Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures

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

www.avantssar.eu

Deliverable D2.1
Requirements for modeling and ASLan v.1

Abstract

This deliverable describes the requirements for the ASLan specification language, as well as the first version of the language (called ASLan v.1). ASLan is a formal language for specifying security-sensitive service-oriented architectures, the associated security policies as well as their trust and security properties.

Deliverable details

Deliverable version: *v1.1*
Classification: *public*
Date of delivery: *31.12.2008*
Due on: *31.12.2008*
Editors: *all*
Total pages: *72*

Project details

Start date: *January 01, 2008*
Duration: *36 months*
Project Coordinator: *Luca Viganò*
Partners: UNIVR, ETH Zurich, INRIA, UPS-IRIT, UGDIST, IBM, OpenTrust, IEAT, SAP, SIEMENS

Contents

1 Introduction **5**

2 Requirements of ASLan per case study **7**
2.1 Banking services 7
2.2 Software distribution 8
2.3 Credential anonymizer 10
2.4 Citizen and Service Portals 12
2.5 Document exchange procedures 15
2.5.1 Digital contract signing 15
2.5.2 Public bidding 17
2.5.3 Requirements for property specification language (the digital contract signing/public bidding case) 18
2.6 E-health infrastructures 18

3 Language for specifying atomic services and non-composed policies **22**
3.1 Motivations 22
3.2 Requirements Coverage 23
3.3 ASLan Syntax 24
3.4 ASLan Semantics 28

4 Example ASLan specifications **31**
4.1 Example 1: Public bidding protocol with DKAL policies (an excerpt) 32
4.2 Example 2: LOP in ASLan (an excerpt) 48
4.3 Example 3: Car Registration in ASLan (an excerpt) 51

5 Conclusion **67**

A Prelude File **70**

List of Figures

1 Task assignment upon clerk request evaluated with respect to SSOD constraints. 51

List of Tables

1 List of features 6
2 Facts and their informal meaning 50

1 Introduction

This deliverable describes the requirements for the ASLan specification language, as well the first version of the language called ASLan v.1. ASLan is a formal language for specifying security-sensitive service-oriented architectures, the associated security policies, as well as their trust and security properties.

The collection of case studies described in Deliverable D5.1 (Problem Cases and their Trust and Security Requirements) [4] provides us with a basis for compiling a list of features that ASLan should contain. We extract from each case study its security and trust requirements, in the form of the minimal language features needed to formalize the case study. We also identify what features ASLan should contain beyond existing formal languages designed for similar purposes. As a reference here, we compare the minimal requirements for ASLan, in each case study, with the existing formalisms as described in Deliverable 6.2.1 (State-of-the-art on specification languages for service-oriented architectures) [5].

The case studies that are considered in AVANTSSAR are diverse and require various specification primitives. We propose a fine-grained specification language as ASLan v.1; namely, we extend the Intermediate Format [6] with Horn clauses and LTL for this purpose. This choice allows us to already cover a considerable part of ASLan's requirements, while certain features are not included in this version of the language, as will be pointed out in the following. Extending ASLan v.1 to address the remaining issues is foreseen for future deliverables.

Structure of the document Each case study explained in Deliverable D5.1 (Problem Cases and their Trust and Security Requirements) is considered in Section 2 as a subsection. In each of these subsections, detailed specification requirements are described that are needed for formalizing the corresponding case studies. The extracted requirements are summarized in Table 1. The table serves as reference for the requirements of ASLan in the current deliverable, as well as in future ones.

In Section 3, we turn to the syntax and semantics of ASLan v.1. As examples, in Section 4, we specify in ASLan v.1 three case studies described in Deliverable D5.1 (Problem Cases and their Trust and Security Requirements), namely the public bidding portal, the loan origination business process, and the car registration service.

Table 1: List of features

Feature	Referenced in Section
A notion of process	general
Send and receive actions	general
Sequential composition	general
Parallel composition	general
Cryptographic primitives (encryption, hash, digital signature, etc.) and constructs to check digital signatures	general
Indistinguishability	§ 2.3
Channels with associated security properties	§ 2.3, § 2.4
Data bases	§ 2.3, § 2.4
Composed messages and algebraic data types including • Selectors • Pattern matching • Assign security mechanisms to every part or set of parts of a message or of a algebraic data type individually • Projections • Dynamic allocation • Dynamic initialization and manipulation	§ 2.4
Authorization policies (role and attribute based) including rights delegation	§ 2.4
Guards or other conditional mechanisms	§ 2.4
Loops in workflows	§ 2.4
Composed services/workflows	§ 2.4
Workflow escalation and Duration annotations	§ 2.4
Trusted third parties, Certification authorities, Public key infrastructures, Certificates, Attribute certificates	§ 2.3, § 2.4
Trust policies, assertions	§ 2.3, § 2.4
Remote authentication, session, single-sign-on, translate and store different credentials, derived assertions, security tokens	§ 2.4
Non-repudiation	§ 2.3, § 2.4
Simple data flow secrecy	§ 2.3, § 2.4
Security properties based on workflow	§ 2.4
Time constraints	§ 2.3, § 2.4
Time stamps	§ 2.5.1
List construction primitives	§ 2.5.1
Guarded choice operator	§ 2.5.1

2 Requirements of ASLan per case study

2.1 Banking services

This case study emphasizes the need for constructs for expressing access control and the control flow between services. The communication part is limited, and can be abstracted under the assumption that secure channels exist between services.

Overview. In our tentative modeling of the LOP, we have considered a system composed of a multiset of interacting *entities*. Each entity is defined by its knowledge, and executes *actions* according to its *policy*.

Knowledge: The knowledge encompasses the *certificates* that can be sent are received by the entity, the *pieces of data* it has created, acquired during the execution of the system, and the *messages* that it issues or receives;

Actions: The messages can be *sent* or *received*, the pieces of data can be *removed* or *added* to an entity's state, and the certificates are issuable or acquirable. The difference of treatment between certificates and pieces of data is justified by the instant availability of data;

Policy: The policy is essentially an access control policy defining the conditions in which an action can take place. It organizes the trust negotiation

In the following description, human agents, service interfaces, objects, applications, organizations and business processes are modeled by communicating entities.

We have found that the modeling of the LOP required to consider various entities:

- The client is a human agent that does not have a fixed set of methods (as a service would have), but has an implicit security policy employed to specify who, and under which circumstances, can access the data it submits to the bank.
- A constraint on the LOP is that all human agents that have the *administrative* role of clerk have the possibility to participate in the Business Process by playing the *functional* role of clerk in the LOP. Access control in the business process (and, application-level security policy) defines the conditions under which an agent having an administrative

role of clerk can acquire a functional role of pre/post-processing clerk, as well as the possible actions for pre/post-processing clerks.

- Services are interface to processes and to objects. Both processes and objects have an access control policy and define a set of methods that can be used to interact with them. One may have several services to interact with a given object (*e.g.* in the LOP, the customer's credit rating is accessed by two different services, one for the bank and one for the credit rating bureau). To model this situation, we have defined an object and the service permitting to access this object as distinct entities, and assumed that there is a secure communication channel (actually a local channel, inaccessible to an outsider) between an object and the service permitting the access to this object.

- Parallel composition is ubiquitous, but it is restricted by some constraints stemming from the application's logic and/or security policy: an object shall be initialized before being read, actions can be executed or not depending on the values transmitted (application logic), and only agents having a certain role can perform some actions (security policy). This corresponds to the flow/link constructs in BPEL.

- To express that the client accepts or refuses access to the data he has sent to the bank, this data is collected in an object within the bank, but the access control policy on this object is controlled by the client. This corresponds to the definition of a trust relationship combined with the delegation of an access control decision. We note that this encoding is standard in the sense that often dedicated servers are employed to distribute authorizations, and individual services only check for the presence of an authorization token.

2.2 Software distribution

Language requirements The case studies requires the usual constructs needed to model security protocols and web services, for instance:

- a notion of process, for the various agents taking part in the scenario (suppliers, distributors, operators, targets). Most of them are fully automated, but software approval does contain human tasks. The only manual tasks that we presumably need to model are the decision whether to apply an approval signature or not, and which destinations the software is approved for,

- parallel composition of processes,

- primitives for communication between processes (potentially at various security levels; for this case study simple insecure message send/receive is likely sufficient),
- cryptographic primitives (encryption, digital signatures) used e.g. to achieve confidentiality, authentication and non-repudiation properties, and
- the ability to express audit log mechanisms.

In addition, there are several features which are specifically relevant for this case study:

- a notion of process *instantiation and specialization/parametrization*, in which several process instances (agents) can be derived from a common base. The key element in the software distribution case study is the *Software Signer Verifier* (SSV) which checks digital signatures on received software and applies another signature before re-sending it further. Several instances of the SSV having a common behavioral pattern need to be modeled.
- Provisions for expressing aspects of public-key infrastructures, in particular the notions of certificates.
- Provisions for expressing *trust* relationships between entities, related to the requirement that software items should only be accepted from trusted sources. Certificates are used for securely communicating trust relations. This may involve the use of policies as described below.
- Provisions for expressing *policies* defining what software may be sent and installed on each target, and the ability to model software licenses. Policies should be specifiable locally for each agent and may need to be dynamically changeable. The policies are used to dynamically and automatically create and revoke trust relations, for instance if the OEM declares that a certain supplier is a trusted source of software items, then the target owners may accept the supplier as a source. This can be specified in Prolog/Datalog-style rules. Software licenses typically contain the software identifier, the identifier of the authorized target owner or target device, and an expiration date. The license is signed typically by a software distributor.
- The ability to specify properties related to time intervals: software licenses may be valid only for a specified period of time, and signatures

that must be verifiable for long time (e.g. non-repudiation for 50 years) need to be refreshed before they expire.

2.3 Credential anonymizer

Patterns. It is a standard feature of specification languages that one can specify patterns, i.e. terms with variables, to express for instance a set of acceptable messages for a receiving agent.

A crucial example is as follows. All protocols of Identity Mixer are based on non-interactive zero-knowledge proofs. For instance, a prover P shows that for a publicly known value y it knows a secret value x (that is not revealed) such that $y = f(x)$.

As discussed in Deliverable 3.3 on attacker models [3], in the section on zero-knowledge proofs, we introduce a model that is purely based on pattern matching. We thus require that ASLan allows us to specify patterns accordingly.

Databases. We need the concept of a simple (relational) database for every participant, shared over all its sessions, to keep track of the pseudonyms and certificates issued or received.

We assume the following operations:

- $\texttt{Store}_U(\, M\,)$: user U stores message M in its database.
- $M \leftarrow \texttt{Load}_U$: user U loads a message M from its database, where M is a term with variables. If there is no instance of M in U's database, we cannot proceed. Otherwise we can proceed with any appropriate instance of the variables, i.e. matching M against an element in the data-base.
- $\texttt{CheckStore}_U(\, M\,)$: user U checks (and only proceeds) if there is at least one message of the form M (i.e. an instance of M) in its data-base.

Privacy Goals. Besides classical secrecy and authentication/integrity goals, the whole point of Identity Mixer is the protection of the privacy of the users. In particular, no server should learn more about a user than the information that the user intentionally transmits. Also, it should not be possible to link several actions of a client, even when some servers collaborate. These goals are not classical properties of traces but rather properties of indistinguishability between traces.

For instance, a user's name or birthdate is not a secret in the classical sense (like a secret password) but rather the intruder should not find out whether a particular user performed a particular action. Also, the intruder should not find out whether two actions were performed by the same or by different users.

Anonymous Channels. In order to achieve anonymity of agents, one needs to consider channels that hide any information about the client that could make its actions linkable (such as IP addresses). This can be achieved using techniques like onion-routing [16].

For our formal model, we abstract from the realization of anonymous channels between a pseudonymous user U and an authenticated organization O. This combines two aspects of channels:

- Sender and receiver invariance, i.e. authentication and secrecy with respect to a pseudonymous user and an authenticated organization [20, 12, 25].
- Unobservability: all other parties cannot observe that any communication takes place. The organization cannot link several sessions with the same user, if the user uses a different pseudonym in every session.

The anonymity channels are only necessary to achieve privacy. All goals that are not related to privacy must hold even when using a completely insecure, non-anonymous channel, like unsecured Internet connections.

Thus, even when an intruder is able to break the mechanisms of anonymization, the correct behavior of credentials shall be guaranteed; as a consequence, we will analyze the Identity Mixer system with both anonymous-secure and standard insecure channels.

Advanced Features. The above features are necessary to analyze the basic version of Identity Mixer as of [15]. Modeling further recent extensions of Identity Mixer requires also the following features of the specification language:

- An abstract model of arithmetic and time for expressing and dealing with statements like "$today() - Birthdate \geq 18y$". This does not require a complete model of arithmetic, of course. Depending on the concrete application, we may completely abstract from the properties of the operators and leave them as uninterpreted functions, or we may model some basic properties such as associativity and commutativity of addition.

- Support for policy languages that are expressive enough for statements as the above age-example.
- We also expect that algebraic properties of the used (abstract) function symbols need to be considered in some cases.
- Also, we assume that general temporal properties of goals can be specified as the system's requirements, e.g. to express non-repudiation/accountability.

2.4 Citizen and Service Portals

Requirements for system specification language In addition to the general constructs listed in Table 1, the Citizen and Service Portals scenario needs several more specialized constructs. These are listed in the following enumeration and emphasized using bold font.

- **Non-monolithic (composed) messages and algebraic data types:** Using the street party application as an example in Deliverable 5.1 [4, 3.4.1, Scene 3], a citizen has to enter data in a form that encompasses all data necessary for at least one of the authorities involved into the approval process. The different authorities might receive different parts of the form containing exactly the data they are allowed to see.

  In ASLan, it must therefore be possible to access the parts of a message or of an algebraic data type using **selectors** or **pattern matching techniques**. It must be possible to **assign security mechanisms to individual parts or sets of parts of a message or of an algebraic data type**, e.g., to specify, that a specific set of parts of a message is integrity protected or confidential. Parts of algebraic data types might be encrypted for more than one receiver, e.g., two, such that the encrypted part is available in two "flavors" for the two different receivers. To be able to alter algebraic data types or instances thereof, ASLan needs constructs for the **construction** of new algebraic data types and messages, including the **data initialization and manipulation** of those data types.

