Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures

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

www.avantssar.eu

Deliverable D4.1
AVANTSSAR Validation Platform v.1

Abstract

This deliverable describes the development and prototypical implementation of the AVANTSSAR Validation Platform. More specifically, we describe here the architecture, the input and output languages, and the two main components of the platform: the Orchestrator and the Validator.

Deliverable details

Deliverable version: *v1.0* Classification: *public*
Date of delivery: *30.12.2009* Due on: *31.12.2009*
Editors: *UNIVR, ETH Zurich, INRIA, UPS-IRIT, UGDIST, IBM, IEAT (principal editors).*
OpenTrust, SAP, SIEMENS (secondary editors) Total pages: *73*

Project details

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

Contents

List of Tables

1 Introduction

This deliverable describes the development and prototype implementation of the AVANTSSAR Validation Platform (AVANTSSAR Platform, for short). A schema of the platform is presented in Figure 1. The platform takes as input a policy stating the functional and security requirements of a goal service and a description of the available services (including a specification of their security-relevant behavior, possibly including the local policies they satisfy) and aims at building an orchestration of the available services that meet the security requirements stated in the policy.

The main components of the platform are the TS Orchestrator and the TS Validator (Orchestrator and Validator, for short):

- The *Orchestrator* tries to build an orchestration, i.e. a composition, of the available services in a way that is expected (but not yet guaranteed) to satisfy the input policy. We call *mediator* the service that exposes the orchestration functionality to the outside world. In the case of dynamic composition of services, this orchestration is synthesized utilizing *TS Wrappers* (Wrappers, for short) that add security functionality not provided by the initial set of services. In § 3.4.5, we provide a Mock-up version of the *Wrappers*; we have a preliminary implementation and we will describe it in more detail in a future deliverable.

- The *Validator* automatically analyses the validation problem resulting from the Orchestrator output. Failed validation means the existence of vulnerabilities that need to be fixed; otherwise, the composition of the services is guaranteed to be secure, i.e. to meet the input policy.

In the final version of the platform, whenever the Validator detects a vulnerability on the composed service, a feedback loop to the Orchestrator is initiated. In this prototype version, as described in § 3.2, the Orchestrator does not support this feature, providing only one possible orchestration.

The schema of Figure 1 refines that given in the Description of Work as it explicitly includes a *connectors layer*, i.e. a layer of software modules that carry out the translation from application-level specification languages (e.g. BPEL) into the ASLan v.1 language (and vice versa). The ASLan v.1 language, which was defined in Deliverable D2.1 (“Requirements for modelling and ASLan v.1” [8]), is the input and output format of the logical level of the platform. In this deliverable, we focus on the logical level of the platform and also briefly touch upon the high-level languages that we plan to take into account at the application level. The aspects related to the

