(f)`
Globally	`G(f)`	`G(f)`
Historically	`H(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)`

```
witness(agent,agent,SID,msg)
request(agent,agent,SID,msg)
whisper(agent,msg)
hear(agent,msg)
```

where `SID` is an appropriate type 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 § 3.4) implies the creation of such events within the execution of a particular entity; in particular the translation to ASLan v.1 decorates appropriate transition rules with these facts in a way that we describe now.

For ease of description, consider the following example:

```
entity Session (A, B: agent) {
  symbols Payload: payload;
  entity Alice(Actor, B: agent) {
    ...
    Actor.send(B,...Payload...)
    ...
  }
```

```
  entity Bob(Actor, A: agent) {
    symbols Payload: payload;
    ...
    Actor.receive(A,...?Payload...)
    ...
  }
  body {
    new Alice(A,B);
    new Bob  (B,A);
  }
  goals
    g1: A *->* B: Payload;
}
```

The translator shall decorate this with the following events:

```
entity Session (A, B: agent) {
  symbols Payload: payload;
  entity Alice(Actor, B: agent) {
    witness(Actor,B,g1,Payload)
    whisper(B,Payload)
    ...
    Actor.send(B,...Payload...)
    ...
  }
  entity Bob(Actor, A: agent) {
    symbols Payload: payload;
    ...
    Actor.receive(A,...?Payload...)
    ...
    request(A,Actor,g1,Payload)
    hear(Actor,Payload)
  }
  body {
    new Alice(A,B);
    new Bob  (B,A);
  }
  goals
    g1: A *->* B: Payload;
}
```

where `witness` and `request` are generated through the authenticity part of the goal, and `whisper` and `hear` result from the confidentiality part. Thus,

the respective events are inserted as early as possible in the sender entity, and as late as possible in the receiver role. When using a pseudonym `P`, then sender or receiver name will simply be replaced by that in the goal, e.g. `witness(P,B,g1,Payload)`.

We can now define the goals of confidentiality and authenticity as LTL formulae over the given facts without referring to the particularities of the specified entities such as the message formats.

Consider the following formulae.

```
forall A,B,S,M. G
  (request(A,B,S,M) => (O (witness(A,B,S,M)) ||
                          (dishonest(A) & iknows(M))))

forall B,S,M. G
  ((whisper(B,S,M) & iknows(M)) => dishonest(B))

forall B,S,M. G
  (hear(B,S,M) => (O (whisper(B,S,M)) || iknows(M)))
```

The first is a standard authentication goal (non-injective agreement [10]) with the additional condition that, if the sender is dishonest, then the intruder must know this message. This requirement is necessary for the compositionality property of our channel definition (otherwise the channels are too weak to achieve compositionality) as explained in more detail in [14]. The second formula gives the standard secrecy goal (if the intruder knows a secret of an honest agent for another agent `B`, then `B` must be a dishonest agent or it is a violation). Finally, the last goal is a similar extension to secrecy as the above mentioned extension to channels: in case an honest agent receives a secret, it was indeed sent either by some honest or by some dishonest agent, and in the latter case, the intruder must know it.

This concludes the procedure for translating ASLan v.2 specifications into ASLan v.1.1 specifications. We now consider some concrete examples.

5 ASLan v.2 examples

In this section, we assess ASLan v.2 with its new features against a selection of problem cases taken from Deliverable D5.1 [5]; see also Deliverable D2.1 [3], in which we described the requirements for ASLan that we extracted from the case studies and the corresponding features of ASLan v.1. In order to be able to formally model static service and policy composition, ASLan v.2 shows a number of advantages with respect to v.1 such as:

- Control flow constructs (e.g. `while` and `if`) allow for better readability and conciseness of the specifications, and make the specification easier for modelers who are already familiar with programming languages.
- Modularity can be seen in all the examples by the use of entities. Each entity is specified separately and can then be composed with others, in the spirit of object-oriented programming. This allows the specifier to localize policies in each entity by making it clear, e.g., who is responsible to grant or deny certain authorization requests as well as the various trust relationships between entities. All these features are particularly evident in the Car Registration scenario in § 5.1, but they are also illustrated in the case studies Loan Origination Process § 5.2 and Digital Contract Signing in § 5.3.
- Furthermore, ASLan v.2 provides easier ways to specify communication and service compositionality by using a suitable, intuitive notation for channels that are used both as service assumptions and as service goals. This feature is illustrated in the Identity Mixer case study in § 5.4.

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 (Requirements for modeling and ASLan v.1 [3, §4.3]).

Here, we present the same example in the same simplified version but expressed in the new ASLan v.2, to demonstrate the greatly enhanced readability and modularity of the ASLan v.2 model in comparison with v.1.