- Another requirement from the street party application example mentioned above is that ASLan must be capable of expressing **channels including their security properties**. An example might be VPN based communication between the authorities involved into the approval process. These channels guarantee confidentiality and integrity during the message passing process, including mutual authentication.

- Authorization is another basic requirement of the Citizen and Service Portals scenario. ASLan therefore needs constructs for **authorization policies**, such as role based authorization policies or attribute based authorization policies. It must be possible to express **rights delegation**, e.g., a citizen must be able to explicitly declare his/her tax counselor as authorized delegate for tax affairs.

- In order to model non-repudiation and rights delegation, ASLan must provide constructs for **digital signatures** and **checking of digital signatures**, which should be available via the cryptographic primitives mentioned in Table 1. All this can be probably subsumed by the function symbols and algebraic data types discussed above, that allow us to create and manipulate new data types and messages from others via abstract function symbols. The security properties of the constructed data is given by the intruder deduction rules on these constructions.

- Apart from sequential and parallel composition of actions of a service to model its workflow, ASLan must be able to handle **directed cycles** (that is, loops) in a workflow graph. For example, in the annual tax declaration scene, Mike fetches and stores his tax form several times before he finishes his work and submits the form. This can be modeled using a bounded loop in the workflow.

- Another requirement on the workflow modeling capabilities of ASLan is that **composed services/workflows** are needed. For example, in the street party application scene, Mike sends a message to the portal and receives the result of the application. This is a rather simple workflow from Mike's point of view. But in the background, several public offices and institutions are involved by the portal. Thus, the "real" workflow that is triggered by Mike is much more complicated. Furthermore, internally to each public office and institution, there is a specific workflow to process the street party application. Thus, the full workflow triggered by Mike is the street party application process defined by the portal composed with the internal workflows of the specific offices and institutions. ASLan must be able to compose workflows when necessary and desired by the user.

- **Duration annotations** and **workflow escalation**, (that is, dynamically routing workflow content to different entities than the "usual" ones) are needed to model time constraints of workflows, or to find solutions where the allocated persons is on vacation, traveling, or too

busy to complete their tasks. For example, in the annual tax declaration scene, the tax authority must decide on Mike's tax declaration within three months. The tax authority might try to enforce this duration by an automatic escalation, e.g., the manager of the assigned employee is informed after 2 months about the delay. (Probably we are able here to abstract time and related issues, like the number of messages can be sent within that period, etc.)

- Provisions to express **trusted third parties**, in particular **certification authorities** and **public key infrastructures** including the notion of **certificates** and **attribute certificates** are required. **Attribute certificates** are a means to express **assertions**. One document may contain several certificates. Certificates are not really a further concept, not contained elsewhere in the list, but rather is a special type of a signed statements, with some special interrelation to policies and to secrets (keys).

- A further requirement on ASLan are provisions for expressing **trust policies** in the form of *Who trusts whom in saying what*. Additionally, **assertions** are needed to specify, *who says what*.

- ASLan must provide constructs to model **remote authentication** (log into remote servers) and of **session** during which an authentication is valid.

- Last, but not least, another language requirement is the support of **single-sign-on** techniques. A single-sign-on server must be able to **translate and store different credentials**, representing SAML assertions or other tokens, as well as knowledge (facts) about users, gained via different mechanisms. Additionally, a SSO server must be able to **derive assertions**, e.g., the assertion *person is older than 21* includes the assertion *person is older than 18*.

Requirements for property specification language In addition to the general security properties we want be able to formalize, the Citizen and Service Portals scenario needs several more specialized properties. These are listed in the following enumeration and emphasized using bold font.

- In ASLan, we need a property for **non-repudiation of origin, receipt, and/or content** of messages

- Additionally, e.g., for data austerity (data minimization), we need a **secrecy property** to reason about simple data flow (not full non-interference) characteristics. For example, it should be possible to express the security property that the tax authority must not be able to see specific private information (e.g., gender) but the citizen's PII and facts crucial to tax declaration, e.g., annual income.

- ASLan must have the potential to prove **security properties based on the workflow of a service**. For example, in the car registration scene, the following security property was postulated:

  Final documents stored in the CentrRep must be consistent, i.e., signatures are correct, etc.

  If the presented work flow is adhered to, the inner signature of a citizen is always verified by a RegOffEmpl before it is written to the CentrRep and thus, the documents in CentrRep are always consistent.

- **Time constraints** are necessary in ASLan to express timeliness properties. For example, the succeeding two security requirements must be expressible in ASLan:

  – The decision on Mike's tax declaration is made within three months. This could be rather hard to prove or guarantee.

  – A weaker requirement is: if there is no decision on Mike's tax declaration within three months, at least a high official/manager of the tax authority is informed about the delay.

2.5 Document exchange procedures

This section is divided into two subsection: One on the digital contract signing, and one on the public bidding scheme.

2.5.1 Digital contract signing

General Language requirements This section contain language requirements for specifying services and protocols.

The constructs usually available to model security protocols are generally necessary here. For instance, the following constructs will be needed in ASLan in order to model this case study. Observe these are "traditional" security protocol modeling ingredients.

- A notion of process, or service, for modeling honest processes or services.

- Each process is equipped with actions for communication, e.g. *send* and *receive* primitives.
- Each process needs a collection of cryptographic functions, such as hash functions, encryption schemes, etc.
- ASLan has to facilitate the parallel composition of honest processes, possibly also incorporating an attacker model.

We also need constructions for:

- The specification of a workflow, so that we can keep track of the current state of a business process in this workflow (and thus know which queries are acceptable);
- The specification of local variables to keep an internal state. This part shall be translatable to BPEL (or WSBPEL, or BPEL4WS, or...).

Certificates We need a high-level language to describe the semantics of certificates. We may need also rules and negation to reason about them.

There are however some features paramount to this case study which are not often made explicit in specification languages for security protocols. These consists of:

- Time stamps: The time stamping service plays an explicit role in the contract signing case study by OpenTrust.
- Guarded choice operators: Honest processes need to make choices based on the value of internal variables. See the SPI and HHH in the contract signing case study (Deliverable D 5.1), where local security policies determine which messages are to be sent and received.
- List creation constructs: Honest processes need to create lists and communicate them to the other processes. See the PKI and the archiver service in the contract signing case study Deliverable D 5.1.

Communication security We need to describe channels with associated properties (such as encrypted public channels or secure, or local channels). This part will probably be translated to WSDL (for the definition of the data within the messages) with an attached WS-SecurityPolicy specification.

Application Security Policy An Application Security Policy is the part of the access control specification related to application-level concerns (*e.g.* separation of duty). We anticipate to specify this policy with XACML in which rules describe the conditions that have to be satisfied in order to fire a transition or invoke a method, or take an action. In order to meet complex requirement we should also provide for a way to collect the certificates that will be evaluated by these rules. We plan to do so either during a runtime trust negotiation phase (within the application) or to pre-compile the certificate exchanges by orchestrating the application with the relevant trusted certificate servers.

Finally, in order to meet complex access control constraints, we may have to account for the side-effects of rules, the different decision combination algorithms and their interactions.

2.5.2 Public bidding

Language requirements Besides what is required in the contract signing case study, public bidding relies on list (or, set) processing in two critical points:

- The protocol concerns a set of tuples, $(B_i, a(B_i), t(B_i), f(B_i))$, where B_i is the ith bidder's ID, $a(B_i)$ is its application, $t(B_i)$ is its technical offer, and $f(B_i)$ is its financial offer. The set of all these tuples are initially stored by the BPA (Bidding Portal, see deliverable 5.1 [4]). Once the deadline is reached, the set of $(B_i, a(B_i))$ is forwarded to BM. A subset of these is selected by BM, and sent back to BPA. Then, BPA sends the set of $t(B_i)$ of those tuples which were deemed suitable by BM to PC. Next, PC evaluates these, and gives back the set of tuples $t(B_i), ev(B_i)$ to BPA. The BPA forwards $(B_i, t(B_i), ev(B_i), f(B_i))$ to BM, etc. It is possible to model these steps statically (i.e. assuming a finite fixed set of bidders), but, first, this would be a very abstract model of the way an actual BM or BPA would in practice be programmed, and second, an exponential growth in the specification size becomes inevitable.

- In the *extended public bidding* variant, BPA (and SPI) receive a set of signatures on offers, and later receive the offers themselves, which should agree with the initial signatures.

2.5.3 Requirements for property specification language (the digital contract signing/public bidding case)

Security requirements of the document exchange scenario consist of authentication, authorization, integrity, traceability, timeliness, and non-repudiation. In order to express these properties, the following constructs are needed beyond what is provided in existing tools such as Avispa.

1. A logic of knowledge/belief to establish whether the items collected in a proof record always provide a non-refutable proof.
2. Liveness: asserting liveness properties would normally require certain assumptions on the message delivery system.
3. Timed properties to refer to the exact time an action happens, etc., as opposed to precedence or casual relation between different events.
4. Separation of duty: Requirement 22 in deliverable 5.1 [4] concerns separation of duties. For instance the technical committee should not be one of the bidder in the bid (they can be bidders in other bid sessions, though).

2.6 E-health infrastructures

The E-health infrastructure case study is concerned with the problem of handling personal health information in a network of hospitals and/or clinics. The challenge underlying this case study is to allow the most flexible and easy access to crucial information about a patient whenever this is required (e.g., an emergency) and provide a high degree of privacy for sensitive data (e.g., a disease with an important social impact).

Language requirements. The case study requires a specification language capable of describing a 'virtual' database of (health) records. The database is virtual in the sense that it may be distributed over a network and each node may have different ways to store and represent the same information. It may be the case that information stored at different nodes is inconsistent because it has changed over time and it has not been updated at one location. This implies that we should be able to guarantee *coherence* among (partially) overlapping pieces of information. In general, coherence is a functional and not a security property, apart from the general integrity protection that it implies. More importantly for the specification language of AVANTSSAR, there should be a way to specify the "sensitivity level" of each

health record as this tag permits the enforcement of various policies. Thus, the sensitivity level may be interpreted in a strict way (as for instance in the classical models of Bell-LaPadula or Biba), or, if the situation requires it, in a controlled but more relaxed way. An example of policy that respects sensitivity levels differently, depending on the situation, is the following: some information that should be known only by the treating doctor in a normal situation could be disclosed to another doctor if (and only if) there is an emergency.

The database of health records is updated and queried by various actors (such as the patient, doctors, nurses, receptionists) so as to extract, append or delete information according to the policies of the hospital/clinic. There are some general concepts that should be available in ASLan in order to model the various actors and part of their behavior.

- First, the notion of process, or service, to specify the various actors.
- Second, each process should be able to query or to update the database. These operations can be seen as transactions on the database, i.e. modifications of the database which should be perceived as atomic but are internally composed of several smaller actions including sending/receiving messages over a network, logging, etc.
- Third, for accountability, all actions on the database should be logged and non-repudiation should be guaranteed; for this purpose, standard techniques available in standard databases may be sufficient. In such case, ASLan needs some constructs to model the behavior of the logging.
- Fourth, processes should be composed in parallel as their access to the database may occur at any time; there are no constraints on the sequencing of actions, except perhaps avoiding races when updating records but this can be enforced by using locks or some other mechanism which, as above, is standard in databases.

The most important constraints on actions concern their executability provided that some crucial properties — such as confidentiality, privacy, communication security, and trust (Is the originator or the consumer of an information trusted to produce or use the information?) — are guaranteed. In order to specify these properties, ASLan should contain the following ingredients.

- Constructs to specify *policies, their modular composition and their dynamical aspects* to access certain records according to their sensitivity levels. In particular, identifying suitable mechanisms for the static and

dynamic composition of several policies seems to be a crucial point as it is very easy to get contradictory policies in distributed systems (if negation is allowed). Of course, there should also be a mechanism to enforce the policies, i.e. when querying or modifying the database, it must be possible to establish if the policies allow or not the the executability of the various actions.

- Constructs to describe an *identification/authentication mechanism* including single sign-on.
- A mechanism for *certificates* is crucial in this case study as they enable an actor A to make available to another actor B some information which, according to the enforced policy, should not be accessed by B. They are also useful to guarantee authenticity and integrity of the health records as each modification of the database should be accompanied by a certificate containing the identity of the actor carrying out the operation.
- Certificates (especially those implying exceptions to the enforced policies) should have limited time-validity; hence, a mechanism for *time stamping* is another crucial ingredient.
- Accessing a distributed database requires one to send and receive messages over a network. Hence, it ought to be possible to specify actions for circulating messages over secure or insecure channels.
- Constructs to model anonymization.
- Constructs to represent signatures.

Relation to existing languages. The policies that regulate the access to distributed databases can be specified in several ways. In particular, XACML is a good starting point to look at for specifying XML-based access control policies to describe access control restrictions to the actions on e-health records. In fact, XACML adopts the principle of basing authorization decision on attributes: identities and 'sensitivity levels' can be seen as an instance of such attributes. Two other languages that could be used as starting points are SecPAL and DKAL:

SecPAL (Security Policy Assertion Language) [13] is a relatively simple declarative authorization language with a rather high expressiveness and a good execution efficiency. The syntax is close to natural language, and the semantics consists of just three deduction rules. The language can express

many common policy idioms using constraints, controlled delegation, recursive predicates, and negated queries.

DKAL (Distributed Knowledge Authorization Language) [18] is a new tractable high-level authorization language for distributed systems based on existential fixed-point logic and is considerably more expressive than many existing authorization languages in the literature.

WS-SecurityPolicy is language to define policies for message security (in WSS SOAP) and trust (in WS-Trust). Of particular interest for the case study is the capability of the WS-SecurityPolicy to identify the parts of the message that is to be protected and the level of protection in terms of confidentiality and integrity as this matches the 'sensitivity levels' of e-health records. WS-Policy is a meta-policy composition language capable of modularly combining policies, written in WS-SecurityPolicy, to represent whether and how a message must be secured, whether and how a message should be delivered reliably or whether the request must follow a transaction flow.

The notion of process or service is present in a form or another in all languages for web-service specification; PSL being the language where 'process' is its founding notion. The notion of transaction, modeling an operation that can be executed by a process, is present in transaction-based formalisms such as WS-AtomicTransaction and some of the languages in the Business Process Layer. For example, WS-AtomicTransaction is capable of handling transactions in distributed databases by using variants of a two-phase commit protocol to ensure basic properties such as atomicity. The notion of transaction in this kind of languages is a good starting point to model the operations of this case-study as many aspects of such a notion of transactions (e.g., accountability, non-repudiation) are variants of those readily available. Also, the parallel composition of processes is implicit when talking about transactions and need no further specifications.