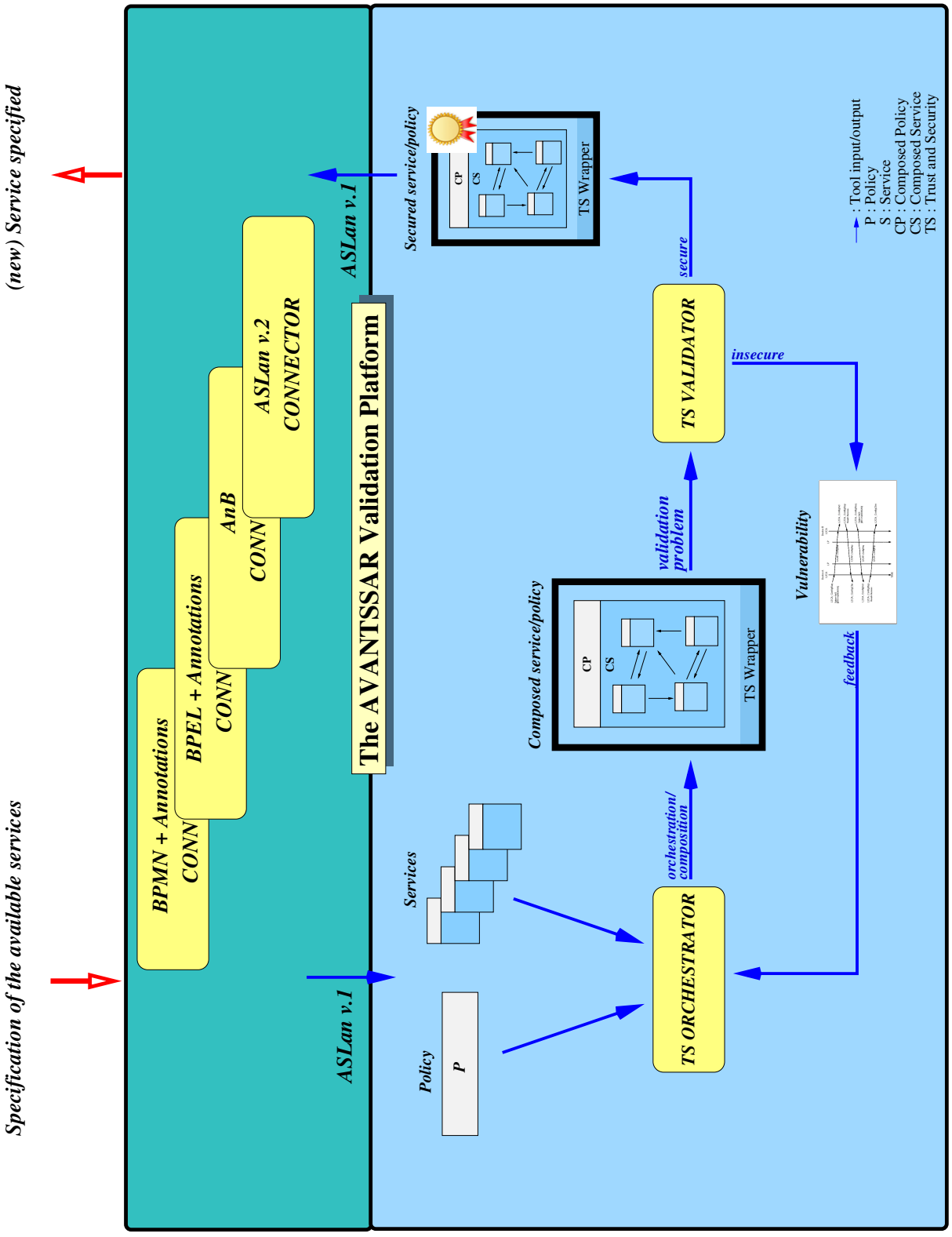

Figure 1: The AVANTSSAR Validation Platform

connectors layer (i.e. the translation from a high-level language into ASLan) will be described in future deliverables.

An input specification consists of:

- the *available web services* (i.e. services that the orchestrator can compose to reach its goal), and
- the *policy* stating both the functional and the security goals of the desired service.

The functional goals express functionalities that must be synthesized by composing the available services, while the security goals express the trust and security properties that must be satisfied by the synthesized service.

The output is a validated orchestration, i.e. a composed service that employs the available services and achieves both the functional and the security goals. In more detail:

- The Orchestrator looks for a combination of the available services meeting the functional goals. It may additionally receive as input a counterexample found by the validator, if any. The output of the Orchestrator is an ASLan specification of the goal service that is guaranteed to satisfy the functional goals.
- The Validator checks the security of the orchestration (i.e. the output of the Orchestrator) by checking if it satisfies the security goals. Whenever a counterexample is found, it is returned to the Orchestrator, otherwise the orchestration produced by the Orchestrator is returned as output.

ASLan specifications are amenable to formal analysis, but high-level languages are better suited for developers. The specifications provided to the logical level of the platform (and resulting from the validation and synthesis activities) need to be translated from (and to) the modelling artifacts and languages used at the application level, such as those described in Deliverable D6.2.1 ("State-of-the-art on specification languages for service-oriented architectures" [11]). The languages we consider at this stage are indicated in Figure 1: BPMN and BPEL, possibly annotated, ASLan v.2 [13], and the novel language AnB, based on an extended Alice-and-Bob notation [28]. These transformations implemented in the connectors layer will be explicitly addressed in the Industry Migration workpackage WP 6. In particular, a security-augmented BPMN plays a primary role as it is used in the migration to the industrial development environments of SAP and OpenTrust for the specification of the Industrially-Suited Specification Language (ISSL) as described in Deliverable D6.2.2 ("Industrial language requirements" [12]).

Moreover, the ASLan v.1 specification can also be obtained by translating a specification in ASLan v.2, as described in Deliverable 2.2 ("ASLan v.2 with static service and policy composition" [13]). An automatic translator from ASLan v.2 to ASLan v.1 is under development. Notice that other languages can be used in the AVANTSSAR Platform by defining their connectors to ASLan v.1.