When comparing the model given in [3, §4.3] specified in ASLan v.1 with the one given below specified in ASLan v.2, 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 ASLan 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 v.1.

```
%% Specification of the Car Registration Office in D5.1
entity Environment { %% contains shared (global) declarations
                     %% and all system components as sub-entities

  import DKAL %% see DKAL module above

  %% ------------------------------------------------------------
  %% Global declarations
  %% ------------------------------------------------------------

  types

    doc      < message;
    decision < message;
    action   < message;
    role     < message;

  symbols

    % Identifier for the request sent by mike
    % (cf. scenario given in Deliverable D5.1)
    doc5_1: doc;

    % Flags saying that a request has been refused or accepted
```

```
  refused, accepted: decision;

  % Possible actions about a document with respect to the
  % central repository: reading or storing
  readDoc,          % unused in this example
  storeDoc: action;

  % 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;

macros

    signedRoleCert(Empl,Role) =
      sign(inv(public_key(theCA)),rolecert(Empl,Role));

symbols

  % flags that a citizen sent a certain request document
  citizen_sent (doc): info;

  % Constants identifying the central repository and the
  % certification authority of the car registration office
  centrRep, theCA: agent;

  % Constants identifying persons in scenario in D5.1:
  % melinda (head), peter (employee), and mike (citizen)
  melinda, peter, mike: agent;


%% ------------------------------------------------------------
%% individual entities
%% ------------------------------------------------------------

entity CA(Actor: agent) {

  body {

    % The certification authority of the car registration
```

```
      % office has generated a certificate about the role of
      % Melinda (i.e. head of the car registration office)
      send(peter,signedRoleCert(melinda,head));
      % Generally, should be broadcast to all employees

      % The certification authority of the car registration
      % office has generated a certificate about the role of
      % Peter (i.e. employee of the car registration office)
      send(peter,signedRoleCert(peter,employee));
    }
}

entity Citizen(Actor: agent) {

  symbols

    Empl: agent;
    Decision: decision;

  body {

    send(peter,sign(inv(public_key(Actor)),doc5_1.Actor));
    Actor->saysTo0(centrRep,citizen_sent(doc5_1));

    receive(peter,?Empl.?Decision.doc5_1); % better if signed
    % any reaction to the decision
  }
}

% Registration office head %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Note that we cannot express (but actually do not need
% here) the fact that Head is a special form of Employee.
entity Head(Actor: agent, Empls: agent set) {

  clauses

    % 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 one of
    % his/her employees, he/she is willing to share this
    % information with anyone, in particular with Peter
    GenerateCert(Actor,Empl,AnyOne):
      Actor->saysTo0(AnyOne,Empl->canStoreDoc) :-
      Actor->knows0(Empl->canStoreDoc);

    % The head of the car registration office, e.g. Melinda,
    % decides that all his/her employees, e.g. Peter,
    % have enough experience to store
    % accepted requests in the central repository
```

```
    AllEmplsCanStoreDoc: forall Empl.
      Actor->knows0(Empl->canStoreDoc) :- Empls->contains(Empl);
}

% (Simple) employee %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
entity Employee(Actor: agent) {

  symbols

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

  clauses

    % An employee getting a certificate from the certification
    % authority about some agent in the car registration office
    % is willing to tell this to anyone, in particular centrRep
    Cert1(Actor,Cert,AnyOne):
      Actor->saysTo(AnyOne,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 anyone, in particular to centrRep
    Cert2(Actor,Head,Cert,AnyOne):
      Actor->saysTo(AnyOne,Head->said0(Cert)) :-
      Actor->knows(Head->said0(Cert)) &
      Actor->knows(theCA->said0(Head->hasRole(head)));

  symbols

    Empl1, Citizen: agent;
    Key: inv(public_key);
    Signed_Doc, Doc: doc;

  body {

    % The contents of the request sent by the citizen
    % correctly supports his/her request
    Actor->knows0(isok(doc5_1));

    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 receive(theCA,signedRoleCert(?Empl1,?Role)) {
```

```
        Actor->knows0(theCA->said0(Empl1->hasRole(Role)));
      }

      on receive(Citizen,?Signed_Doc)
         & Signed_Doc = sign(?Key,?Doc.?Citizen) {
        if(Key = inv(public_key(Citizen)))
          if(Actor->knows(isok(Doc))) {
            % The employee accepts a request Doc by a Citizen
            % if the request is correctly signed
            % and all criteria are met.
            % He causes two actions by the employee who has
            % processed the request: sending an acknowledgement
            % to the citizen that the request has been accepted
            % and asking the central repository to store the
            % processed request
            send(centrRep,sign(inv(public_key(Actor)),
                       Actor.accepted.Signed_Doc).storeDoc);
            % typically, at this point the employee should
            % check for feedback from centrRep and if negative,
            % it should not claim acceptance to the Citizen
            send(Citizen,Actor.accepted.Doc); % better if signed
          }
          else
            % The employee refuses a request Doc by a Citizen
            % if the request is correcly signed but some other
            % criterion (abstracted away in this specification)
            % is not met.
            % Rejection is acknowledged to the Citizen.
            send(Citizen,Actor.refused.Doc) % better if signed
          fi
        else
        % The employee refuses a request Doc by a Citizen if
        % the request is not signed by the Citizen sending it.
        % Rejection is acknowledged to the Citizen.
          send(Citizen,Actor.refused.Doc) % better if signed
        fi
      }
    }
  }
  }
}

% Central Repository %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
entity CentralRepository(Actor: agent) {

  clauses

    % unused: The central repository trusts the
    % certification authority of the car registration office
```

```
    centrRepTrustCA(AnyThing):
     Actor->trusts0(theCA,AnyThing);

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

    % 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(AnyOne,AnyThing):
      Actor->trusts(AnyOne,Head->said0(AnyThing)) :-
      Actor->knows(theCA->said0(Head->hasRole(head)));

    % The central repository trusts the head of a car
    % registration office when this 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(Head,Empl):
      Actor->trusts0(Head,Empl->canStoreDoc) :-
      Actor->knows(theCA->said0(Head->hasRole(head))) &
      Actor->knows(theCA->said0(Empl->hasRole(employee));

  symbols

    DocDB: message set;
    Signed_Doc: message;
    Empl: agent;

  body {

    DocDB := {};

    while(true) {
    select {
      % Upon reception of a request to store a document
      % in the data base, check whether the employee asking
      % for this to be done has the right to do this.
      % If so, the request is added to the data base
      on receive(Empl,?Signed_Doc.storeDoc) &
          Signed_Doc = sign(inv(public_key(?Empl)),
                            Empl.accepted.?Doc) &
          Actor->knows(Empl->canStoreDoc) {
        assert stored_req_is_double_signed_and_sent_by_citizen:
```

```
            exists Cit Req.
             Signed_Doc = sign(inv(public_key(Empl)),Empl.accepted.
                          sign(inv(public_key(Cit)),Req.Cit)) &
               Actor->knows(Cit->said0(citizen_sent(Req)));
           DocDB->insert(Signed_Doc);
         }
       }
       }
    }
  }

  body { % initialization of the overall system

    new CA(theCA);
    new CentralRepository(centrRep);

    new Head(melinda,{peter});
    new Employee(peter);

    % A citizen, called Mike, sends a request to the car
    % registration office. The request which will be handled
    % in the office by the employee Peter.
    new Citizen(mike);
  }

  goals

    %% add any further goals here as LTL formulas

} % end Environment
```