3 Language for specifying atomic services and non-composed policies

This section describes the syntax and semantics of ASLan v.1. We proceed as follows. ASLan v.1 extends the specification language Intermediate Format (IF) of the AVISPA Tool [7] and in Section 3.1 we motivate and illustrate this in more detail. In Section 3.2, we discuss the coverage of the language requirements of Table 1. In Section 3.3, we give the syntax of ASLan, and we then describe the semantics of the language in Section 3.4.

3.1 Motivations

We define ASLan 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: it provides the user with an expressive language for specifying security protocols and their properties, based on multiset rewriting. Moreover, IF comes with mature tool support, namely the AVISPA Tool and all of its back-ends. However, the IF language has three major shortcomings that make it unsuited for the analysis of complex services:

- In IF, the behavior of the system can only be described by means of transitions and this makes IF inadequate to express security policies which are usually best described as invariants.
- In IF, security goals can only be expressed as reachability properties and this makes it inadequate to express complex security goals (e.g. fair exchange) that occur in complex services.
- In IF, the intruder can always overhear or intercept messages and fake new ones. This is not always possible in recent protocols and services, which often assume for their proper functioning that communication is carried out over secure (confidential and/or authentic) channels (cf. Deliverable 3.3, Attacker models, [3]).

To overcome these shortcomings, we have thus defined ASLan by extending IF with the following two important features:

Horn Clauses: In ASLan, invariants of the system can be defined by a set of Horn Clauses. Horn clauses allow us not only to capture the deduction abilities of the attacker in a natural way, but also, and most importantly, they allow for the incorporation of authorization logics in specifications of services.

LTL: In ASLan, complex security properties can be specified in LTL. As shown, for instance, in [4], 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.

Various languages have been proposed in recent years to specify trust and security of service-oriented architectures, for example BPEL [26], π calculus [24], F# [14], to name a few. These languages, however, focus only on some of the aspects of service-oriented architectures, and covering all the features described in section 2 in these languages, if not impossible, would be artificial. We have opted for ASLan v.1 to be an extension of IF with Horn clauses and LTL because in that way ASLan is expressive enough that many high-level languages, such as BPEL, can be translated to it. (An automated translation of BPEL specifications into ASLan is well underway and we are working on similar translations for other high-level languages.) Moreover, and more importantly, ASLan provides a solid basis from which to provide tool support: defining ASLan as an extension of IF allows for immediate partial evaluation of some of the case studies, which in turn provides guidelines for adding high-level constructs to ASLan v.2. In fact, as described 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. Our preliminary studies show that atomic web services with non-composed policies can be specified already in ASLan v.1, e.g. [1], and some concrete examples are discussed in the following.

3.2 Requirements Coverage

Before we proceed to the description of the syntax and semantics of ASLan, it is important to observe that all but a few of the language requirements stated in Table 1 are met by ASLan v.1: the language and property specification requirements that explicitly refer to time (workflow escalation and duration annotation, time constraints, and time stamps) are not yet included in ASLan v.1; property specification requirements that do not refer to trace properties (e.g. indistinguishability in Table 1) are not covered in the property specification language of ASLan, namely LTL; composed services and policies cannot be modeled in ASLan v.1. Future versions of ASLan will encompass these features.

3.3 ASLan Syntax

We first present the entire ASLan v.1 language in BNF (with the usual conventions), and then give explanations and examples. The symbol for comments is %. The grammar has two start symbols, `Prelude` and `ASLanFile`. Correspondingly, we consider a fixed prelude file that contains all declarations that are service-independent, and, for each service, an ASLan-file that contains only service-specific declarations. An example of a prelude file is given in appendix A.

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

ASLanFile  ::=  SignatureSection
                TypesSection
                InitsSection
                HornClausesSection
                RulesSection
                GoalsSection

TypeSymbolsSection ::= "section typeSymbols:" Types
SignatureSection   ::= "section signature:"
                       ( SuperType | OpDecl )*
TypesSection       ::= "section types:" TypeDecl*
EquationsSection   ::= "section equations:" Equation*
InitsSection       ::= "section inits:"
                       ("initial_state" ConstId ":=" Facts)+
RulesSection       ::= "section rules:" Rule*
GoalsSection       ::= "section goals:" Goal*
IntruderSection    ::= "section intruder:"
                       (HornClause | Rule)*
HornClausesSection ::= "section hornClauses:" HornClause*

HornClause ::=  "hc" ConstId "(" Vars ")"
                ":=" Fact ":-" Fact ("," Fact)*
Rule ::= "step" ConstId "(" Vars ")"
         ":=" PNFacts Conditions ExistsVars? "=>" Facts
Facts   ::= Fact ("." Fact)*
Conditions ::= ("&" Condition)*
```

```
Condition ::=  AtomicCondition | "not(" AtomicCondition ")"
AtomicCondition ::=  "equal(" Term "," Term ")"
                     | "leq(" Term "," Term ")"

% PNFact stands for Possibly Negated Fact
PNFacts ::= PNFact ("." PNfact)*
PNFact ::= Fact | "not(" Fact ")"
% A Fact must be a Term of type 'fact'
Fact ::= Term
ExistsVars ::= "=[exists" Vars "]"

Goal ::= LTLGoal | AttackStates

LTLGoal ::= "goal" ConstId "(" Vars ")" ":=" Formula
AttackStates ::= "attack_state" ConstId "(" Vars ")" ":="
                 PNFacts Conditions

Formula ::= Fact |
            "equal(" Term "," Term ")" |
            "not" "(" Formula ")" |
            "and" "(" Formula "," Formula ")" |
            "or" "(" Formula "," Formula ")" |
            "implies" "(" Formula "," Formula ")" |
            "forall" Var . Formula |
            "exists" Var . Formula |
            LTLOp1 "(" Formula ")" |
            LTLOp2 "(" Formula "," Formula ")"

% neXt | Yesterday | Finally | Once | Globally | Historically
LTLOp1 ::= "X" | "Y" | "F" | "O" | "G" | "H"
% Until | Release | Since
LTLOp2 ::= "U" | "R" | "S"

Equation ::= Term "=" Term
% The number and types of Terms must match
% the declared arity of OpId
Term ::= Const | VarId | OpId "(" Terms ")"

% In TypeDecl of the form "VarId : Type" Type cannot be 'fact'
TypeDecl ::= (ConstId | VarId) ":" Type
Type ::= TypeId | OpId "(" Types ")" | "{" Consts "}"
```

```
SuperType ::= TypeId ">" TypeId
OpDecl ::= OpId ":" TypeStar "->" Type
TypeStar ::= Type | Type "*" TypeStar

Var := VarId
Vars ::= Var ("," Var )*
Terms ::= Term ("," Term )*
Types ::= Type ("," Type )*

OpId ::= ConstId
TypeId ::= ConstId

VarId ::= [A-Z_][a-zA-Z0-9_]*
ConstId ::= [a-z][a-zA-Z0-9_]*
Numeral ::= [1-9][0-9]*
Const ::= Numeral | ConstId
Consts ::= Const ("," Const )*
```

An ASLan specification (and, similarly, a prelude file) consists of a sequence of sections.

Section Type Symbols (`TypeSymbolsSection`). In this section, all basic (message) types are declared, for example `nonce`.

In the sections signature (`SignatureSection`) and types (`TypesSection`), the types of variables, constants, function symbols and fact symbols are declared.

Section Signature (`SignatureSection`). This section contains declarations of the used function and fact symbols, and, more specifically, their types. It also contains supertype declarations.

Section Types (`TypesSection`). In this section, the types for all constants and variables can be specified. This implies that throughout an ASLan file an identifier cannot be used with two different types (while the scope of each variable is limited to the rule it appears in).

Notice that a declaration of the form $\mathtt{M} : op(t_1, ..., t_n)$ is equivalent to the declarations $\mathtt{M_1} : t_1, \ldots, \mathtt{M_n} : t_n$ where $\mathtt{M}_i$ (with $i = 1, \ldots, n$) are fresh variables (i.e. that do not appear in the ASLan file) and every occurrence of `M` in the ASLan file is replaced with the term $op(\mathtt{M_1}, \ldots, \mathtt{M_n})$.

Section Equations (`EquationsSection`). The equations contained in this section define the algebraic properties of the function symbols.

The sections inits (`InitsSection`), rules (`RulesSection`), and intruder (`IntruderSection`) describe the service as a transition system, and the section goals (`GoalsSection`) describes the security goals or the attack states.

Section Inits (`InitsSection`). In this section, we specify one or more initial states of the service.

Section Rules (`RulesSection`). This section specifies the transition rules of the honest agents executing the service.

For the declaration of rules, we also use the following syntactic sugar. We assume that the `iknows` fact (`iknows(M)` means that the intruder knows message `M`) is persistent, in the sense that if an `iknows` fact holds in a state, then it holds in the successor states (i.e. the intruder never forgets messages). To simplify the rules, however, we do not write the `iknows` facts that already appears in the left-hand side of the rules.

Also, a rule can be labeled with a list of existentially quantified variables. Their purpose is to introduce new constants representing fresh data (e.g. nonces).

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

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

Section Intruder (`IntruderSection`). The rules and Horn clauses in this section describe the abilities of the intruder, namely composition and decomposition of known messages. As these abilities are again independent of the service, they are included in the prelude.

All used identifiers must be different from the ASLan keywords (`step`, `section`, `intruder`, `equal`, `leq`, `not`, `state`). The identifiers for types (`ASLan_Type`) used in declarations can only be those identifiers that have been introduced as type identifiers in the prelude. Identifiers for operators (`ASLan_Op`) are only those that have been declared in the signature section of the prelude as having range type message. Similarly, fact symbols (`ASLan_Fact`) are only the ones declared in the signature section of the prelude or the ASLan file as having range type fact. The identifiers that name

initial states, rules, or goals must be unique and distinct from all constants and variables and declared identifiers.

For a rule declaration, the variables in the variable list must contain exactly those variables that occur in the LHS of the rule. The variables of the RHS must be a subset of the variables in the positive facts of the LHS (excluding those variables that occur only in the conditions or the negative facts of the rule) and the existentially quantified variables. Analogous restrictions apply for initial states. Further, variables cannot occur in an initial state as it can be seen as the RHS of a rewrite rule with an empty LHS.

For a Horn clause declaration, the variable list must contain exactly those variables that occur in the clause. The variables of the fact on the left-hand side of the "`:-`" must be a subset of the variables in the facts of the right-hand side.

3.4 ASLan Semantics

All terms in ASLan are interpreted in the quotient algebra $\mathcal{T}_\Sigma/_E$, where E is the set of algebraic equations declared in the prelude specification. Thus, in the following, we consider two terms as equal if and only if this is a consequence of the algebraic equations. To distinguish from syntactical equivalence of terms, we write $t \approx s$ for two equivalent terms t and s. Also, we assume in the following that the type declarations of the ASLan file are satisfied in all substitutions of variables, e.g. variables of type agent are only substituted for constants (or other terms) of type agent.

Let $\mathcal{F}$ be the set of ground (i.e. variable free) facts, i.e. expressions of type `Fact`. An ASLan specification defines a transition system

$$M = \langle \mathcal{S}, \mathcal{I}, \rightarrow \rangle \ ,$$

where $\mathcal{S}$ is the set of states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, and $\rightarrow \subseteq S \times S$ is the transition relation. We assume that the states in $\mathcal{S}$ are represented by sets of ground facts, i.e. $\mathcal{S} = 2^{\mathcal{F}}$. If $S \in \mathcal{S}$, then we interpret the ground facts in S as the propositions holding in that state, all other ground facts being false (closed-world assumption). $\mathcal{I}$ is defined by the `InitsSection` in the obvious way. Let H be the set of Horn clauses defined in the section `HornClausesSection`. Let C be a conjunction of possibly negated atomic conditions. If C is ground then we say that C *holds* if and only if *(i)* C is of the form `equal`(t_1, t_2) and $t_1 \approx t_2$, *(ii)* C is of the form `leq`(t_1, t_2) where the ground terms t_1 and t_2 evaluate as expected to yield $n_1, n_2 \in \mathbb{N}$ and $n_1 \leq n_2$ holds, *(iii)* C is of the form `not`(C) and C does not hold, and *(iv)* C is of the form `and`(c_1, c_2) and both c_1 and c_2 hold. The transition relation $\rightarrow$ is

defined as follows: $S \rightarrow S'$ if and only if there exists a rule

$$PF \,.\, NF \; C \; -[V]\Rightarrow \; R$$

in the section `RulesSection` (where PF and NF are sets of facts and negated facts respectively and C is a conjunction of possibly negated atomic conditions) and a substitution $\sigma : \{v_1, \ldots, v_n\} \rightarrow T_\Sigma$, where $v_1, \ldots, v_n$ are the variables that occur only in PF, such that

1. $PF\sigma \subseteq S$,
2. $NF\sigma\sigma' \cap S = \emptyset$ for all substitutions σ' such that $NF\sigma\sigma'$ is ground.
3. $C\sigma\sigma'$ holds for all substitutions σ' such that $C\sigma\sigma'$ is ground.
4. $S' = \{f \in \mathcal{F} \mid S \setminus PF\sigma, R\sigma\sigma'', H \models f\}$, where σ'' is any substitution such that $v\sigma''$ does not occur in S or in any algebraic equation for all $v \in V$ and $\models$ is the consequence relation in classical propositional logic.

We remark that the closure of a state under the Horn clauses H (as implied by 4) yields in general an infinite set of facts. It is however immediate that we can always obtain a finite representation of every reachable state, by using the closure under H only implicitly.

A simple way to describe safety properties of a transition system is by defining a subset of so-called *bad* states or *attack* states. The attack states specification in ASLan is syntactically similar to a rule, only that there is no right-hand side. The declaration of an attack state A amounts to adding a rule $A \Rightarrow A.attack$ for a nullary fact symbol *attack* and defining every state that contains *attack* to be an attack state.

Another way to specify in ASLan the trust and security properties is by using LTL goals. To this end, we first consider the following definitions. A *path* π is a sequence of states $S_0 S_1 \ldots$ such that $S_i \rightarrow S_{i+1}$ for $i = 0, 1, \ldots$. We define $\pi(i) = S_i$ for all $i \in \mathbb{N}$. If $S_0 \subseteq \mathcal{I}$, then we say that the path is *initialized*. Let π be an initialized path of M and $\sigma : \mathcal{V} \rightarrow T_\Sigma$. An LTL formula ϕ *is valid on* π *under* σ, in symbols $\pi \models_\sigma \phi$, if and only if $(\pi, 0) \models_\sigma \phi$, where $(\pi, i) \models_\sigma \phi$ is inductively defined as follows:

if ϕ is a fact, then $(\pi, i) \models_\sigma \phi$ iff $\phi\sigma \in \pi(i)$

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

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

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

4 Example ASLan specifications

Policies are expressed formally in ASLan by means of policy facts and Horn clauses that encode the rules for managing and deriving trust assertions. A number of different logic-based trust management languages have been proposed in recent years in order to capture rights that are not necessarily assigned and maintained by central authorities, but may instead result from credentials issued by many different entities, so that authorization policies must handle not only permissions for the end user, but also permissions for entities to issue credentials, where these permissions themselves may depend on other credentials. The most interesting (from the point of view of AVANTSSAR) such languages are Delegation Logic [21,22], Binder [17], SecPAL [13], and DKAL [18], which all exploit Datalog, albeit at different levels, so as to allow the user to write high-level policy rules in a human-readable form while maintaining polynomial time decidability. We follow a similar approach: several of the example formal specifications of the case studies that we have been developing make use of a DKAL-like formal language to express and reason about policies. We will report on these examples in more detail in future deliverables and publications, but below we give excerpts of some of our specifications as an illustration of the capabilities of ASLan v.1 and of the formal specification discipline that we are pursuing.

In order to specify the authorization logic used by the honest agents in ASLan, the user must thus define the predicates and functions that govern the logic in the `signature` section, and the deduction system of the logic in terms of Horn clauses in the section `HornClausesSection`. For this specification to provide the basis for an automated analysis, which is the ultimate goal of the AVANTSSAR Validation Platform that we are developing, the modeler should also specify an interface between the workflow of the service under consideration and the policy level as formalized by the logic. To that end, we have been pursuing different approaches, e.g. [8, 9, 10], which share the goal of providing a framework that clearly separates service and policy into a dynamic part that evolves according to actions performed by agents and a static part, and thereby separate the enforcement of authorization policies and the workflow of the services/applications. Here the difficulty lies in the formalization of the interface functionalities that allow the policy level and the workflow level to interact in a principled way so as to enable the specification of the dynamic behavior of trusted and distributed systems. For instance, we must specify how to map concrete messages exchanged in the workflow by the agents executing the service to facts in the policy language expressing the authorization logic, and vice versa. We believe that ASLan v.1 provides a basis for this and, as remarked above, we will report on this

in much more detail soon.

Three case studies from Deliverable D5.1 (Problem Cases and their Trust and Security Requirements) [4], namely the public bidding portal, the loan origination protocol, and the car registration office are specified in ASLan in Sections 4.1, 4.2 and 4.3. These demonstrate the features of the language, and its capabilities in specifying security and trust aspects of service oriented architectures.

4.1 Example 1: Public bidding protocol with DKAL policies (an excerpt)

The public bidding service has been described in detail in Deliverable D5.1 (Problem Cases and their Trust and Security Requirements) [4]. In this section, we specify the "evaluation" phase of this service in ASLan, while highlighting the policies of individual processes.

Description of the evaluation phase As a first step in the evaluation phase, following the submission of a proposal, the security server (SS) validates the signature of the proposal. The technical merits of bids are assessed by the technical committee (TC). Based on the rating provided by TC, and financial offers of the bidders, the bid manager (BM) decides the winner of the bid. In the following ASLan formalization, we abstract away the BM decision criteria. We instead model it as a non-deterministic choice between the two participating bidders. The winner is announced by the bidding portal (BP). In the following, these steps are described:

1. $BP \rightarrow all : \{sid\}_{K_{BP}^{-1}}$
2. $bidder \rightarrow BP : \{\{bidder, \{FP\}_{K_{BM}}, \{TP\}_{K_{TC}}, sid\}_{K_{bidder}^{-1}}\}_{K_{BP}}$
3. $BP \rightarrow SS : \{\{bidder, \{FP\}_{K_{BM}}, \{TP\}_{K_{TC}}, sid\}_{K_{bidder}^{-1}}\}_{K_{BP-SS}}$
4. $SS \rightarrow BP : \{bidder, \{FP\}_{K_{BM}}, \{TP\}_{K_{TC}}, sid\}_{K_{BP-SS}}$
5. $BP \rightarrow BM : \{bidder, \{FP\}_{K_{BM}}, \{TP\}_{K_{TC}}, sid\}_{K_{BP-BM}}$
6. $BM \rightarrow BP : \{\{TP\}_{K_{TC}}\}_{K_{BP-BM}}$
7. $BP \rightarrow TC : \{\{TP\}_{K_{TC}}\}_{K_{BP-TC}}$
8. $TC \rightarrow BP : \{\{TP\}_{K_{TC}}, \mathsf{rating}\}_{K_{BP-TC}}$
9. $BP \rightarrow BM : \{\{TP\}_{K_{TC}}, \mathsf{rating}\}_{K_{BP-BM}}$

---

10. $BM \rightarrow BP : \{winner, sid\}_{K_{BP-BM}}$
11. $BP \rightarrow bidder : \{winner, sid\}_{K_{BP}^{-1}}$

Here, *sid* is the session identifier for a specific call for tender, while FP and TP are respectively the financial and the technical proposal of *bidder*.

We write K_X for the public key of participant X, and K_{X-Y} is a symmetric key shared between agents X and Y. Note that in the last phase (below the horizontal line), BM decides after receiving ratings of all the technical proposals.

The policy of the bid manager is the following: The bid manager, once sure that a proposal carries a genuine signature, decides whether the bidder is eligible or not. If yes, BM sends back a message to BP, who will let the TC check the TP of the bidder. Note that TC does not see the financial proposal of the bidder. Then, based on the ratings that BM receives from the TC, via BP, it decides the merits of each bidder's proposal, and finally announces a winner. In DKAL terms [19]:

```
BM knows TC tdon rated(x,tp)
BM knows SS tdon sigOk(p)
```

Where `tdon` stands for *trusted on*, and `rated(x,tp)` is a predicate, stating that the technical proposal tp, which bidder x puts forward, has been rated. Similarly, `sigOk` is a predicate, stating that the signature on proposal p is valid.

We also adopt the *trust application* rule of DKAL:

$$\frac{\text{A knows p tdon x} \quad \text{A knows p said x to A}}{\text{A knows x}}$$

This derivation rule is implemented via Horn clauses in our specification. How an assertion such as "p said x" is made will be explained below.

Similarly, the bidders trust BM on announcing the winner:

```
Alice knows BM tdon winner(x)
Bob knows BM tdon winner(x)
```

Where `winner` is a predicate, indicating the winner party in the bid. Since the winner is announced by BP, BM needs to delegate this trust to BP. We assume this delegation has happened before the evaluation phase starts (see the `init` part in the specification):

```
Alice knows BM said BP tdon winner(x) to Alice
Bob knows BM said BP tdon winner(x) to Bob
```

Then, we need to adopt the *trust delegation* rule of DKAL as well:

$$\frac{\text{A knows p tdon x} \quad \text{A knows p said q tdon x to A}}{\text{A knows q tdon x}}$$

This derivation rule is implemented via Horn clauses in our specification.

As a way to map concrete messages to policies we assume the following: If A receives a message containing m signed by B (while A has the public key of B), then A knows B said m to every one. Moreover, if A sees a message m encrypted by symmetric key K_{A-B}, then A knows that B said m to A, after checking that A itself has not sent that message. In the following specification, we leave out this check for conciseness.

```
section signature:

  % Bidder's signature
  % step number in the protocol - nat
  % himself - agent
  % BP - agent
  % financial proposal - message
  % technical proposal - message
  % his public key - public_key
  % BP's public key - public_key
  % BM's public key - public_key
  % TC's public_key - public_key
  % session ID - nat

  state_bidder : nat * agent * agent * message * message
                     * public_key * public_key * public_key
                     * public_key * nat -> fact

  % BP's signature
  % step number in the protocol - nat
  % himself - agent
  % bidder - agent
  % SS - agent
  % BM - agent
  % TC - agent
  % BP_sbmtd_prop - set
  % his public key - public_key
  % bidder's public key - public key
  % BP-SS shared key - symmetric_key
  % BP-BM shared key - symmetric_key
  % BP-TC shared key - symmetric_key
  % session ID - nat

  state_BP : nat * agent * agent * agent * agent * agent
```

```
          * set * public_key * public_key * symmetric_key
          * symmetric_key * symmetric_key * nat -> fact

% SS's signature
% himself - agent
% BP - agent
% PB-SS shared key - symmetric_key

state_SS : agent * agent * symmetric_key -> fact

% BM's signature
% step number in the protocol - nat
% himself - agent
% BP - agent
% SS - agent
% TC - agent
% bidder - agent
% BM_rcvd_prop - set
% his public key - public_key
% PB-BM shared key - symmetric_key
% session ID - nat

state_BM : nat * agent * agent * agent * agent * agent
           * set * public_key * symmetric_key * nat -> fact

% TC's signature
% step number in the protocol - nat
% himself - agent
% BP - agent
% BM - agent
% bidder - agent
% technical proposal - message
% his public key - public_key
% PB-TC shared key - symmetric_key

state_TC : nat * agent * agent * agent * agent
           * message * public_key * symmetric_key -> fact

% (DKAL like) predicates and functions

knows  : agent * message -> fact
```

```
  trusts : agent * agent * message -> fact
  tdOn   : agent * agent * message -> message
  saysTo : agent * agent * message -> message

  proposal : agent * message * crypt(public_key,message)
             -> message
  % bidder * financial proposal * technical
  % proposal encrypted with TC's public key
  sigOk    : message -> message
  eligible : agent -> message
  rated    : agent * message -> message
  % bidder * technical proposal
  winner   : agent -> message

section types:

  % Constants for agents
  % ua stands for unknown agent
  a,b,bp,bm,ss,tc,i,ua: agent
  % Variables for agents
  A,A1,BP,BM,SS,TC: agent

  % Constants for public keys
  % uk stands for unknown key
  ka,kb,kbp,kbm,ktc,ki,uk: public_key
  % Variables for public keys
  KA,KBP,KBM,KTC: public_key

  % Constants for symmetric keys
  kbp-ss,kbp-bm,kbp-tc: symmetric_key
  % Variables for symmetric keys
  KBP-SS,KBP-BM,KBP-TC: symmetric_key

  % Constants and variables for financial and technical
  % proposals. utp stands for unknown technical proposal.
  FP,TP,FP1,TP1,fpa,tpa,fpb,tpb,utp: message

  % Naturals
  0,1,2,3,4,5,6,7: nat

  % Constants and variables for session IDs
```

```
  % usid stands for unknown sid
  SID,usid,sid: nat

  % Sets
  % agents public keys (kept by SS)
  SS_keyset: set
  % proposals submitted to the BP
  BP_sbmtd_prop, sbmtd_prop: set
  % proposals received by BM after SS validation
  BM_rcvd_prop, rcvd_prop: set

  % Variable for the rating/evaluation of technical proposals
  RAT: message

  % Fact asserting the winning bidder has not
  % been announced yet
  notAnnounced: fact

section inits:

  initial_state init_1 :=

    % Local states
    state_bidder(0,a,bp,fpa,tpa,ka,kbp,kbm,ktc,usid).
    state_bidder(0,b,bp,fpb,tpb,kb,kbp,kbm,ktc,usid).
    state_BP(0,bp,ua,ss,bm,tc,sbmtd_prop,kbp,uk,kbp-ss,kbp-bm,
             kbp-tc,usid).
    state_SS(ss,bp,kbp-ss).
    state_BM(0,bm,bp,ss,tc,rcvd_prop,ua,kbm,kbp-bm,sid).
    state_TC(0,tc,bp,bm,ua,utp,ktc,kbp-tc).

    % the intruder knows:
    % himself, his public key and its converse
    iknows(i).iknows(ki).iknows(inv(ki))
    % all agents
    iknows(a).iknows(bp).iknows(ss).iknows(bm).iknows(tc).
    % and all their public keys
    iknows(ka).iknows(kbp).iknows(kbm).iknows(ktc).

    % the winning bidder has yet to be announced
    notAnnounced.
```

```
% principals' facts

% a
% trusts bm on announcing the winner or delegating
% somebody for it
trusts(a,bm,winner(a)).
trusts(a,bm,winner(b)).
knows(a,saysTo(bm,a,tdOn(a,bp,winner(a)))).
knows(a,saysTo(bm,a,tdOn(a,bp,winner(b)))).

% b
% trusts bm on announcing the winner or delegating
% somebody for it
trusts(b,bm,winner(a)).
trusts(b,bm,winner(b)).
knows(b,saysTo(bm,b,tdOn(a,bp,winner(a)))).
knows(b,saysTo(bm,b,tdOn(b,bp,winner(b)))).

% bp
% trusts bm on announcing the winner
trusts(bp,bm,winner(a)).
trusts(bp,bm,winner(b)).

% ss
% knows a, b and i and the respective public keys
contains(pair(a,ka),SS_keyset).
contains(pair(b,kb),SS_keyset).
contains(pair(i,ki),SS_keyset).

% bm
% trusts ss on the validation of a proposal's signature
% and bp on forwarding this message from ss
trusts(bm,bp,saysTo(ss,bp,
             sigOk(proposal(a,fpa,crypt(ktc,tpa))))).
trusts(bm,ss,sigOk(proposal(a,fpa,crypt(ktc,tpa)))).
trusts(bm,bp,saysTo(ss,bp,
             sigOk(proposal(b,fpb,crypt(ktc,tpb))))).
trusts(bm,ss,sigOk(proposal(b,fpb,crypt(ktc,tpb)))).

% trusts tc on the evaluation of a technical proposal
```

```
    % and bp on forwarding this message from tc
    trusts(bm,bp,saysTo(tc,bp,rated(a,crypt(ktc,tpa)))).
    trusts(bm,tc,rated(a,crypt(ktc,tpa))).
    trusts(bm,bp,saysTo(tc,bp,rated(b,crypt(ktc,tpb)))).
    trusts(bm,tc,rated(b,crypt(ktc,tpb))).

    % knows who's eligible
    knows(bm,eligible(a)).
    knows(bm,eligible(b)).