Structure of the document. In § 2, we define the service orchestration problem and describe the common interface we use for specifying problems for the Orchestrator. In § 3 and § 4, the two main components of the platform are described: the Orchestrator and the Validator. In § 5, the AVANTSSAR Platform implemented as a service-oriented architecture is described. In § 6, we describe the experimental results obtained by running the platform against a selection of problem cases taken from Deliverable D5.1 [10] formally specified in ASLan. We conclude in § 7 by summing up and highlighting the main features that will be included in the next version of the platform.

2 Service Orchestration Problems in ASLan

Given a client expressed as a service C and a set of available services $\{S_1, \ldots, S_n\}$, the *service orchestration problem* amounts to building a Goal service G capable to answer to all requests coming from a client C using his (the Goal's) initial knowledge and, if needed, invocations of the available services. In other words, G must be such that the system $C|G|S_1|\ldots|S_n$ can reach (from its initial state) a state where the local state of C is final, where $|$ denotes parallel composition. A detailed account of the approach is given in [21].

We also consider a variation of the problem whereby an abstract characterization of the goal service G is given in place of C [14]. In this case, the orchestration problem amounts to finding a service G' that simulates G using the available services and a mediator for managing their communications.

2.1 ASLan format of the input

An input specification of an orchestration problem is an ASLan v.1 file, where some identifiers are reserved and given a special meaning (see § 2.2). A service is defined by some initial knowledge and its behavior (specified by a transition system with an initial state).

Here, we will only recall the main features of ASLan v.1, pointing the reader to Deliverable D2.1 ("Requirements for modelling and ASLan v.1" [8]) for more details on the language.

An ASLan file consists of several sections, among which:

Section Inits contains one or more initial states of the transition system. A state of a transition system is a set of variable-free facts.

Section Rules specifies the transitions (see example below) of the transition system.

A *transition* is a rule containing two parts, a left-hand side (LHS) and right-hand side (RHS). The rule can fire in a state whenever its LHS holds in that state. Moreover, a transition can be labeled with a list of existentially quantified variables whose purpose is to introduce new constants representing fresh data (e.g. nonces).

Example 1 *Sample transition.*

```
step sampleTransition(BankAgent):=
        state_BankingService(BankAgent, 1).
        iknows(request)
```

```
            =>
                state_BankingService(BankAgent, 2).
                iknows(response)
```

Here

step is a keyword used to define a new transition;

`sampleTransition` is a transition name;

`BankAgent` is a parameter of the transition;

`state_BankingService(BankAgent, 1)`, `iknows(request)`, `state_BankingService(BankAgent, 2)`, `iknows(response)` are facts;

`state_BankingService(BankAgent, 1).iknows(request)` is a LHS of the transition;

`state_BankingService(BankAgent, 2).iknows(response)` is a RHS of the transition.

This transition represents the behavior of a banking service that receives a `request` and then reacts by replying with `response` and moving to another state.

More precisely, the transition can be fired if there exists a value `val` of variable `BankAgent` such that `state_BankingService(val, 1)` and `iknows(request)` are in the current state. The result of firing the transition is to replace the fact `state_BankingService(val, 1)` by the fact `state_BankingService(val, 2)` and add a new fact `iknows(response)`.

We observe that fact `iknows(request)` of Example 1 will not disappear from the current state, because the predicate `iknows` is persistent: once a message is emitted, it becomes a part of the knowledge of the environment (i.e. of the network or of the intruder) and the environment does not "forget" it.

If the LHS of a transition holds in the current state, then it is supposed that the knowledge (represented by a set of ground facts) of the corresponding service is enough to build the messages stated in `iknows` in the RHS of the transition.

In order to specify service states, we distinguish one predicate per service. By convention the predicate name starts with `state_` followed by the service name, e.g. `state_BankingService` from Example 1.

Section Goals contains security goals that can be defined as attack states (special states of the transition system) or by means of LTL formulae.

Example 2 *Sample attack state.*

```
attack_state stateName(Msg):=
    fact1(Msg).
    fact2(Msg)
```

Here, attack state `stateName` is reached, if there exists a value `val` of variable `Msg` such that `fact1(val)` and `fact2(val)` are in the current state of the transition system.

Some attack states and LTL formulas (with reserved names) are dedicated to specify the orchestration problem (see § 2.2).