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
```

5.2 Loan Origination Process

The loan origination business process (LOP) has also been described in Deliverable D5.1 [5] and an excerpt of the formalization was presented in Deliverable D2.1 [3].

Thanks to the better modularity, readability and conciseness of ASLan v.2, we can present here a more complete specification of the LOP where the behavior of all the entities (bank process engine, task engine, etc) is defined, and show that the aspects we wanted to model, like the static separation of duty (SSOD) check, are still expressible in the new language. Certain features such as communication have been abstracted away, as they are irrelevant for the security aspects we want to show here. All these important and security relevant features could be easily added (and will be added in the future) by moving to a different abstraction level better capturing the real system, but featuring a bigger search space.

We recall that the LOP describes a bank's evaluation of a customer's request for a loan. The bank comprises two main entities: the process engine which listens for customers' requests and initiates new LOP workflows, and a task engine which manages access control and task execution.

The customer and the process engine are modeled here in a simple way, where the customer just sends a request for a loan, and the bank's process engine receives and processes it. But other customer's requests could be easily added to trigger additional processes at the bank side (e.g. opening a bank account).

```
%% A customer can send a loan request
%% to a Bank Process Engine
entity Customer(Actor,BankPE : agent)  {
  body
    send(BankPE,loan_request);
}

%% The bank process engine listens for loan request
%% from customers and initiates LOP workflows
entity BankPE(Actor : agent) {

  types
    BPI : nat;
    C : agent;

  body
    while(true) {
      %% wait for a customer's request
      receive(?C,loan_request);
      %% generate a new business process instance (BPI)
```

```
      BPI := fresh();
      %% initiate a new LOP worflow associated to BPI
      done(start,BPI);
    }

}
```

For the sake of simplicity, we model here a simplified version of the workflow (with respect to D5.1 [5]), where we focus on the internal executions at the bank (the interaction with the credit bureau is not considered). But this is sufficient to show how a business process can be specified in ASLan v.2. One key ingredient is to capture standard control workflow artifacts such as fork, join, choice or merge. For instance, this can be done by using two facts: `done` and `ready`. A task is marked as `done` when it has been executed, and as `ready` when it is ready to be executed. For example, the execution of `inputCustData` enables two tasks, corresponding to a fork. And the `approval` task depends on two tasks, like a join. We could easily model choice and merge as well.

```
%% simplified LOP workflow
entity LOP {

  body

    while(true) {
     compressed {

      select {
        on done(start,?BPI) {
          ready(inputCustData,BPI);
        }
        %% fork
        on done(inputCustData,?BPI) {
          ready(intRating,BPI);
          ready(extRating,BPI);
        }
        %% join
        on done(intRating,?BPI) & done(extRating,BPI) {
          ready(approval,BPI);
        }
        on done(approval,?BPI) {
          ready(signForm,BPI);
        }
      }

     }
    }
}
```

In specification languages of business processes such as BPMN, it is possible to specify access control for business process execution. There are two notions: potential and excluded owners. The former defines the users or roles entitled to execute a given task. The latter defines the users or roles not allowed to execute a given task. The excluded owner has priority over the potential owner.

This is all modeled in the task engine entity, where the local policy describes the access control by means of Horn clauses. This policy checks whether a user is able to execute a task or not, based on the `hasRole`, potential owner (`poto`) and excluded owner (`exo`) relations. The SSOD check is also done here, preventing a user to execute two tasks under a SSOD constraint.

The delegation and execution of tasks are specified in the body, as guarded statements whose firing depends from the evaluation of the clauses.

```
%% The task engine manages the access control
%% and execution of tasks
entity TaskEngine (Actor: agent) {

  symbols

    %% acces control
    forbidden(user,task_name) : fact;
    granted(user,role,task_name) : fact;

    U,U1,U2 : user;
    R,R1,R2 : role;
    T,T1,T2 : task;
    TI,BPI : nat;

  clauses

    %% if there is a SSOD constraint for T1 and T2
    %% and U executed T2
    %% then U is not allowed to executed T1
    forbid_ssod(U,T1,R,T2,TI,BPI) :
      forbidden(U,T1) :- ssod(T1,T2),
                         executed(U,R,task(T2,TI,BPI))

    %% forbid access to excluded owners (users)
    forbid_excluded(U,T) :
      forbidden(U,T) :- exo(U,T)

    %% forbid access to excluded owners (roles)
    forbid_excluded(U,T,R) :
      forbidden(U,T) :- hasRole(U,R), exo(R,T))
```

```
    %% grant access to potential owners (users)
    grant(U,R,T,BPI) :
      granted(U,R,T) :- ready(T,BPI), hasRole(U,R),
                        poto(U,T).

    %% grant access to potential owners (roles)
    grant(U,R,T,BPI) :
      granted(U,R,T) :- ready(T,BPI), hasRole(U,R),
                        poto(R,T).

  body

    while(true) {
     compressed {

      select {
        %% delegation
        %% if U1 is allowed to execute T1 with role R1,
        %% he can delegate T1 to another user U2
        on hasRole(?U2,?R2) & granted(?U1,?R1,?T1) {
          granted(U2,R1,T1)
        }

        %% execution
        %% an user can execute a task which is ready,
        %% if he is granted and not forbidden by the AC
        on granted(?U,?R,?T) & !forbidden(U,T)
                             & ready(T,?BPI) {
          TI := fresh();
          executed(U,R,task(T,TI,BPI))
          done(T,BPI);
        }
      }

     }
    }
}
```

In the top level entity `Environment`, we define a specific scenario of the business process. We define the users, roles and tasks involved in that scenario, the different relations between them (`hasRole`, `ssod`, etc) and we create real instances of the different entities. The `Environment` also comprises all the entities presented previously.

A very simple scenario can be defined as a task engine and a process engine of one bank, one customer of that bank, and the LOP workflow. Of course more complex scenarios could be defined.