    % tc
    % trusts bm on who's eligible
    % and bp on forwarding this message from bm
    trusts(tc,bp,saysTo(bm,bp,eligible(a))).
    trusts(tc,bm,eligible(a))
    trusts(tc,bp,saysTo(bm,bp,eligible(b))).
    trusts(tc,bm,eligible(b))

section Hornclauses:

  % if A trusts B on saying M, and B says it,
  % then A knows M
  hc trustapp (A,B,M) :=
    knows(A,M) :- trusts(A,B,M),knows(A,saysTo(B,A,M))

  % if A trusts B on M, and B says A can trust D on M,
  % then A trusts D on M
  hc trustdel (A,B,D,M) :=
    trusts(A,D,M) :-
      trusts(A,B,M),knows(A,saysTo(B,A,tdOn(A,D,M)))

section rules:

  % BP decides and circulates the session ID for this call
  % for tender two instances of the portal are created to
  % handle the bidding process, one for a and another for b

  step step_0 (BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID)
   :=
   state_BP(0,BP,ua,SS,BM,TC,KBP,uk,KBP-SS,KBP-BM,KBP-TC,usid)
     =[exists sid]=>
```

```
 state_BP(1,BP,ua,SS,BM,TC,KBP,uk,KBP-SS,KBP-BM,KBP-TC,sid).
 iknows(crypt(inv(KBP),sid))

% A submits his/her proposal to BP

step step_1 (A,BP,FP,TP,KA,KBP,KBM,KTC) :=
  state_bidder(0,A,BP,FP,TP,KA,KBP,KBM,KTC,usid).
  iknows(crypt(inv(KBP),sid))
    =>
  state_bidder(1,A,BP,FP,TC,KA,KBP,KBM,KTC,sid).
  iknows(crypt(KBP,crypt(inv(KA),
    pair(pair(A,pair(crypt(KBM,FP),crypt(KTC,TP))),sid))))

% BP receives A's proposal and forwards it to SS
% A's proposal is stored as submitted and a new instance of
% Bp is created to handle future submissions

step step_2 (BP,A,SS,BM,TC,FP,TP,FP1,TP1,BP_sbmtd_prop,
  KBP,KBM,KTC,KA,KBP-SS,KBP-BM,KBP-TC,SID) :=
  state_BP(1,BP,ua,SS,BM,TC,BP_sbmtd_prop,KBP,uk,KBP-SS,
           KBP-BM,KBP-TC,SID).
  iknows(crypt(KBP,crypt(inv(KA),
    pair(pair(A,pair(crypt(KBM,FP),crypt(KTC,TP))),SID)))).
  not(contains(proposal(A,FP1,TP1),BP_sbmtd_prop))
    =>
  state_BP(2,BP,A,SS,BM,TC,BP_sbmtd_prop.KBP,KA,KBP-SS,
           KBP-BM,KBP-TC,SID).
  state_BP(1,BP,ua,SS,BM,TC,BP_sbmtd_prop,KBP,uk,KBP-SS,
           KBP-BM,KBP-TC,SID).
  iknows(scrypt(KBP-SS,crypt(inv(KA),
    pair(pair(A,pair(crypt(KBM,FP),crypt(KTC,TP))),SID)))).
  contains(proposal(A,crypt(KBM,FP),crypt(KTC,TP)),
           BP_sbmtd_prop)

% SS receives A's proposal from BP and sends it back only
% if the signature on it is valid

step step_3 (SS,BP,A,FP,TP,SS_keyset,KA,KBM,KTC,KPB-SS,SID)
 := state_SS(0,SS,BP,KBP-SS).
  iknows(scrypt(KBP-SS,crypt(inv(KA),
    pair(pair(A,pair(crypt(KBM,FP),crypt(KTC,TP))),SID))))
```

```
  contains(pair(A,KA),SS_keyset)
    =>
  state_SS(0,SS,BP,KBP-SS).
  iknows(scrypt(KBP-SS,pair(A,pair(pair(crypt(KBM,FP),
         crypt(KTC,TP))),SID)))

% BP receives the validated proposal from SS and forwards it
% to BM

step step_4 (BP,SS,A,BM,TC,FP,TP,KBP,KA,KBM,KTC,KBP-SS,
             KBP-BM,KBP-TC,SID) :=
  state_BP(2,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  iknows(scrypt(KBP-SS,pair(A,pair(pair(crypt(KBM,FP),
         crypt(KTC,TP))),SID)))
    =>
  state_BP(3,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  iknows(scrypt(KBP-BM,pair(pair(A,pair(crypt(KBM,FP),
         crypt(KTC,TP))),SID)))

% BM receives the proposal from BP and infers
% SS' validation the proposal is accepted if no
% other proposals were received already from the
% same bidder. Besides, a new instance of BM is
% creating for handling further submissions

step step_5 (BM,BP,SS,TC,A,FP,TP,FP1,TP1,BM_rcvd_prop,KBM,
             KTC,KBP-BM,SID) :=
  state_BM(0,BM,BP,SS,TC,ua,BM_rcvd_prop,KBM,KBP-BM,usid).
  iknows(scrypt(KBP-BM,pair(pair(A,pair(crypt(KBM,FP),
         crypt(KTC,TP))),SID))).
  not(contains(proposal(A,FP1,TP1),Rcvd_prop))
    =>
  state_BM(1,BM,BP,SS,TC,A,BM_rcvd_prop,KBM,KBP-BM,SID).
  state_BM(0,BM,BP,SS,TC,ua,BM_rcvd_prop,KBM,KBP-BM,usid).
  knows(BM,saysTo(BP,BM,saysTo(SS,BP,sigOk(proposal(A,FP,
        crypt(KTC,TP)))))).
  contains(proposal(A,FP,crypt(KTC,TP)),BM_rcvd_prop)

% BM evaluates if A is eligible, in which case sends the TP
% to the BP
```

```
step step_6 (BM,BP,SS,TC,A,BM_rcvd_prop,FP,TP,KBM,KTC,
             KBP-BM,SID) :=
  state_BM(1,BM,BP,SS,TC,A,BM_rcvd_prop,KBM,KBP-BM,SID).
  knows(BM,eligible(A)).knows(BM,sigOk(proposal(A,FP,
        crypt(KTC,TP))))
    =>
  state_BM(2,BM,BP,SS,TC,A,BM_rcvd_prop,KBM,KBP-BM,SID).
  iknows(scrypt(KBP-BM,pair(A,crypt(KTC,TP)))).
  knows(BM,eligible(A))

% BP receives A's technical proposal from BM and forwards it
% to TC

step step_7 (BP,BM,A,SS,TC,TP,KBP,KA,KTC,KBP-SS,KBP-BM,
             KBP-TC,SID) :=
  state_BP(3,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  iknows(scrypt(KBP-BM,pair(A,crypt(KTC,TP))))
    =>
  state_BP(4,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  iknows(scrypt(KBP-TC,pair(A,crypt(KTC,TP))))

% TC receives A's technical proposal from BP and infers A is
% eligible

step step_8 (TC,BP,BM,A,TP,KTC,KBP-TC)
  state_TC(0,TC,BP,BM,ua,utp,KTC,KBP-TC).
  iknows(scrypt(KBP-TC,pair(A,crypt(KTC,TP))))
    =>
  state_TC(1,TC,BP,BM,A,TP,KTC,KBP-TC).
  knows(TC,saysTo(BP,TC,saysTo(BP,BM,eligible(A))))

% TC evaluates A's technical proposal if A is eligible, then
% sends it to BP

step step_9 (TC,BP,BM,A,TP,RAT,KTC,KBP-TC)
  state_TC(1,TC,BP,A,TP,KTC,KBP-TC).
  knows(TC,eligible(A))
    =[exists RAT]=>
  state_TC(0,TC,BP,BM,ua,utp,KTC,KBP-TC).
  iknows(scrypt(KBP-TC,pair(A,pair(TP,RAT))))
```

```
% BP receives A's technical evaluation from TC and sends it
% to BM

step step_10 (BP,BM,A,SS,TC,TP,RAT,KBP,KA,KTC,KBP-SS,KBP-BM,
              KBP-TC,SID) :=
  state_BP(4,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  iknows(scrypt(KBP-TC,pair(A,pair(TP,RAT))))
    =>
  state_BP(5,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  iknows(scrypt(KBP-BM,pair(A,pair(TP,RAT))))

% BM receives A's technical evaluation from BP and checks
% it corresponds to the TP before sent

step step_11 (BM,BP,A,TP,FP1,TP1,BM_rcvd_prop,RAT,KBM,KTC,
              KBP-BM,SID) :=
  state_BM(2,BM,BP,SS,TC,A,BM_rcvd_prop,KBM,KBP-BM,SID).
  iknows(scrypt(KBP-BM,pair(A,pair(TP,RAT)))).
  contains(proposal(A,FP1,crypt(KTC,TP1)),BM_rcvd_prop) &
  equal(crypt(KTC,TP),crypt(KTC,TP1))
    =>
  state_BM(3,BM,BP,SS,TC,A,BM_rcvd_prop,KBM,KBP-BM,SID).
  knows(BM,saysTo(BP,BM,saysTo(TC,BP,rated(A,
        crypt(KTC,TP)))))

% When BM has received all technical reports, it decides who
% wins (nondeterministically here) and sends the decision to
% BP

step step_12 (BM,BP,SS,TC,A,FP,TP,FP1,TP1,BM_rcvd_prop,KBM,
              KTC,KBP-BM,SID) :=
  state_BM(3,BM,BP,SS,TC,A,BM_rcvd_prop,KBM,KBP-BM,SID).
  contains(proposal(a,FP,crypt(KTC,TP)),BM_rcvd_prop).
           rated(a,crypt(KTC,TP))
  contains(proposal(b,FP1,crypt(KTC,TP1)),BM_rcvd_prop).
           rated(b,crypt(KTC,TP1))
  notAnnounced
    =>
  state_BM(4,BM,BP,SS,TC,A,BM_rcvd_prop,KBM,KBP-BM,SID).
  iknows(scrypt(KBP-BM,pair(A,SID)))
```

```
% BP receives the winner

step step_12 (BP,BM,A,SS,TC,TP,KBP,KA,KTC,KBP-SS,KBP-BM,
              KBP-TC,SID) :=
  state_BP(5,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  iknows(scrypt(KBP-BM,pair(A,SID)))
    =>
  state_BP(6,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  knows(BP,saysTo(BM,BP,winner(A)))

% BP announces the winner to everybody

step step_13 (BP,BM,A,SS,TC,TP,KBP,KA,KTC,KBP-SS,KBP-BM,
              KBP-TC,SID) :=
  state_BP(6,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  knows(BP,winner(A))
    =>
  state_BP(7,BP,A,SS,BM,TC,KBP,KA,KBP-SS,KBP-BM,KBP-TC,SID).
  iknows(crypt(inv(KBP),pair(A,SID)))