Section HornClauses contains a finite set of Horn clauses. They can specify, for instance, the authorization logic.

Two kinds of input specification are supported by the platform. In both cases, the available services are defined by their transition system.

The first option is to define a Client service, a service whose requests should all be satisfied. That is, the result of solving the orchestration problem is a service (Goal service) that is able to reply to every request of the Client. To this end it may use the available services. All the communications of the Goal service should be reflected in the output. See § 2.2 and § 2.2.1 for details on how to specify the input using this option.

The second option is to partially define a Goal service. One should only specify the part related to the communication with the putative client. As for the output, a new Goal service is issued, which is an extended Goal service given in the input with communication with available services. See § 2.2 and § 2.2.2 for details on how to specify the input using this option.

Note that all internal abilities (generation of new terms under some conditions, storage of data, etc.) are supposed to be designed as available services.

2.2 Specification of the orchestration problem

To specify orchestration problems, we need to distinguish at least such key parts as Goal service or Client service.

As a state of the Goal service we use

`state_OrchestrationGoal`(<list of parameters>);

similarly, we denote the state of the Client by the predicate:

`state_OrchestrationClient`(<list of parameters>).

A sample transition of the Client service is given in Example 3. Generally, a state of a service is represented by a predicate whose name starts with `state_` followed by a service name, e.g. `state_Service1`. (An example of an hypothetical banking service transition was given in Example 1.)

The orchestration problem is specified (besides the set of available services) with an LTL formula that must be satisfied by the computed orchestration. For example, to specify an orchestration where the emergency service is not invoked (the service stays in its initial state `state_Emergency(0)`) while the Client is in state `state_OrchestrationClient(A,99)` (for some `A`), we define an LTL formula:

```
goal orchestrationConstraint(A) :=
   F(and(state_OrchestrationClient(A,99),
      state_Emergency(0)))
```

The `orchestrationConstraint` word is a reserved keyword to express that the goal is to be processed by the Orchestrator, in contrast with the other (security) goals that are to be processed by the Validator.

Another way to specify an orchestration problem is to define a special attack state using the keyword `orchestrationFinalState`. The orchestration problem is solved when this state is reached. An example of such a specification is given in Example 3.

Another aspect that affects the orchestration is the knowledge of the Goal service. In fact the abilities of the Goal service directly depend on his knowledge: the credentials he possesses, the functions he can apply, etc. To define the initial knowledge, a Goal state predicate (`state_OrchestrationGoal`) is employed: the arguments placed in the initial state are considered as the initial knowledge of the Goal. The same holds for `state_OrchestrationClient` in the initial state for the Client.

As usual, message sending and receiving are specified using `iknows` facts: the `iknows` in the LHS of a transition stands for receiving a message, while in the RHS of a transition it stands for sending a message.

The full list of conventions is given in Appendix A.

2.2.1 Orchestration problem with a specified Client

In the case where a Client is given as input, one must provide the specification of

- the available services;

- the Client service;
- the initial knowledge of the Goal service;
- `attack_state orchestrationFinalState` and/or `goal orchestrationConstraint`.

For instance, to define the orchestration problem as a state to be reached, one can identify the final transition to be satisfied by the goal by writing `iknows(some_fresh_constant)` on the RHS of the Client's final transition and define an attack state named `orchestrationFinalState` as a one-fact state that contains `iknows(some_fresh_constant)` (see Example 3).

Example 3

```
        ....
        step step_5(Ag, Stp, Dummy_A):=
        state_OrchestrationClient(Ag, 5,Dummy_A).
        iknows(a)
    =>
        state_OrchestrationClient(Ag, 6, a).
        iknows(finish)

        ...
    attack_state orchestrationFinalState(Dummy) :=
    iknows(finish)
```

2.2.2 Orchestration problem with a partially specified Goal service

In the case where a Goal is specified as input, one must provide:

- the specification of available services;
- the partial specification of the Goal service, related to communication with the Client;
- the initial knowledge of the Goal service.

Notice that `goal orchestrationConstraint` is optional. The attack state `orchestrationFinalState` will be ignored.

2.3 Conclusion

The ASLan language provides useful primitives to specify orchestration problems. Still, a number of constraints are currently limiting the scope of the method to services that are mainly specified in terms of message communications along simple workflows. We shall relax some of these limitations in future versions of the AVANTSSAR Platform. In spite of these limitations we can already tackle interesting problems as shown in § 6.