```
entity Environment { %% shared (global) entities
```

```
types

  owner; user < owner; role < owner;
  task_name; task;

symbols

  %% ready(TN,BPI) : task TN is ready to be executed
  %% in the BP instance BPI
  ready(task_name,nat) : fact;

  %% done(TN,BPI) : task TN has been executed
  %% in the BP instance BPI
  done(task_name,nat) : fact;

  %% task(TN,TI,BPI) : task name TN + task instance TI
  %%                   + bp instance BPI
  task(task_name,nat,nat) : task;

  %% executed(U,R,T) : user U has executed task T
  %%                   with role R
  executed(user,role,task) : fact;

  %% static separation of duty
  ssod(task_name,task_name) : fact;

  %% potential owners
  poto(owner,task_name) : fact;

  %% excluded owners
  exo(owner,task_name) : fact;

  %% role attribution
  hasRole(user,role) : fact;

  pierPaolo,marco,davide : user;
  preclerk,postclerk,manager : role;
  start,inputCustData,intRating
       ,extRating,signForm : task_name;
  loan_request : msg;

  TE,BPE,C : agent;

body

  ssod(inputCustData,approval);

  poto(preclerk,inputCustData);
  poto(postclerk,intRating);
```

```
    poto(manager,approval);
    poto(manager,signForm);

    hasRole(pierPaolo,preclerk);
    hasRole(pierPaolo,postclerk);
    hasRole(pierSilvio,manager);
    hasRole(marco,manager);
    hasRole(davide,preclerk);

    TE := fresh();
    new TaskEngine(TE);

    BPE := fresh();
    new BankPE(BPE);

    C := fresh();
    new Customer(C,BPE);
    new LOP;

  %% all the sub-entities can be inserted here
```

Now that we have specified the model, we can describe security properties to be checked during analysis. For instance, there might be a compliance rule put in place to avoid fraud, which specifies that sign forms must be signed only by managers. The goal `check_signform_access` is a simple LTL formula to check that.

```
  goals
    check_signform_access:
      G ( executed(U,R,task(signForm,TI,BPI))
          => hasRole(U,manager) );

} % end Environment
```

Notice that the property could actually be violated. We can have an execution where a manager (e.g. `marco`) delegates the task signForm to another user (e.g. `davide`) who is not a manager, but is still able to execute that task. Depending on the context, this violation can be negligible or not. It is up to the modeler to decide and possibly refine his model and/or properties.

5.3 Digital contract signing

The digital contract signing scenario has been described in Deliverable D5.1 (Problem Cases and their Trust and Security Requirements) [5]. For the sake of conciseness, we simplify this scenario as follows.

Two *signers* have secure access to a trusted third party, a *business portal*,

so as to sign a contract. After the three agents have agreed on the terms of the contract (this phase is not modeled and its successful completion is assumed), the business portal receives the required information to write down the contract from the two signers (e.g. their full names and affiliations) and prepares a contract. In order to handle the signatures, the time stamping, and store the (signed) contract, the business portal assumes the availability of a *Signature and Proof management Infrastructure* (SPI) which, in turn, relies on the following two agents:

- the *certification authority* guarantees the identity of the various agents involved in scenario so that the validity of any certificate generated by one of the agents can be established by discovering a suitable chain of certificates (this is much in the spirit of the SPKI/SDSI infrastructure); and

- the *security server* handles the requests to store signed copies of a contract received by the business portal and takes care of checking whether signatures are associated to identity certificates which have not been revoked, i.e. such certificates are not in the (last updated version of the) *Certification Revocation List* (CRL) issued by the certification authority (CRLs are also part of the SPKI/SDSI infrastructure).

The time stamper and the archiver (present in the description of the scenario in [5]) have been abstracted away for the sake of conciseness. However, at the expense of making the specification longer, we see no particular problems in specifying them in ASLan v.2. Another simplifying assumption consists of avoiding to explicitly model private and public keys necessary to support the SPI infrastructure so as to focus on the control flow of each entity involved and on the exchange of messages.

The scenario goes as follows. First, the two signers send the necessary information for building up the contract to the business portal. This creates the contract and asks to the security server to reserve some space in a database to store the signed copies. The security server provides the business portal with a unique identifier of the record of the database where to store that particular contract. At this point, the business portal signals the availability of the contract to both signers who ask to have a copy of the contract. After checking that the contract conforms to what has been negotiated before, they sign it and send the signed copies to the business portal. In turn, the business portal forwards the copies to the security server which checks that the signatures correspond to valid identity certificates by consulting the most recent version of the CRL published by the certification authority. If the checks are successful, the signed copies of the certificates are stored in

the previously identified record in the database. The business portal receives an acknowledgment that the signed copies of the contract have been successfully stored (if the case) and then closes the procedure of the digital contract signing by asking the security server to "seal" the record where the copies have been stored.

We now describe each entity composing the specification of the case study. First of all, we (partially) specify the entity `SPI` for the SPI infrastructure. It contains messages and common functionalities to handle signatures and CRLs. The messages `signature_ok(s)` and `signature_ko(s)` signal that the signature `s` is valid or not, respectively, i.e. it is not or it is in the currently available version of the CRL (of the certification authority); the message `req_new_crl` is used to ask for an updated version of the CRL; and the message `crl_still_ok(a)` signals that the CRL belonging to agent `a` is still valid. The variable `crl_is_expired` signals when the currently available CRL is no more valid and a new one should be computed. The function `sign(c,a)` produces a signed copy of the document `c` by the agent `a`. The function `verify_signature(a,sc,ca,crl)` checks whether the certificates of an agent `a` who has produced the signed version `sc` of a given document (in our case, a contract) are not in the CRL `crl` published by an agent `ca`. Furthermore, `SPI` provides predicates to handle identity and action certificates at the policy level: `name_crt(a_1,a_2)` corresponds to a certificate signed by agent `a_1` about the identity of `a_2`, and `actn_crt(a_1,e,a_2)` corresponds to a certificate signed by agent `a_1` giving `a_2` the authorization to execute a certain action `e`.