% A receives the result and infers A1 is the winner

step step_14 (A,A1,BP,FP,TP,KA,KBP,KBM,KTC,SID) :=
  state_bidder(1,A,BP,FP,TC,KA,KBP,KBM,KTC,SID).
  iknows(crypt(inv(KBP),pair(A1,SID))))
    =>
  state_bidder(2,A,BP,FP,TC,KA,KBP,KBM,KTC,SID).
  knows(A,saysTo(BP,A,winner(A1)))
```

In the specification above, we used fact symbols, such as `state_BP` and `state_BM`, for representing the local state of an agent in role BP or BM. The format of such `state` terms is service-dependent (therefore it is explicitly declared in the signature section): they allow us to represent all information relevant for participating in one session of the public bidding scheme, e.g. the step number of the last executed protocol step, the names of the involved agents, their public keys, a session identifier, etc. Initially, the slots for the session identifier are filled with the dummy value `usid` which expresses that no session identifier has been created or received yet.

Step compression The rules incorporate an optimization often called *step-compression*. Step-compression is based on the idea that we can iden-

tify the intruder and the network: every message sent by an honest agent is received by the intruder and every message received by an honest agent comes from the intruder. Formally, we compose (or "compress") several steps: when the intruder sends a message, an agent reacts to it according to his rules, and the intruder diverts immediately the agent's answer. A bisimulation proof [11] shows that the model with such composed actions (which we present here) is "attack-equivalent" to the model with single (uncompressed) transitions (i.e. we end up in an attack state using composed transitions iff that was the case using uncomposed transitions).

Regarding the policies We remark that certain policy facts are not explicitly derived in the specification. For instance, if b is announced as the winner, $\texttt{knows}(\texttt{a}, \texttt{winner}(\texttt{b}))$ is derived only via using trust application and trust delegation rules encoded in Horn clauses. When Alice receives the message $\texttt{crypt}(\texttt{inv}(\texttt{kbp}), \texttt{pair}(\texttt{b}, \texttt{sid}))$ in `step_14`, she adds the fact $\texttt{knows}(\texttt{a}, \texttt{saysto}(\texttt{pb}, \texttt{a}, \texttt{winner}(\texttt{b})))$ to the next state. The closure of this state, with respect to the Horn clauses `trustapp` and `trustdel` will then contain $\texttt{knows}(\texttt{a}, \texttt{winner}(\texttt{b}))$.

Security requirements

Below, we specify the security requirements of the Public Bidding case study, as described in Deliverable D 5.1, using LTL. Our formalization of the public bidding process focuses on the evaluation phase, hence for security properties pertaining to other phases only generic examples are given.

Authentication. The Public Bidding scenario assumes that all authentications between the Bidding Portal and other principals (both bidders and back-ends) happen prior to the bidding process, therefore no authentication appears in the ASLan formalization above. In general, though, we might have rewrite rules as follows:

$$\texttt{LHS} \Rightarrow \texttt{RHS.authenticate}(\texttt{a}, \texttt{b}, \texttt{Cert})$$

$$\texttt{LHS} \Rightarrow \texttt{RHS.authenticated}(\texttt{a}, \texttt{b}, \texttt{Cert})$$

where the fact $\texttt{authenticate}(a,b,Cert)$ indicates the message (the certificate *Cert*) sent by a in order to authenticate with b. The success of the authentication is acknowledged by the fact $\texttt{authenticated}(a,b,Cert)$. Subsequently an authentication[1] requirement of a to b could be expressed in LTL as follows:

$$G(\texttt{authenticated}(\texttt{a}, \texttt{b}, \texttt{Cert}) \Rightarrow O\ \texttt{authenticate}(\texttt{a}, \texttt{b}, \texttt{Cert}))$$

[1] We here consider only one variant of the possible kinds of authentication see [23].

meaning that for each trace of the system, a state of the trace where a is authenticated with b is preceded by a prior state where a sent *Cert* to b, to this end. Note that the facts `authenticate` and `authenticated` must be introduced in the Signature section of the ASLan file (being protocol-dependent).

Authorization. As well as for authentication, we can keep track of authorization in the same fashion by adding to the RHS of appropriate rewrite rules the facts `allowed`*(a,Action)* and `authorize`*(b,a,Action)* stating, respectively, that agent a is allowed to perform the action *Action*, and that agent b authorizes a on doing *Action*. Afterwards, authorization constraints can be expressed in LTL as follows:

$$G(\texttt{allowed}(\texttt{a}, \texttt{Action}) \Rightarrow O\ \texttt{authorize}(\texttt{b}, \texttt{a}, \texttt{Action}))$$

meaning that if a state of the system allows a to perform *Action*, in a prior state agent b authorized a to do so. The above formula takes into account static authorization, i.e. once granted is never revoked, but we could handle dynamic authorization as well by introducing a further persistent fact `revoked`*(a,Action)* (authorization for a to perform *Action* is revoked). The new LTL formula would then be:

$$G(\texttt{allowed}(\texttt{a}, \texttt{Action}) \Rightarrow (O\ \texttt{authorize}(\texttt{b}, \texttt{a}, \texttt{Action})) \wedge \neg\texttt{revoked}(\texttt{a}, \texttt{Action}))$$

As well as for the authentication example the auxiliary facts (here `authorize`, `revoked` and `allowed`) must be first defined in the Signature section of the ASLan file. An instance of authorization appears in the Public Bidding scheme when the *BM* decides which proposals should be admitted for technical evaluation. This is handled by introducing the fact `eligible`*(a)*, used to assert a static authorization on a's proposal to be evaluated. There the correspondent authorization constraint in LTL would be

$$G(\texttt{iknows}(\texttt{scrypt}(\texttt{k}_{\texttt{BP}-\texttt{BM}}, \texttt{pair}(\texttt{a}, \texttt{crypt}(\texttt{k}_{\texttt{TC}}, \texttt{tp}))))) \Rightarrow O\ \texttt{knows}(\texttt{bm}, \texttt{eligible}(\texttt{a})))$$

expressing the requirement that the technical proposal of bidder a is sent out to the *TC* for evaluation only if a was deemed eligible.

Integrity. Formalization of integrity requirements can be achieved using the fact `witness`*(a,b,var,val)*, defined in the Prelude file (see appendix A), having the meaning that agent a created a value *val* in place of the variable

var, for agent b. With respect to the Public Bidding scenario as formalized in the ASLan file, we could for instance require integrity of the proposal sent to the *SS* via the following two formulae:

$$
\begin{aligned}
&G(\texttt{iknows}(\texttt{crypt}(\texttt{k}_{\texttt{BP}},\\
&\texttt{crypt}(\texttt{inv}(\texttt{k}_{\texttt{A}}),\texttt{pair}(\texttt{pair}(\texttt{a},\texttt{pair}(\texttt{crypt}(\texttt{k}_{\texttt{BM}},\texttt{fp}),\texttt{crypt}(\texttt{k}_{\texttt{TC}},\texttt{tp}))),\texttt{sid}))))\\
&\Rightarrow\\
&O\ \texttt{witness}(\texttt{a},\texttt{bp},\texttt{proposal},\\
&\texttt{crypt}(\texttt{inv}(\texttt{k}_{\texttt{A}}),\texttt{pair}(\texttt{pair}(\texttt{a},\texttt{pair}(\texttt{crypt}(\texttt{k}_{\texttt{BM}},\texttt{fp}),\texttt{crypt}(\texttt{k}_{\texttt{TC}},\texttt{tp}))),\texttt{sid}))))
\end{aligned}
$$

$$
\begin{aligned}
&G(\texttt{iknows}(\texttt{scrypt}(\texttt{k}_{\texttt{BP-SS}},\\
&\texttt{crypt}(\texttt{inv}(\texttt{k}_{\texttt{A}}),\texttt{pair}(\texttt{pair}(\texttt{a},\texttt{pair}(\texttt{crypt}(\texttt{k}_{\texttt{BM}},\texttt{fp}),\texttt{crypt}(\texttt{k}_{\texttt{TC}},\texttt{tp}))),\texttt{sid}))))\\
&\Rightarrow\\
&O(\texttt{witness}(\texttt{bp},\texttt{ss},\texttt{proposal},\\
&\texttt{crypt}(\texttt{inv}(\texttt{k}_{\texttt{A}}),\texttt{pair}(\texttt{pair}(\texttt{a},\texttt{pair}(\texttt{crypt}(\texttt{k}_{\texttt{BM}},\texttt{fp}),\texttt{crypt}(\texttt{K}_{\texttt{TC}},\texttt{tp}))),\texttt{sid})))\\
&\wedge\ O\ \texttt{witness}(\texttt{a},\texttt{bp},\texttt{proposal},\\
&\texttt{crypt}(\texttt{inv}(\texttt{k}_{\texttt{A}}),\texttt{pair}(\texttt{pair}(\texttt{a},\texttt{pair}(\texttt{crypt}(\texttt{k}_{\texttt{BM}},\texttt{fp}),\texttt{crypt}(\texttt{k}_{\texttt{TC}},\texttt{tp}))),\texttt{sid}))))
\end{aligned}
$$

The first formula expresses the requirement that when the *Bidding Portal* receives a's proposal (corresponding, due to step compression, to the intruder knowing the message encrypted with *BP*'s public key), sometime in the past a created the same proposal for *BP*. The second formula asserts that not only when the *SS* receives a's proposal from *BP* (i.e. the intruder knows the proposal encrypted with *BP-SS* shared key) there must be a previous state s where *BP* created the message containing the proposal for *SS*, but also that in a state s', prior to s, a created the proposal for *BP*. While the first formula aims at detecting when a message has been modified in between the transmission, the second one additionally verifies that when the sender has the role of intermediary (as is the case here for *BP*) he/she does not send a different proposal from what a submitted.

Fairness. The Public Bidding scenario introduces also some fairness constraints, such as "The Technical Committee must not omit to evaluate an eligible tender, i.e. the list of evaluated tenders must match the eligible list submitted by the BM". In our formalization of the problem this corresponds to receiving an evaluation from the *TC* for each *technical proposal* sent to it by the *BM*; in LTL:

$$
\begin{aligned}
&G(\texttt{state}_{\texttt{TC}}(1,\texttt{tc},\texttt{bp},\texttt{a},\texttt{tp},\texttt{k}_{\texttt{TC}},\texttt{k}_{\texttt{BP-TC}})\ \wedge\ \texttt{knows}(\texttt{tc},\texttt{eligible}(\texttt{a}))\\
&\Rightarrow\\
&F\ \texttt{iknows}(\texttt{scrypt}(\texttt{k}_{\texttt{BP-TC}},\texttt{pair}(\texttt{a},\texttt{pair}(\texttt{tp},\texttt{rating})))))
\end{aligned}
$$

The above formula asserts that for each state of the system where TC has received a's technical proposal tp and knows a is an eligible bidder, eventually another state will follow where TC sends out its evaluation of tp. In the same fashion more fairness constraints can be specified, e.g. "All bidders must have a proof of receipt of their tender".

Non-repudiation. A common requirement in document exchange procedures in general is non-repudiation, aiming at preventing a party from disclaiming its actions to invalidate the process. Non-repudiation could easily be formalized by means of the following formula:

$$G(\texttt{owns}(\texttt{b}, \texttt{crypt}(\texttt{k}_\texttt{A}, \texttt{m})) \Rightarrow O\ \texttt{witness}(\texttt{a}, \texttt{b}, \texttt{msg}, \texttt{crypt}(\texttt{k}_\texttt{A}, \texttt{m})))$$

that uses a new fact predicate `owns`*(b,msg)*, which means agent b "owns" the message *msg*) and the previously introduced fact `witness`. This formula intuitively asserts that for every state in which b owns the message `crypt`(k_A,m), some time in the past a has created that message for b.

4.2 Example 2: LOP in ASLan (an excerpt)

The loan origination business process (LOP), one of the core processes of a banking scenario, has been described in Deliverable D5.1 (Problem Cases and their Trust and Security Requirements) [4]. It formalizes a bank's evaluation of a customer's request for a loan. Besides the customer, two organizations are involved: the bank, and the credit bureau. The customer's request for a loan triggers an instance of the LOP at the bank which requires some services offered by the credit bureau in order to decide whether to grant the loan or not.

Organizations rely on process engines to execute all the instances of the business processes required to accomplish their business goals. A process engine orchestrates all the tasks of the business process instances under execution according to the flow specified at designed time. Process engines outsource to task engines the duty of assigning and dispatching the human and automated tasks of a business process instance among the available users and services respectively. In general, a task engine handles the entire task lifecycle. For instance, the bank's process engine will make sure that, for each instance of the LOP, the task Check Credit Worthiness is executed just after the task Customer Identification.[2] The assignment and dispatchment

[2]See the workflow presented in Figure 1 of Deliverable D5.1 (Problem Cases and their Trust and Security Requirements) [4].

of these two tasks to the proper pre-processing and post-processing clerks is handled by the bank's task engine associated to the bank's process engine.

Clearly, many security and trust aspects of a business process are tightly linked to both the process and task engines used to execute the business process itself. For instance, a naive task engine might grant a task to an unauthorized user. In order to validate the security and trust requirements of business processes, it is thus necessary to model the security and trust relevant aspects of these two engines. The ASLan has to be expressive enough to model all these aspects. In the rest of this section, we show how the ASLan specification language allows for modeling the static separation of duty (SSOD in short) check, a critical check that a task engine has to perform before granting a task to a user.

Modeling the static separation of duty check. Suppose that a certain human task of the LOP has to be executed and that a certain bank clerk is willing to do that. The bank clerk will send this request to the task engine that will grant the task provided that certain conditions hold. Among these conditions, static separation of duty has to be checked to mitigate the risk of fraudulent actions. SSOD constraints are specified at design time as part of the policy associated to the organizations and to the business processes executed there. LTL and horn clauses allow us to easily model various aspects of SSOD.

To illustrate we will use the ASLan facts summarized with their informal meaning in Table 2. In our setting, SSOD constraints are then fulfilled if and only if condition (1) holds.

$$\forall \texttt{ssod}(t_1, t_2)(\texttt{ex}(\texttt{task}(t_1, ti_1, bpi), u_1) \wedge \texttt{ex}(\texttt{task}(t_2, ti_2, bpi), u_2) \Rightarrow u_1 \neq u_2) \quad (1)$$

This condition should be checked by the task engine before granting a task to a bank clerk that is requesting for it. While modeling condition (1) in IF is costly and involved (it requires the introduction of many other fictitious facts and rules), the addition of Horn clauses allow us to do that smoothly by simply adding one new fact and the following horn clause:

```
hc task_assignment_ssod (T_1,U,T_2,TI_1,TI_2,BPI) :=
  denied(task(T_1,TI_1,BPI),U) :- ssod(T_1,T_2),
                                  ex(task(T_2,TI_2,BPI),U)
```

where the new fact $\texttt{denied}(\texttt{task}(t, ti, bpi), u)$ holds any time a user u is not entitled, because of SSOD constraints, to perform the task instance ti of task t in the execution of the business process instance bpi.

Table 2: Facts and their informal meaning

Fact	Holds when
state_$r(j, a, e_1, \ldots, e_k, s)$	Active entity a playing function r is ready to execute step j in session s. $e_1, \ldots, e_k$ is a list of expressions representing part of a's internal state.
task_status(t, ts)	The status of task t is ts, that is whether task t has been assigned to a user (e.g., clerk), or not.
msg(se, rc, d, ch)	Active entity se has sent data d to active entity rc on the communication channel ch.
ex(task$(t, ti, bpi), u)$	Task instance ti of task t has been executed by user (e.g., clerk) u during the run of the business process instance bpi.
ssod(t_1, t_2)	There is a SSOD constraint between tasks t_1 and t_2, which means they should not be executed by the same user.
contains(db, d)	Data d is contained in set db.

It is now immediate to capture the entire task assignment upon user request into the ASLan rules depicted in Figure 1. More precisely, the left-hand-side of the rule models the task engine TE receiving a message sent by the bank clerk U that desires to perform task T, one of those tasks that is still in the task todo-list TL of the task engine and that is ready to be executed, but not assigned yet. Provided that no SSOD constraint is violated, the task engine will grant to the clerk the execution of the task as captured by the message fact occurring in the right-hand-side. The actual execution of the task by the clerk can then be done. A standard ASLan rule can easily capture this clerk execution action, and add the corresponding fact ex(task(T, TI, BPI), U) to the state.

It is worth pointing out that the denied fact can be used to capture security aspects of task assignment other than SSOD. For instance, adding the following horn clause would prevent a clerk to execute a task for which he does not have the proper rights:

```
hc task_assignment_ac (T,U,TI,BPI,R,U) :=
  denied(task(T,TI,BPI),U) :- user_to_role(U,R),
                              role_to_forbiddentask(R,T)
```

where user_to_role and role_to_forbiddentask are facts used to map,