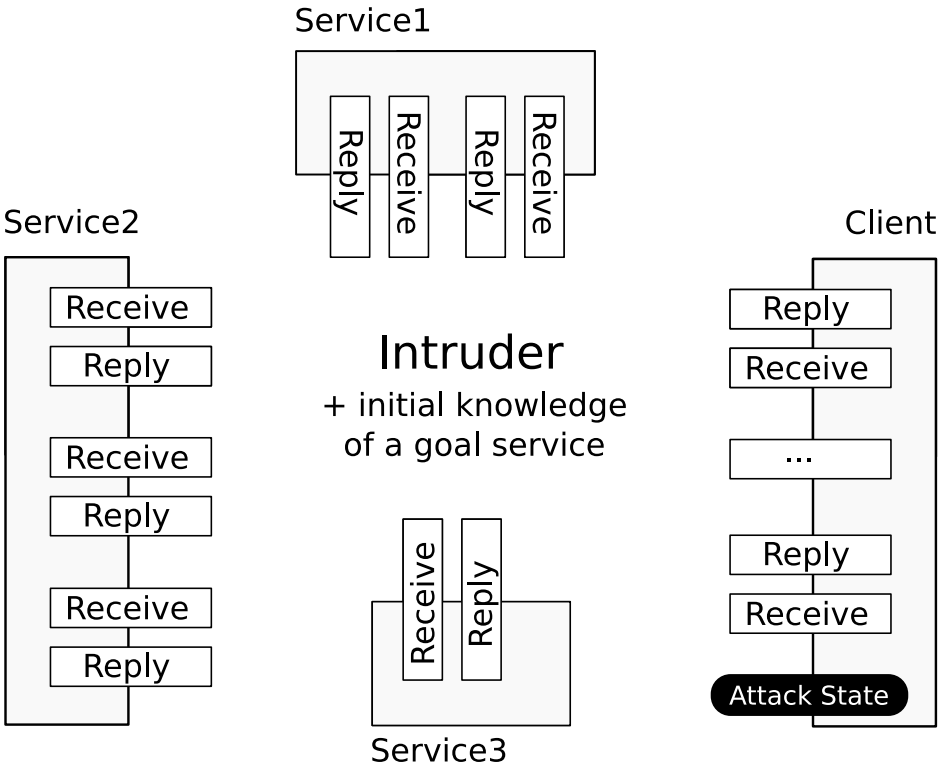

Figure 2: Services as protocol roles

3 The Orchestrator

3.1 Approach overview

The general idea is to represent the available services and the client service as protocol roles. The intruder with Dolev-Yao capabilities, who has a full control over the network, will play the role of an orchestrator: he tries to lead the given transition system from its initial state to a final accepting one. That is why final states are encoded as attack states (the point of view of the intruder). If he succeeds that means that he is able to satisfy all Client's requests having only initial knowledge of the Goal service and being able to invoke the available services (see Figure 2).

In order to check whether the attack state can be reached, we have employed a version of the back-end CL-AtSe: the result is a trace containing the sequence of messages sent and received by the intruder. From the trace we then extract an executable ASLan specification of the Goal service (see Figure 3). In fact, if the intruder can perform the necessary steps to build the messages that appear in the trace, we are guaranteed that the resulting