```
entity SPI {
  symbols
   %%% Messages
   signature_ok, signature_ko : message;
   crl_still_ok (agent) : message;
   req_new_crl : message;
   %%% Variables and functions
   crl_is_expired : fact;
   sign (info, agent) : info;
   verify_signature (agent,info,agent,info) : fact;
   %%% Certificates
   name_crt (agent, agent) : fact;
   actn_crt (agent, info, agent) : fact;
  ...
}
```

For the sake of conciseness, the specification of the `SPI` entity is incomplete as the meaning of the function and the certificates is not formally defined. We notice that, for the case study under consideration, there is no need to further constrain the meaning of predicate certificates as the mere existence of such

certificates allows agents to perform some actions. If an entity specifying an agent would like to use any of the functionalities supported by the SPI infrastructure, it is sufficient that it `imports SPI`.

We discuss now how to specify the certification authority `CA`. Since we can have many such authorities, the entity is parameterized with respect to an agent identifier `theCA`. At the policy level, `CA` imports the SPI infrastructure so that it can create a certificate about its own identity (such a fact is stored in its policy rules upon instantiation of the entity). At the workflow level, `CA` declares a variable `new_crl` storing the CRL, i.e. a list of certificates whose validity is expired. The (Boolean) variable `crl_is_expired` abstracts away part of the functionalities of the time stamper since sending the value of the CRL currently stored in `new_crl` or sending the message that the previously sent version of the CRL is still valid non-deterministically depends on `crl_is_expired`. Since the content of the CRL is public, there is no need to check the identity of the agent sending in the request.

```
entity CA(Actor : agent) {

  import SPI;

  clauses
   CA_identity : name_crt(Actor, Actor);

  symbols
   new_crl : info;
   Id : agent;

  body {
   while (true) {
    select {
     on (receive(Id,?Id.req_new_crl.Actor)) {
       if (crl_is_expired) {
         send(Id,Actor.new_crl.Id);
       }
       else {
         send(Id,Actor.crl_still_ok(Actor).Id);
        }
       fi
     }
    }
   }
  }
}
```

We are ready to describe the three main entities of the scenario: a signer, the business portal, and the security server. In the following, we assume that only two signers are involved. It is interesting to consider the list of

parameters of the entity `Signer`: `Id` is the identifier of the signer, `BP` is the identifier of the business portal providing the digital contract signing service, `CA` is the identifier of the certification authority signing the identity certificate and handling the CRL, and `ContractId` is the unique identifier used by the business portal and the security server to refer to the contract being signed (this identifier has been given to the contract in the phase of contract negation not modeled here). When creating an instance of the entity `Signer`, all its messages and the values of the variables in the workflow refer to a given contract. In this way, there is no risk of confusion even if the same agent `Id` is involved as a signer in more than one signing procedure as two distinct instances, which differ in the value of `ContractId`, will be created. At the policy level, upon creation of an instance of the entity, the suitable identity certificate is asserted (`signer_identity`). At the workflow level, the variable `InfoContract` stores all the information required to create the contract (e.g. name and affiliation of the signer), `verify_contract` encapsulates the criteria of the signer to evaluate if the produced contract is satisfactory (`contract_ok` stores the result of invoking this function), and `signed_ok` tells if the contract has been signed or not.

```
entity Signer(Actor, BP, CA : agent, ContractId : info) {

  import SPI;

  clauses
   signer_identity : name_crt(CA, Actor);

  symbols
   InfoContract, Contract : info;
   verify_contract (info) : fact;
   contract_ok, signed_ok : fact;

  init
   contract_ok := false;
   signed_ok := false;
   send(BP,Actor.ContractId.InfoContract.BP);

  body {
   while (true) {
    select {
     % upon reception of a message from the business portal
     % of the availability of contract, the signer sends back
     % a request to have a copy
     on (receive(BP,BP.contract_available.Actor) &
         name_crt(CA,BP)) {
       send(BP,Actor.request_contract.BP);
     }
```

```
      % upon reception of a message containing the contract
      % from the business portal, the signer checks if the
      % contract is conformant to its criteria and, if so,
      % signs it and sends back the signed copy to the
      % business portal
      on (receive(BP,BP.?Contract.Actor) &
          name_crt(CA,BP)) {
        contract_ok = verify_contract(Contract);
        if (contract_ok) {
          contract_signature = sign(Contract,Actor);
          signed_ok := true;
          send(BP,Actor.contract_signature.BP);
        }
        fi
      }
     }
    }
   }
}
```

Two remarks are in order. First, at the beginning, the sender is required to forward all the information to create the contract to the business portal. Second, the identity of the sender of the messages is checked by using suitable identity certificates (`name_crt(CA,BP)`).

Now, we consider the business portal `BusinessPortal`. As above, it is interesting to look at the parameters of the entity. Similarly to signers, we have an identifier for the entity itself, two identifiers for two signers (`Sig1` and `Sig2`), the identifier of the certification authority (`CA`), the identifier of the service provider (`SS`), and one for the contract identifier (`ContractId`). In this way, each instance of the business portal can handle separately several contract signing procedures (involving several signers) while possibly using multiple security servers. At the policy level, we find the identity certificate (`BP_identity`). At the workflow level, several variables are declared in order to distinguish between the various steps of the digital contract signing procedure. The possible behaviors of `BusinessPortal` are as follows:

1. upon reception of a message from a signer containing part of the information required to make the contract (identified by `ContractId`), the business portal creates the contract (by invoking the function `create_contract` whose precise semantics is left unspecified as it plays no role in our specification) when it has received the information for the contract from both signers (the variables `contract_available`, `isFirstSigner` and `allSigners` are used to check this); finally, the business portal sends the contract to the security server with the request that a record for it should be created;

2. upon reception of a message from the security server containing the confirmation that the record for the contract has been created, the unique record identifier of the record is stored in a local variable for future use (namely when forwarding signed the copies to the security server for signature checking and storing the signed copies); then, the signers are acknowledged that the contract is ready to be signed and two action certificates are created at the policy level to permit the signers to obtain a copy of the created contract;

3. upon reception of a message from a signer to get a copy of the contract, the business portal sends it under the assumption that a suitable contract to perform such an action is available (see previous point);

4. upon reception of a message from a signer containing its signature for the contract, it stores it in a local variable (`sigA` or `sigB`) for future use, and it forwards it to the security server which is supposed to check its validity (see the entity `SecurityServer` below for more on this point);

5. upon reception of a message from the security server that the signatures for the contract of both signers are valid (to detect this situation, the local variables `sigA_verified` and `sigB_verified` are used), the business portal sends a message to the security server to permanently store the signed copies of the contract (`store(RecordId)`).

Notice that the identity of the sender of all messages is checked by using suitable identity certificates and when the sender asks for obtaining a copy of the newly created message, it is also checked that suitable action certificates have been created.