```
step task_engine_manages_task_request(TE,TL,S,T,TI,BPI,U,Ch) :=
  state_taskEngine(1,TE,TL,S).
  contains(task(T,TI,BPI),TL).
  task_status(task(T,TI,BPI),ready_notassigned).
  msg(U,TE,claim(T,U),Ch).
  not(denied(T,U))                          %% SSOD check
=>
  task_status(task(T,TI,BPI),ready_assigned).
  msg(TE,U,proceed(task(T,TI,BPI)),Ch).
  state_taskEngine(1,TE,TL,S).
  contains(task(T,TI,BPI),TL)
```

Figure 1: Task assignment upon clerk request evaluated with respect to SSOD constraints.

respectively, users to roles and roles to those tasks that are forbidden for them.

It is also interesting to consider how LTL allows us to undertake a formal analysis of the business process, under the hypothesis that the task engine is properly enforcing SSOD. This hypothesis can be expressed in LTL as the following:

$$C_{SSOD} := \mathbf{G}\,\forall \mathtt{ssod(T1,T2)} \Rightarrow ((\mathtt{ex(task(T1,TI1,BPI),U1)} \wedge \mathtt{ex(task(T2,TI2,BPI),U2)}) \Rightarrow \mathtt{U1} \neq \mathtt{U2}) \quad (2)$$

Then, one can use C_{SSOD} as a fairness constraint for the properties that have to be checked: If G is the security property, one can check $C_{SSOD} \Rightarrow G$, i.e. G is checked only on those traces for which C_{SSOD} holds.

To summarize, ASLan allows us to easily model certain security and trust relevant aspects of business process management systems that would be difficult to be captured otherwise.

4.3 Example 3: Car Registration in ASLan (an excerpt)

The car registration scenario has been described in Deliverable D5.1 (Problem Cases and their Trust and Security Requirements) [4]. For the sake of conciseness, we simplify this scenario as follows.

A **citizen**, called Mike, submits a request to register his new car to an **employee**, called Peter, of the local **car registration office**.[3] Mike's message contains all the documents to support his request and it is suitably

[3]We have abstracted away the mechanism assigning a citizen request to a certain em-

signed. Upon reception of the request, Peter has appropriate support for checking the signature of the document and compare it with the identity of the sender of the request; if the signature and the identity of the requester do not match, then the request is immediately refused and the sender is acknowledged of this fact. Otherwise, Peter starts to consider the content of the request for the car registration: if, according to some criteria (which are abstracted away in the specification), the request is not suitably supported by the documents, then the request is refused and, again, the sender is acknowledged of this fact. Otherwise, the request is accepted, the sender is acknowledged of acceptance and the request is marked as accepted, signed by Peter, and finally sent to the **central repository** to be archived.

This process is completely transparent to Mike and, in order to be successfully completed, Peter should have the right to store documents in the central repository. The right to store documents in the central repository can only be granted by the **head of the car registration office**, a (special) employee called Melinda. Upon reception of the request by Peter to store a processed request in its internal database, the central repository checks whether Peter has been granted the right to do so. If the case, the central repository stores the document; otherwise, it refuses to comply.

Roles are distributed to employees (of the registration office) by circulating appropriate certificates; such as, e.g., "Peter is an employee" or "Melinda is the head of the car registration office." These certificates are emitted by a **certification authority** which is recognized by the employees of the car registration office and the central repository. Permission to store documents in the central repository are also distributed to employees by creating appropriate certificates; however, these certificates are created by the head of the car registration office (not by the certification authority).

The central repository, before storing a processed request in its internal database, checks whether the employee has the right to do so. For this to be successfully executed, the following policy should be enforced:

- an employee of the car registration office can store documents in the central repository, if the head of the car registration office permits it,

and the following trust relationships should have preliminarily been established:

- the certification authority of the car registration office is trusted by all employees, by the head of the car registration office, and the central repository concerning role certificates; and

ployee of the car registration scenario.

- the head of the car registration office is trusted by the central repository for action (e.g., storing documents) certificates.

Finally, to be able to successfully execute the scenario with Mike and Peter described above, the following certificates should be available in the system:

- Peter is an employee of the car registration office (by a certificate emitted by `regOffCA`),
- Melinda is the head of the car registration office (by a certificate emitted by `regOffCA`), and
- Melinda permits Peter to store documents in the central repository (by a certificate emitted by Melinda).

We now discuss some of our modeling decisions. Since only the exchange of messages drives the workflow of the system (which is otherwise almost stateless, except for the modifications of the database in the central repository) and the most interesting part of the case study concerns its policies, we adopt a very abstract model for messages and their exchange. A message is simply a record with three fields: sender, body, and receiver.[4] The model for exchanging messages is also very abstract: the network is modeled by a set to which a message is added (when sending) or it is tested for membership (when receiving). Messages are never deleted. Although simple, this model allows us to specify safety properties. It is well-known that many security properties can be reduced to this class of (temporal) properties. Formally, the set of messages is represented by the unary predicate `iknows` below. Since both citizens and employees should be able to sign documents and the latter should also be able to check signatures, appropriate primitives to generate signatures (`sign`), attaching them to documents (`augdocwithsign`), and checking that the signature attached to a document belongs to a certain principal (`matchuser`) are provided. An employee has also the primitive to attach a decision (`accept` or `refuse`) to a document containing a citizen request (`augdocwithact`). Finally, as role certificates should be distributed over the network, we provide an appropriate primitive (`rolecert`) to create these documents. (Role certificates are handled at the policy level only; see below for more details.) The following ASLan prelude details our model for the communication and (part of the) workflow in the system.

[4] In other words, we have abstracted away all the details about the cryptographic primitives used to prepare documents to "safely" travel over the network.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prelude ASLan %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

section typeSymbols:

 message, agent, body, doc, signature, action, role,
 decision, fact

section signature:

 message > agent
 message > body
 message > doc
 message > signature
 message > action
 message > role
 message > decision

% In this specification, messages are very abstractly
% described as records composed of three fields: sender,
% body (carrying the content of the message), and receiver.
            msg : agent * body * agent -> message
% Predicate modelling the communication at the workflow
% level, i.e. it is the state of the network
         iknows : message              -> fact
% Any document (e.g., a request sent by a citizen or the
% processed request by an employee) should be suitably
% processed so as to become the body of a message
       embeddoc : doc                  -> body
% Primitives for creating the signature of a document and
% attaching it to the document so as to create a new document
% (e.g., requests by citizens should be signed or requests
% processed by employees should also be signed)
           sign : agent * doc          -> signature
 augdocwithsign : doc * signature      -> doc
% Primitive to attach a decision (either accept or refuse)
% to a request by a citizen (used by an employee encharged
% to handle the request)
  augdocwithact : doc * action         -> doc
% 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
% Primitive to check whether a signed document is signed
% by a certain agent
     matchuser : doc * agent           -> fact
% Recognizer for agents (required to circumvent the
% constraint that variables in the hypotheses of a
% Horn rule should be a subset of the variables
% occurring in the head of the rule)
      is_agent : agent                 -> fact
% Recognizer for pieces of information exchanged at the
% policy level (this is introduced to circumvent some
% limitations in writing Horns rules, namely that Horn
% rules without hypotheses are not allowed)
 is_piece_of_info : message            -> fact


section types:

% Standard documents explaining that a request has been
% refused or accepted
  refusedoc : doc
  acceptdoc : doc
% Flags saying that a request has been refused or accepted
     refuse : decision
     accept : decision
% Possible actions about a document with respect to the
% central repository: reading or storing
    readdoc : action
   storedoc : action
% Possible roles of employees
   employee : role
       head : role

% Identifiers for the central repository and the car
% certification authority of the car registration office
   centrRep : agent
   regOffCA : agent

% Variables
```

```
   AnyThing : message
     AnyOne : agent
       User : agent
         Id : agent
        Id1 : agent
        Id2 : agent
        Doc : doc
        Act : action
       Sign : signature
       Role : role

section equations:

% Checking that a document has been signed by a certain user
 matchUser(augmentdocwithsig(Doc,sign(User,Doc)),User) = true

% Recognizer for agents and pieces of information exchanged
% at the policy level
           is_agent(AnyOne) = true
 is_piece_of_info(AnyThing) = true

section intruder:

% This section has intentionally been left empty as it is
% not yet clear what exactly is an intruder (the high level
% style in specifying messages forbids the possibility to
% adopt the standard Dolev-Yao attacker model)
```

As we have said above, the workflow of the system is almost stateless. There are however two exceptions. One is the database of the central repository which is modeled by the unary predicate `dbdoc` to which documents may only be added (and never deleted). The other is the unary predicate `isok` which allows us to abstract away the criteria according to which a citizen request is accepted or refused. This completes the description of the (static part of the) workflow.

We now switch to describe the policy level of the system. We adapted the DKAL [19] approach to specifying policies in ASLan. To this end, we have introduced predicates (`knows` and `knows0`) to represent the knowledge of the various agents and predicates (`saysTo` and `saysTo0`) for the *communication* between agents. It is important to observe the differences between the communication at the workflow and the policy levels of the system. The

former (modeled via `iknows`) is state-full and thus modeled by an appropriate set of step rules of ASLan (see below). The latter (modeled via `saysTo` or `saysTo0`) is stateless and thus modeled by suitable Horn clauses. Finally, DKAL proposes two functions (`tdOn` or `tdOn0`) to track trust relationships between agents concerning certain facts. All this is formally captured in the following piece of ASLan specification.

```
% Specification of the Car Registration Office in D5.1

section signature:

% Policy predicates and functions (DKAL like):
% A piece of information (represented as an element of
% type message) is known to an agent without involving
% communication at the policy level but possibly at the
% workflow level.  Typically (see, e.g., the step rules
% named GetRoleCertEmployee and GetRoleCertHead below),
% this kind of knowledge is acquired upon reception of a
% message over the network.
     knows0 : agent * message         -> fact
% A piece of information (represented as an element of
% type message) is known to an agent possibly involving
% communication at the policy level (see the saysTo0 and
% saysTo predicates below).
      knows : agent * message         -> fact
% Agents may communicate parts of their knowledge at the
% policy level to other principals (this is a different kind
% of communication with respect to communication at the
% workflow level).  It is worth noticing that the policy
% level communication is targeted; e.g., saysTo(a1,x,a2)
% means that agent a1 says a piece of information x to agent
% a2. Notice that the piece of information x in
% saysTo(a1,x,a2) is part of the knowledge of agent a1 that
% can be obtained from some other agent a3.  Instead, if we
% write saysTo0(a1,x,a2), then the piece of information x is
% part of the "internal" knowledge of agent a1, i.e. it is
% not obtained by other some other principal but only by
% some computation, e.g. reception of a message over the
% network at the workflow level.
      saysTo : agent * message * agent -> fact
     saysTo0 : agent * message * agent -> 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 a piece
% of information i to an agent a2, then a2 knows that a1
% said the piece of information i.
       said : agent * message -> message
% 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
% policy level.
      said0 : agent * message -> message
% The function symbol tdOn and tdOn0 encode trust
% relationships between agents concerning some piece of
% information.
       tdOn : agent * message -> message
% 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.
      tdOn0 : agent * message -> message

% input predicate (for employees): we abstracted away how an
% employee decides whether a request is correct or not
       isok : Doc -> fact
% state predicate (for centrRep): state of the central
% repository, i.e. represents the set of processed requests
      dbdoc : Doc -> fact

% Bridge between the workflow and the policy levels
% Policy level constructs corresponding to role certificates
% (the first two) and action certificates (the last one)
        isRegOffEmpl : agent -> message
        isRegOffHead : agent -> message
canstoredocincentrRep : agent -> message

section types:
       P, Q : agent
       Empl : agent
```

```
       Empl1 : agent
     Citizen : agent
        Head : agent

% Identifiers for agents in scenario in D5.1: mike (citizen),
% peter (employee), and melinda (head)
        mike : agent
       peter : agent
     melinda : agent

% Identifier for the request sent by mike (cf. scenario D5.1)
      doc5_1 : doc
```

We are now ready to describe the workflow, the policies, and their interplay in the system. At the beginning, the network must contain Mike's request, the certificates about the roles of Peter and Melinda, and the workflow must specify that Mike's request is correct according to some criteria (abstracted by the predicate `isok`). At the policy level, the initial state should say that Melinda would like to grant Peter the right to store documents in the central repository (notice the use of `knows0` to denote the fact that this piece of information belongs to the internal knowledge of Melinda).

```
section inits:

% Initial state for scenario in D5.1 %%%%%%%%%%%%%%%%%%%%%%

initial_state init :=

% A citizen, called Mike, sends a request to Peter, the
% employee of the car registration office (we abstracted
% away how requests from citizens are assigned to employees
%  of the office)
iknows(msg(mike,
           embeddoc(augdocwithsig(doc5_1,sign(mike,doc5_1))),
          peter)).
% The certification authority of the car registration office
% has generated a certificate about the role of Peter
% (employee of the car registration office)
iknows(msg(regOffCA,
    embeddoc(augdocwithsign(rolecert(peter,employee),
                     sign(regOffCA,rolecert(peter,employee)))),
```

```
           peter)).
% The certification authority of the car registration office
% has  generated a certificate about the role of Melinda
% (head of the car registration office)
iknows(msg(regOffCA,
    embeddoc(augdocwithsign(rolecert(melinda,head),
                      sign(regOffCA,rolecert(melinda,head)))),
           peter)).
% The content of the request sent by the citizen correctly
% supports his/her request
isok(augdocwithsig(doc5_1,sign(mike, doc5_1))).
% Melinda, the head of the car registration office, decides
% that her employee, Peter, has enough experience to store
% accepted requests in the central repository
knows0(melinda,canstoredocincentrRep(peter))
```

We now describe the policies of the system. First, we provide an (incomplete) characterization of the DKAL like predicates expressing knowledge and communication for policies (this is adapted from [19], to which the interested reader is pointed to for details).

```
section Hornclauses:

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

% Internal knowledge is knowledge
hc knowledge0inf(P,AnyThing) :=
 knows(P,AnyThing) :-
  knows0(P,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
% (say2know0) or not (say2knowinf).
hc say2know0(P,Q,AnyThing) :=
 knows0(P,said0(Q,AnyThing)) :-
  saysTo0(Q,AnyThing,P)
hc say2knowinf(P,Q,AnyThing) :=
 knows(P,said(Q,AnyThing)) :-
```

```
  saysTo(Q,AnyThing,P)

% An agent P knows a piece of information AnyThing whenever
% an agent P knows that another agent Q said the piece of
% information AnyThing and also that P knows that the agent Q
%  is trusted on saying the piece of information AnyThing
hc trustedknowledge0(P,Q,AnyThing) :=
 knows(P,AnyThing) :-
  knows(P,tdOn0(Q,AnyThing)),
  knows(P,said0(Q,AnyThing))
hc trustedknowledgeinf(P,Q,AnyThing) :=
 knows(P,AnyThing) :-
  knows(P,tdOn(Q,AnyThing)),
  knows(P,said(Q,AnyThing))
```