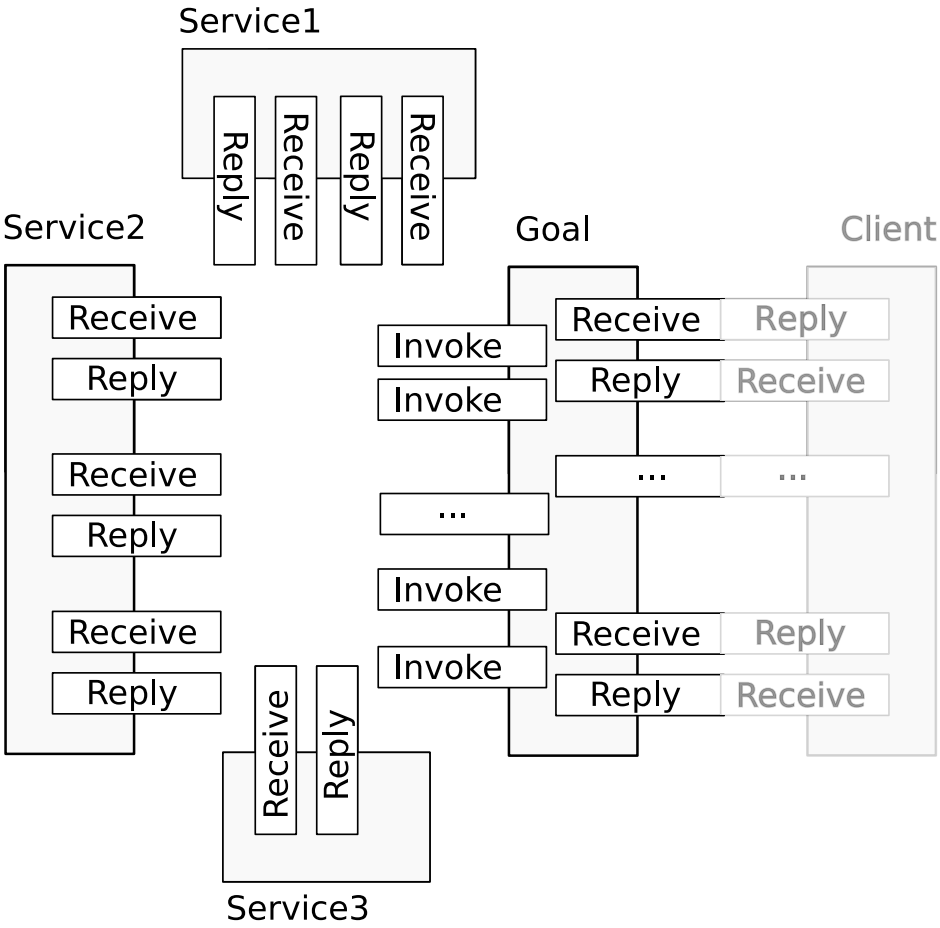

Figure 3: Orchestration problem solution

service is executable. (We assume that the Goal service has at least the same message construction capabilities of a Dolev Yao intruder.) The operations for building messages are not given in the resulting ASLan specification, but only invocations of necessary available services with already constructed request-messages are present.

If the partial specification of the goal is given, the tool first creates a corresponding putative Client service (see Figure 4) and then proceeds as described above.

3.2 Limitations of the current version

The current prototype has a number of limitations, which will be addressed in the next, final version of the AVANTSSAR Platform. For different input options, different restrictions are imposed.

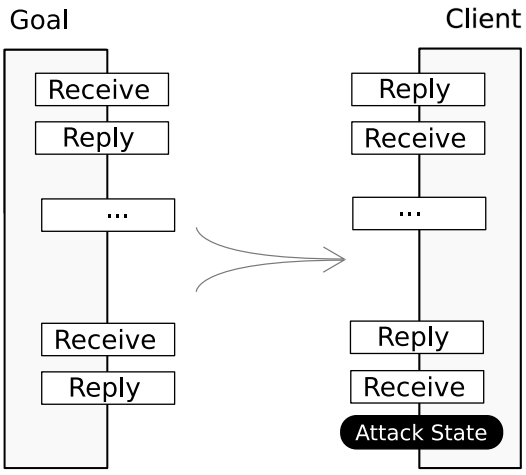

Figure 4: Goal and its putative Client

3.2.1 General limitations

- Horn clauses for specifying the access control policies are not supported yet since the orchestrator back-end CL-AtSe does not support them.
- LTL formulas are only partially supported for the same reasons. They should be added in the next version in order to model orchestrations with more elaborate workflow constraints.
- Automatic backtracking is not supported yet. This means that the tool proposes only one possible orchestration. If the validation platform proves the insecurity of the resulting service, then there is no means to produce another orchestrated service.

3.2.2 Limitations with Client input

- The Client is specified by its communications with the goal service.
- No loops are allowed in the workflow of the Client.

3.2.3 Limitations with Goal service input

- The Goal service is described as a transition system only in terms of exchanging messages (conversation) with the Client (in the style of Alice-and-Bob specifications). In the next version of the platform, all other orchestration constraints should be expressed in `section goals` as LTL formulas.

- The workflow of the Goal service must be linear (no loops, no forks).
- Only equality tests are supported as conditions in transitions containing `state_OrchestrationGoal`.
- All the variables used in the definition of a transition that contains `state_OrchestrationGoal` are supposed to have atomic values (no variables of type `message` or other composed type like `apply(fun, pair(text,text))`).
- The initial knowledge of the client is not taken into consideration (and thus, not supported).