```
entity BusinessPortal(Actor, Sig1, Sig2, CA, SS : agent,
                      ContractId : info) {
  import SPI;

  clauses
   BP_identity : name_crt(CA, Actor);

  symbols
   % flag to establish if it is the first signer or not
   isFirstSigner : fact;
   % flag to establish if both signers sent infos
   allSigners : fact;
   Id, IdA : agent;
   InfoA, InfoB, Contract : info;
   contract_available : fact;
   % similar to isFirstSigner but for signed contract
   isFirstSigned : fact;
```

```
  % similar to allSigners but for signed contract
  allSigned : fact;
  SigA, SigB : info;
  record_Id : info;
  % flags to establish if signatures have been verified
  sigA_verified, sigB_verified : fact;

 init
  isFirstSigner := true;
  allSigners := false;
  contract_available := false;
  isFirstSigned := true;
  allSigned := false;
  sigA_verified := false;
  sigB_verified := false;

 body {
  while (true) {
   select {
    % (1)
    on (receive(Id,?Id.ContractId.?InfoContract.Actor) &
        ((Id=Sig1) | (Id=Sig2)) & name_crt(CA,Id)) {
      if (isFirstSigner & not(allSigners)) {
        IdA := Id;
        InfoA := InfoContract;
        isFirstSigner := false;
      }
      else {
       if (((Id=Sig1) | (Id=Sig2)) &
                name_crt(CA,Id) & not(Id=IdA) &
                not(isFirstSigner) & not(allSigners)) {
              InfoB := InfoContract;
              allSigners := true;
              Contract := create_contract(ContractId,
                                          InfoA,InfoB);
              send(SS,Actor.Contract.SS);
       }
        else {
              %%% Some error has occurred!
              %%% Take necessary actions to recover/signal this
              %%% error situation.
             }
       fi
      }
      fi
    }
    % (2)
    on (receive(SS,SS.contract_created(?rId).Actor) &
        name_crt(CA,SS)) {
```

```
    contract_available := true;
    record_Id := rId;
    send(Sig1,Actor.contract_available.Sig1);
    send(Sig2,Actor.contract_available.Sig2);
    %%% Adding action certificates for both signers
    actn_crt(Actor,request_contract,Sig1);
    actn_crt(Actor,request_contract,Sig2);
  }
  % (3)
  on (receive(Id,?Id.request_contract.Actor) &
      ((Id=Sig1) | (Id=Sig2)) & name_crt(CA,Id) &
      actn_crt(Actor,request_contract,Id)) {
    if (contract_available) { send(Id,Actor.contract.Id);
    }
    fi
  }
  % (4)
  on (receive(Id,?Id.contract_signature.Actor) &
      ((Id=Sig1) | (Id=Sig2)) & name_crt(CA,Id)) {
    if (contract_available &
        isFirstSigned & not(allSigned)) {
      sigA = contract_signature;
      % signature validation and storage
      send(SS,Actor.record_Id.contract_signature.SS);
      isFirstSigned = false;
    }
    else {
    if (not(isFirstSigned) & not(allSigned)) {
          sigB = contract_signature;
          allSigned = true;
          % signature validation and storage
          send(SS,Actor.record_Id.contract_signature.SS);
    }
    else {
        %%% Some error has occurred! too many signature
        %%% Take necessary actions to recover/signal this
        %%% situation.
        }
    fi
    }
    fi
   }

  % (5)
  on (receive(SS,
              SS.signature_ok(?contract_signature).Actor) &
      ((contract_signature=SigA) |
       (contract_signature=SigB)) & name_crt(CA,SS)) {
    if (contract_signature=SigA) { sigA_verified := true;} fi
```

```
        if (contract_signature=SigB) { sigB_verified := true; } fi
        if (sigB_verified & sigB_verified) {
          %%% Contract signed and all signatures valid,
          %%% then ask to store the record
          send(SS,Actor.store(Record_Id).SS);
        }
        fi
      }
     }
    }
   }
}
```

The parameters of the entity `SecurityServer` are just two: one for itself and one for the certification authority. This implies that multiple requests from the several instances of the business portal or even multiple business portals can be served by the same instance of the security server. At the policy level, we find the identity certificate (`SS_identity`). At the workflow level, the variable `crl_available` stores the local copy (possibly not updated) of the CRL published by the certification authority and the predicates `storeRecord` and `sealRecord` model the database where contracts are permanently stored: the first argument of the former is the unique record identifier while the second contains the elements in the tuple of informations concerning a contract (original and signed versions); `sealRecord` models the flag signaling that a contract should be permanently archived. The possible behaviors of `SecurityServer` are as follows:

1. upon reception of a message from the business portal to create a record for a contract, it serves such a request and sends back the record identifier (associated to the contract) to the business portal;

2. upon reception of a message from the business portal to verify a contract signature (generated by a signer), the security server invokes the SPI provided functions `verify_signature` to do this and, if the check is positive, stores the signed copy to the appropriate record (which is part of the message together with the signed contract);

3. upon reception of a message from the business portal to permanently store the record containing the two signed copies of the contract, the security server puts a permanent lock (`sealRecord`) on the record;

4. when the variable `crl_is_expired` (provided by the SPI infrastructure) non-deterministically becomes true, the security server knows

that its local copy of the CRL is no more valid and he sends a request to the certification authority to obtain an update;[5]

5. upon reception of a new version of the CRL (the one published by the certification authority, hence the importance of the check on the identity of the sender of the message), the security server stores it in the local variable `crl_available`.