Then, we consider the policies of each agent. The first three Horn clauses specify the communication of (role and action) certificates at the policy level. (Notice that while the knowledge of an action certificate for Peter is explicitly given in the initial state above, the knowledge of the role certificates for Peter and Melinda will be lifted from the existence of the corresponding messages in the network by appropriate step rules; see `section rules:` below.) The last three Horn clauses are the formal counterparts of the trust relationships described above.

```
% Policies of each agent %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% (Simple) employee %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% An employee getting a certificate from the certification
% authority about some agent that can play the role of an
% employee in the car registration office is willing to tell
% this to anyone
hc Cert1(Empl,Cert,AnyOne) :=
 saysTo(Empl,said0(regOffCA,Cert),AnyOne) :-
  is_agent(AnyOne),
  knows(Empl,said0(regOffCA,Cert)).

% An employee getting a certificate from the head of the car
% registration office about some agent that is entitle to
% store documents in the central repository is willing to
% tell this to anyone
hc Cert2(Empl,Cert,AnyOne) :=
 saysTo(Empl,said0(Head,Cert),AnyOne) :-
```

```
  is_agent(AnyOne),
  knows(Empl,said0(Head,Cert))

% (Head) employee: the two rules above plus the following one
% 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
hc GenerateCert(Head,Empl,AnyOne) :=
 saysTo0(Head,canstoredocincentrRep(Empl),AnyOne) :-
  is_agent(AnyOne),
  knows0(Head,canstoredocincentrRep(Empl))

% Central Repository %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The central repository trusts the certification authority
% of the car registration office
hc centrRepTrustCA(AnyThing) :=
 knows(centrRep,tdOn0(regOffCA,AnyThing)) :-
  is_piece_of_info(AnyThing)

% The central repository trusts anyone when communicating a
% ceritificate emitted by the certification authority of
% the car registration office
hc centrRepTrustAnyoneViaCA(AnyOne,AnyThing) :=
 knows(centrRep,tdOn(AnyOne,said0(regOffCA,AnyThing))) :-
  is_agent(AnyOne),
  is_piece_of_info(AnyThing)

% 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 exists certificates proving that he/she
% is the head of a car registration office and that the
% subject of his/her certificate is an employee
hc centrRepTrustHead(Head,Empl) :=
 knows(centrRep,tdOn0(Head,canstoredocincentrRep(Empl))) :-
  knows(centrRep,said0(regOffCA,isRegOffHead(Head))),
  knows(centrRep,said0(regOffCA,isRegOffEmpl(Empl))

% The central repository trusts anyone presenting a
```

```
% certificate emitted by the head of a car registration
% office provided that there exists a certificate of the
% fact that the emitter is the head of the car registration
% office
hc centrRepTrustAnyoneViaHead(AnyOne,AnyThing) :=
 knows(centrRep,tdOn(AnyOne,said0(Head,AnyThing))) :-
  knows(centrRep,said0(regOffCA,isRegOffHead(Head))),
  is_agent(AnyOne),
  is_piece_of_info(AnyThing)
```

Finally, we give the step rules modeling the dynamics of the system. The first two step rules are part of the interface between the workflow and the policy levels of the system as they allow employees to convert the content of role certificate received from the network to (internal) knowledge which is relevant for the application of policies (compare the right hand sides of these rules with the hypotheses of the Horn clauses `Cert1` and `Cert2`). The following three step rules specify the processing of a citizen request by an employee. The last step rule (`Storedoc`) describes how the central repository handles the request of an employee to store a document in its internal database. This is (the remaining) part of the interface between the workflow and the policy level: the guard

```
knows(centrRep,canstoredocincentrRep(Empl))
```

is a query which is possibly solved by the Horn clauses above.

```
section rules:

% (Simple) employee %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% An employee Empl1 asserts at the policy level that he/she
% has received a certificate about being an employee by (a
% possibly different) employee Empl
step GetRoleCertEmployee(Empl,Empl1) :=
 iknows(msg(regOffCA,
            embeddoc(augdocwithsign(rolecert(Empl,employee),
                     sign(regOffCA,rolecert(Empl,employee)))
                     ),
            Empl1))
 =>
 knows0(Empl1,said0(regOffCA,isRegOffEmpl(Empl)))
% REMARK: one may notice that it is possible to sharply
%         decouple workflow and policy levels as follows.
```

```
%          One may introduce an additional predicate
%          representing the role table of each employee, e.g.
%
%              hasrole : agent * message -> fact
%
%          The step rule above can be modified as follows:
%
%          iknows(msg(regOffCA,
%             embeddoc(augdocwithsign(rolecert(Empl,employee),
%                      sign(regOffCA,rolecert(Empl,employee)
%                      ))),
%             Empl1))
%          =>
%          hasrole(Empl1,isRegOffEmpl(Empl))
%
%          Furthermore, we should add the following Horn rule
%          at the policy level:
%
%          knows0(Empl1,said0(regOffCA,isRegOffEmpl(Empl))) :-
%           hasrole(Empl1,isRegOffEmpl(Empl))

% An employee Empl asserts at the policy level that he/she
% has received a certificate about being the head of a car
% registration office by (a possibly different) employee Head
step GetRoleCertHead(Empl,Head) :=
 iknows(msg(regOffCA,
            embeddoc(augdocwithsign(rolecert(Head,head),
                        sign(regOffCA,rolecert(Head,head)
              ))),
            Empl))
 =>
 knows0(Empl,said0(regOffCA,isRegOffHead(Head)))
% REMARK: a technique similar to the one sketched above would
%          allow one to clearly separate the workflow and the
%          policy level

% An employee Empl refuses a request Doc by a Citizen if the
% request is not signed by the Citizen sending it.  Rejection
% is acknowledged to the Citizen.
step Refuse1(Empl,Doc,Citizen) :=
 iknows(msg(Citizen,embeddoc(Doc),Empl)).
```

```
 not(matchUser(Doc,Citizen))
 =>
 iknows(msg(Empl,embeddoc(refusedoc),Citizen))

% An employee Empl refuses a request Doc by a Citizen if the
% request is correcly signed but some other criterium
% (abstracted away in this specification) is not met.
% Rejection is acknowledged to the Citizen.
step Refuse2(Empl,Doc,Citizen) :=
 iknows(msg(Citizen,embeddoc(Doc),Empl)).
 matchUser(Doc,Citizen).
 not(isok(Doc))
 =>
 iknows(msg(Empl,embeddoc(refusedoc),Citizen))

% An employee Empl accepts a request Doc by a Citizen if
% the request is correcly signed and all criteria are met.
% This means 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
step Accept(Empl,Doc,Citizen) :=
 iknows(msg(Citizen,embeddoc(Doc),Empl)).
 matchUser(Doc,Citizen).
 isok(Doc).
 =>
 iknows(msg(Empl,embeddoc(acceptdoc),Citizen)).
 iknows(msg(Empl,
        embeddoc(augdocwithact(
                  augdocwithsig(augdocwithdec(Doc,accept),
                        sign(Empl,
                              augdocwithdec(Doc,accept))),
                         storedoc)),
        centrRep))

% Central Repository %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Upon reception of a request to store a document in its
% data base (modelled by dbdoc), the central repository
% checks whether the employee asking for this to be done has
% the right to do this. If the case, the request is added to
% its data base
```

```
step Storedoc(Empl,Doc) :=
 iknows(msg(Empl,embeddoc(augdocwithact(Doc,storedoc)),
                                                centrRep)).
 knows(centrRep,canstoredocincentrRep(Empl))
 =>
 dbdoc(Doc)
```

While it is relatively easy to check that the scenario described above involving Peter and Melinda can be executed by a suitable sequence of step rules and solving appropriate queries against the policies of the system, we consider the following interesting property about the integrity of documents stored in the central repository:

Integrity: any processed request *preq* stored in the central repository must be consistent, i.e., it should be double signed (by the citizen *cit* submitting the request *req* and by the employee *empl* handling it) and stamped with the seal of acceptance.

Such a property can be written as the following safety formula in LTL:

$$G\left(\begin{array}{l}\texttt{dbdoc}(preq) \Rightarrow \exists cit, req, empl, preq_1, preq_2. \\ \qquad \left(\begin{array}{lcll} preq_1 & = & \texttt{augdocwithsig}(req, \texttt{sign}(user, req)) & \wedge \\ preq_2 & = & \texttt{augdocwithdec}(preq_1, \texttt{accept}) & \wedge \\ preq & = & \texttt{augdocwithsig}(preq_2, \texttt{sign}(empl, preq_2)) & \end{array}\right)\end{array}\right)$$

Showing that the system ensures integrity is non-trivial, as the central repository treats documents as black-boxes and trusts employees to check signatures and correctly prepare processed requests. Furthermore, it trusts the head of the central repository to judge the capability of employees to perform this job correctly. Ultimately, the central repository also trusts the certification authority to emit role certificates for both employees and the head of the car registration office.

5 Conclusion

ASLan v.1 is a simple, yet expressive, language for specifying security aspects of service oriented architectures. ASLan v.1 is based upon the Intermediate Format [7], extended with Horn clauses. This extension in particular enables us to specify authorization policies of a participant, along with its workflow. The flexibility and expressivity of ASLan v.1 is demonstrated via specifying three case studies from AVANTSSAR Deliverable D 5.1.

References

[1] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra Abad. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008)*. ACM Press, 2008.

[2] Alessandro Armando and Luca Compagna. SAT-based Model-Checking for Security Protocols Analysis. *International Journal of Information Security*, 7(1):3–32, 2008.

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

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

[5] AVANTSSAR. Deliverable 6.2.1: State-of-the-art on specification languages for service-oriented architectures. Available at http://www.avantssar.eu, 2008.

[6] Automated Validation of Internet Security Protocols and Applications. http://www.avispa-project.org/.

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

[8] Philippe Balbiani, Yannick Chevalier, and Marwa El Houri. A Logical Approach to Dynamic Role-Based Access Control. In *Artificial Intelligence: Methodology, Systems, and Applications, 13th International Conference, AIMSA 2008*, LNCS 5253, pages 194–208. Springer, 2008.

[9] Philippe Balbiani, Yannick Chevalier, and Marwa El Houri. An attribute based framework to express dynamic evolution of services in a distributed environment, Manuscript.

[10] Michael Barletta, Silvio Ranise, and Luca Viganò. Modeling the interplay of authorization policies and workflow in service-oriented architectures, Manuscript.

[11] D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In Einar Snekkenes and Dieter Gollmann, editors, *Proceedings of ESORICS'03*, LNCS 2808, pages 253–270. Springer-Verlag, 2003.

[12] David Basin, Paul Hankes Drielsma, Sebastian Mödersheim, and Luca Viganò. Pseudonym Invariance, 2008. Draft.

[13] Moritz Y. Becker, Cedric Fournet, and Andrew D. Gordon. Security Policy Assertion Language (SecPAL). http://research.microsoft.com/en-us/projects/SecPAL/.

[14] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Verified reference implementations of ws-security protocols. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 88–106. Springer, 2006.

[15] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Advances in Cryptology — Eurocrypt 2001*, LNCS 2045, pages 93–118. Springer Verlag, 2001.

[16] Jan Camenisch and Anna Lysyanskaya. A formal treatment of onion routing. In *Crypto*, 2005.

[17] John DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[18] Yuri Gurevich and Itay Neeman. Distributed-Knowledge Authorization Language (DKAL). http://research.microsoft.com/~gurevich/DKAL.htm.

[19] Yuri Gurevich and Itay Neeman. DKAL: Distributed-knowledge authorization language. In *Proceedings of CSF 2008*, pages 149–162. IEEE Computer Society, 2008.

[20] Paul Hankes Drielsma, Sebastian Mödersheim, Luca Viganò, and David Basin. Formalizing and Analyzing Sender Invariance. In *Proceedings of FAST 2006*, LNCS 4691. Springer, 2006.

[21] Ninghui Li. *Delegation Logic: A Logic-based Approach to Distributed Authorization*. PhD thesis, New York University, 2000.

[22] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, 2003.

[23] Gavin Lowe. A hierarchy of authentication specifications. *Computer Security Foundations Workshop, IEEE*, 0:31, 1997.

[24] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for ws-bpel. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.

[25] Sebastian Mödersheim. Extensions of Alice and Bob Notation, 2008. Draft.

[26] Oasis Consortium. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, 11 April, 2007.

A Prelude File

Here we give the entire specification of a typical ASLan prelude file, that reflects the abilities of the Dolev-Yao intruder.

```
% PRELUDE ASLan

section typeSymbols:

  agent, nonce, symmetric_key, public_key, function, set, table,
  nat, message, fact

section signature:

  message       > agent
  message       > nonce
  message       > symmetric_key
  message       > public_key
  message       > function
  message       > set
  message       > table

  pair     : message * message -> message
  crypt    : message * message -> message
  inv      : message -> message
  scrypt   : message * message -> message
  exp      : message * message -> message
  xor      : message * message -> message
  apply    : message * message -> message

  iknows   : message -> fact
  contains : message * message -> fact
  witness  : agent * agent * message * message -> fact
  request  : agent * agent * message * message -> fact
  secret   : message * message -> fact

section types:

  K,M,M1,M2,M3 : message

section equations:
```

```
  pair(M1,pair(M2,M3)) = pair(pair(M1,M2),M3)

  inv(inv(M)) = M

  exp(exp(M1,M2),M3) = exp(exp(M1,M3),M2)
  exp(exp(M1,M2),inv(M2)) = M1

  xor(M1,xor(M2,M3)) = xor(xor(M1,M2),M3)
  xor(M1,M2) = xor(M2,M1)
  xor(xor(M1,M1),M2) = M2


section intruder:

  % generate rules for intruder

  hc gen_pair (M1,M2) :=
     iknows(pair(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_crypt (M1,M2) :=
     iknows(crypt(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_scrypt (M1,M2) :=
     iknows(scrypt(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_exp (M1,M2) :=
     iknows(exp(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_xor (M1,M2) :=
     iknows(xor(M1,M2)) :- iknows(M1), iknows(M2)
  hc gen_apply (M1,M2) :=
     iknows(apply(M1,M2)) :- iknows(M1), iknows(M2)

  % analysis rules for intruder

  hc ana_pair1 (M1,M2) :=
     iknows(M1) :- iknows(pair(M1,M2))
  hc ana_pair2 (M1,M2) :=
     iknows(M2) :- iknows(pair(M1,M2))
  hc ana_crypt (M1,M2) :=
     iknows(M) :- iknows(crypt(K,M)), iknows(inv(K))
  hc ana_scrypt (M1,M2) :=
     iknows(M) :- iknows(scrypt(K,M)), iknows(K)
```

```
step generate (M) :=
  =[exists M]=> iknows(M)
```

The ability of the Dolev-Yao intruder to generate new constants of any type is covered in the last rule above (`generate`).