```
entity SecurityServer(Actor, CA : agent) {

  import SPI;

  clauses
   SS_identity : name_crt(CA, Actor);

  symbols
   %%% The Archiver and the Time Stamper are abstracted
   %%% away in this specification and they are encapsulated
   %%% in the Security Server.

   %%% Archiver (semantics of functions left unspecified)
   % create a record for a certain agent and a certain
   % contract (2nd arg)
   createRecord (agent, info) : info;
   % store a piece of information (2nd arg) in a certain
   % record (1st arg)
   storeRecord  (info, info) : fact;
   % puts a permanent lock on a certain record (arg)
   sealRecord   (info) : fact;

   %%% Local variable storing copy of the CRL published by CA
   crl_available : info;
   %%% Local variable storing the newly created record id
   record_Id : info;
   %%% Local copy of the contract
   Contract : info;
   %%% Local variable storing the identity of a signer
   Id : agent;

  body {
   while (true) {
    select {
     % (1)
     on (receive(BP,BP.?Contract.Actor) &
```

[5] In practice, the CRL comes with an expiration date. So, the value of the variable `crl_is_expired` will simply be established by looking at the CRL and the local time of the server.

```
        name_crt(CA,BP)) {
      record_Id = createRecord(BP,Contract);
      % storing the original version of the contract so as
      % to ensure integrity
      storeRecord(record_Id,Contract);
      send(BP,Actor.contract_created(record_Id).BP);
    }
    % (2)
    on (receive(BP,
                BP.Record_Id.?contract_signature.?Id.Actor) &
      storeRecord(record_Id,Contract) &
      name_crt(CA,BP)) {
      if (verify_signature(Id,contract_signature,
                           CA,crl_available)) {
      storeRecord(Record_Id,contract_signature)
      send(BP,Actor.signature_ok(contract_signature).BP);
      }
      else {
        send(BP,Actor.signature_ko(contract_signature).BP);
      }
      fi
    }
    % (3)
    on (receive(BP,BP.store(?Record_Id).Actor) &
        name_crt(CA,BP)) {
      sealRecord(Record_Id);
    }
    % (4)
    on (crl_is_expired) {
      send(CA,Actor.req_new_crl.CA);
    }
    % (5)
    on (receive(CA,CA.new_crl.Actor) &
        name_crt(CA,CA)) {
      crl_available := new_crl;
    }
   }
  }
 }
}
```

Finally, we are able to describe the entity `Environment` which allows us to create the instances of the previous entity to specify a scenario that is quite similar to that described in D5.1 (except for the absence of the Archiver and the Time Stamper).

```
entity Environment {

  types
```

```
    info  < message

  symbols
   %%% Messages
   contract_available, request_contract : message;
   contract_created (info) : message;
   store (info, info) : message;

   %%% Identifiers for the scenario
   %%% Two signers
   signer1 : agent;
   signer2 : agent;
   %%% Business Portal
   bp : agent;
   %%% Security Server
   ss : agent;
   %%% Certification Authority
   theCA : agent;
   %%% Identifier for the contract
   ci : info;

  init
   %%% Creating two instances of Signer
   new Signer(signer1, bp, theCA, ci);
   new Signer(signer2, bp, theCA, ci);

   %%% Creating an instance of the Business Portal, ...
   new BusinessPortal(bp, signer1, signer2, theCA, ss, ci);
   %%% ... of the Security Server, and ...
   new SecurityServer(ss, theCA);
   %%% ... of the Certification Authority
   new CA(theCA);

   %%% Integrity invariant property to verify at system level
   goals
    integrity:
     G (forall c_orig, c_sig1, c_sig2.
        (storeRecord(ci,c_orig) & storeRecord(ci,c_sig1) &
         storeRecord(ci,c_sig2) & sealRecord(ci))
        =>
        (c_sig1=sign(c_orig,signer1) &
         c_sig2=sign(c_orig,signer2))
       )
}
```

In the goal section we have specified the following integrity property (taken from [5]): the security server must ensure that the document that has been signed and uploaded by the user is the same as the one generated by the portal and submitted to the user for signature.

5.4 Identity Mixer

As a final example, we consider an excerpt from the specification of the Identity Mixer case study. This illustrates the use of interactive zero-knowledge proofs and pseudonymous channels in ASLan, as well as their composition with a protocol implementing these channels (all concepts are as introduced in deliverable D3.3 [4], but see also [14] for further details). Note that Identity Mixer requires a particular property that we do not model in the current formalization of channels: *anonymity*. This can be achieved, for instance, by a onion-routing [8]. The pseudonymous channels that we consider here, in contrast, provide us with the guarantee of a secure channel between an unauthenticated (thus pseudonymous) user and a server. This is crucial as the server must rely on the fact that all messages that appear to come from the same user indeed do, and all answers go back to that user.

Identity Mixer has its own kind of (cryptographic) pseudonyms that are different from the lower-level (channel-related) pseudonyms we have considered so far. For distinction, we will call the idemix pseudonyms just *nyms*. A nym is cryptographically linked to a user's master secret that we denote `masec(U)` for a user's real name `U`. Nyms are used when showing and proving credentials, and thus act for similar purposes as a user name in a non-anonymous setting. For unlinkeability, a user may create new nyms with a server at any time. We consider an exchange that allows the user to create a new nym. To that end, we define two entities for setting up a new nym, a user and a server entity.

```
masec(agent):nat;  %% the master secret of a user
spk(msg,msg):msg %% abstraction of interactive zero-knowledge
                 %% proofs (see D3.3)
commit(agent,nat,nat):msg %% commitment function
ptag(nat,agent,nat,nat):msg %% constructor for pseudonym

entity UserCreateNym(Actor,O: agent){
  symbols
    R_1,R_2 : nat;
    C,P : msg;

  body{
    R_1 := fresh();
    C := commit(O,R_1,masec(Actor));

    [Actor] *->* O: spk((masec(Actor).R_1),C)
    %% interactive zero-knowledge proof that C is a commitment
    %% of the appropriate form commit(O,R_1,X) and that the
    %% user knows X and R_1
```

```
    O *->* [Actor]: ?R_2

    P := ptag(masec(Actor),O,R_1,R_2)
    %% computing the pseudonym of the user

    add(myNyms,(P.O))
    %% the nyms of Actor contain now also the fact that P is
    %% a nym for use with O

    [Actor] *->* O: spk((masec(Actor).R_1),P)
    %% interactive zero-knowledge proof that P is a pseudonym of
    %% the appropriate form and that the user knows X and R_1

  }
}

entity OrgCreateNym(Actor: agent){
  symbols
    PU  : agent;
    R_2 : nat;
    C,P : msg;
    %% Note that the following variables represent values
    %% that the server cannot actually see, but that are
    %% involved with the zero-knowledge proof:
    _masec, _R_1 :: nat;

  body{
    ?PU *->* Actor: spk((?_masec.?_R_1),commit(Actor,_R_1,_masec))
    %% accept interactive zero-knowledge proof as valid when
    %% receiving term of this format for values _masec and _R_1
    %% that "Actor" does not learn.

    R_2 := fresh();

    Actor *->* PU: R_2

    PU *->* Actor: spk((_masec._R_1),ptag(_masec,Actor,_R_1,R_2))

    add(myTags,ptag(_masec,Actor,_R_1,R_2))
  }
}
```

As explained in D3.3, we can model a non-interactive zero-knowledge proof as a function of the involved terms, including secrets, combined with appropriate pattern matching on the receiver side. Thus, to understand the formulation of this example in ASLan requires one to consider the combination of corresponding send and receive actions, in particular. First the user sends a proof about a commitment, namely `spk((masec(Actor).R_1),C)`

where `masec(Actor)` and `R_1` are secrets that are not revealed by the proof, and `C` is a commitment. The server receives this as

```
spk((_masec._R_1),commit(Actor,_R_1,_masec))
```

i.e. the server will accept only those commitments that have the form

```
commit(Actor,_R_1,_masec))
```

where `Actor` is the server's name and where the values `_R_1` and `_masec` must be the same as in the first argument of the `spk` term, proving that the prover indeed knows these two values. To clarify that the verifier of the proof does not learn these values, we have denoted these variables with an underscore on the server side. In the protocol, the server replies with a fresh number `R_2` and the user forms the new nym `P` of the form

```
ptag(masec(Actor),O,R_1,R_2) ,
```

stores this and sends it to the server along with the proof that it has the appropriate format and contains the correct arguments, in particular the same master secret as the commitment before.

This procedure relies on the assumption of pseudonymous secure channels between the user and the organization server. A simple way to implement these assumed pseudonymous channels could be TLS without client authentication, or even simpler, a Diffie-Hellman key-exchange without client authentication.[6] The client side of that looks as follows in ASLan, and the server side is analogous:

```
entity DHClient(Actor, O: agent, Payload: payload){
  symbols
   X : nat;
   GX,GY,K : msg;

  body{
    X:=fresh()
    GX:=exp(g,X)
    Actor -> O: GX
    O *-> Actor: ?GY %% the server O must be authenticated to
                     %% obtain a secure channel
    K := exp(GY,X)
    Actor -> O: {| Payload |}K
  }
  goal: [Actor] *->* O: Payload
}
```

[6]Note that these are not sufficient for providing anonymity; while the privacy aspect is central to Identity Mixer in general, we do not go into further details of modeling it here.

Here, the client sends its half-key `GX:=exp(g,X)` on an insecure channel (denoted simply by `->`) and the server replies by sending his half-key `GY` on an authentic channel. The full key is then `K := exp(GY,X)` and the client uses it to symmetrically encrypt a Payload she sends to the server. We thus have the goal that the payload message is transmitted on a pseudonymous secure channel.

The compositionality result in [4, 14] allows us to conclude that, given that the key exchange indeed satisfies its goal (which it does) and given that it is horizontally and vertically composable with the application service (which is also the case here), we can indeed realize a transmission on the pseudonymous secure channel by the Diffie-Hellman key exchange.

6 Conclusion

We have presented version 2 of the language ASLan, whose semantics is defined by translation to the more low-level ASLan v.1.1. Both of these languages are going to be supported by the AVANTSSAR Platform that we are developing. They 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. ASLan v.2 has the look and feel of procedural and object-oriented programming languages, and thus can be employed by users who are not experts of formal protocol/service specification languages. We demonstrated its flexibility and expressiveness by considering four case studies from Deliverable D5.1, focusing in particular on the modularity aspects that allow us to consider static service and policy composition.

As discussed in the description of work (Annex I), we have planned the definition and use of ASLan in a flexible way, in the sense that ASLan will be continuously extended during the course of the project, in order to cover various features needed by the case studies and the industrial-strength applications that we will consider. In particular, ASLan v.3, the final version of the language, to be delivered in D2.3, will include dynamic service and policy composition, so that the language and the whole AVANTSSAR Platform will be applicable for the full-fledged specification and analysis of the case studies.

As future work, in addition to the language extensions towards ASLan v.3, we will also consider optimizations of the translation from the high-level to the low-level language (as presented in this deliverable), in order to produce formal models that allow for a more efficient automated analysis.

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

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

[8] J. Camenisch and A. Lysyanskaya. A Formal Treatment of Onion Routing. In *Proceedings of Crypto'05*, LNCS 3621, pages 169–187. Springer-Verlag, 2005.

[9] 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.

[10] 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.

[11] 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/.

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

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

[14] S. Mödersheim and L. Viganò. Secure pseudonymous channels. In *Proceedings of Esorics'09*. Springer-Verlag, 2009 (to appear). Extended version: Technical Report RZ3724, IBM Zurich Research Lab, 2008, domino.research.ibm.com/library/cyberdig.nsf.