The pattern of the Goal's transition is:

```
step step_name(<params>):=
      state_OrchestrationGoal(<terms1>).
      iknows(<term0>) &
      equals(term1, term2) &
      equals(term3, term4)
  =[exists <vars>]=>
      state_OrchestrationGoal(<terms2>).
      iknows(<term5>)
```

The transition has a a unique `state_OrchestrationGoal` in the LHS and RHS; only `iknows` as other facts and only equalities as conditions. Labeling with a list of existentially quantified variables is allowed.

3.3 Architecture overview

The Orchestrator takes as input an ASLan file with a specification of the available services and either a specification of the Client or a partial specification of the Goal. It produces as output an ASLan file with the specification of the available services, a full specification of the Goal, and a specification of the Client (a putative one, if it was not given as input).

Figure 5 shows the architecture of the Orchestrator. It is logically partitioned into the generator and the integrator. Input specifications are sent to `Client2Client`, the tool that prepares inputs for CL-AtSe. If `Client2Client` detects that the input is not in a format corresponding to a Client, then `Goal2Client` is invoked with the same input specification. The `Goal2Client` prepares the input for CL-AtSe but assumes that a Goal specification is given. It also generates some additional data to help `Trace2ASLan` to integrate the resulting service into an ASLan file. The output of CL-AtSe is given as input to `Trace2ASLan` together with the input specification, and possibly, with

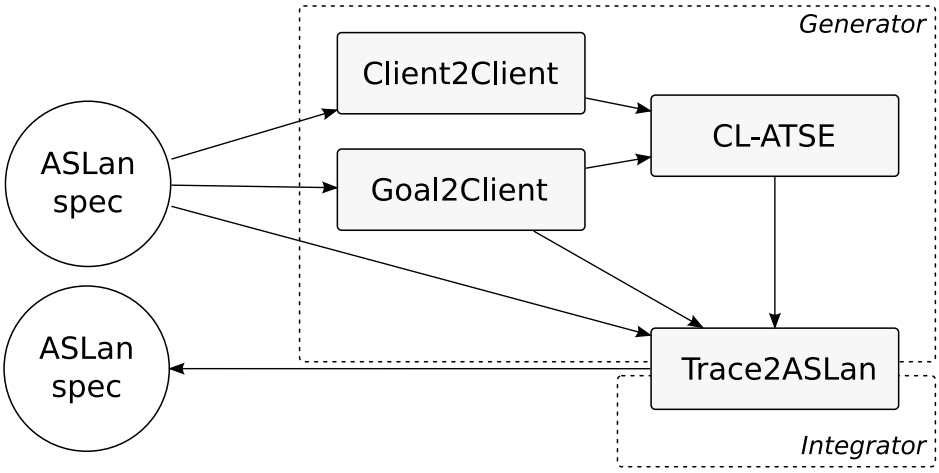

Figure 5: Tool architecture

additional data generated by `Goal2Client`. Then, `Trace2ASLan` integrates the resulting Goal service into the input specification (adding the security properties, specified by the modeler for the validation of the generated orchestration).

3.4 Components

3.4.1 `Client2Client`

The purpose of this simple component is to prepare input for the case where a Client specification is given to CL-AtSe. It checks the suitability of the input against the requirements given in § 2 and § 3.2 and initializes the intruder knowledge with the initial knowledge of the Goal.

3.4.2 `Goal2Client`

The purpose of this component is to prepare input when a partial Goal specification is given to CL-AtSe. It generates a Client specification from the partial Goal specification given as input. More in detail, it

1. checks the suitability of the input against the requirements given in § 2 and § 3.2;
2. generates a Client specification from the Goal specification and integrates it into the input specification, removing a Goal;
3. replaces the intruder knowledge by the initial knowledge of the Goal service; and

4. outputs the generated secret name, the name of the agent executing the newly created Client service, the well-ordered sequence of goal steps and the renaming table that was done during transformation of the Goal to the client to have a link with the input specification, into two auxiliary files, if it was requested with a command-line option.

We give an example of the transformation of a one-transition Goal service to the corresponding Client. Suppose that we have the following specification of the Goal service (an "$(I_1 + I_2)^2$" service described in § 3.5) as input:

Example 4

```
step step_2 (GOAL,D_I1,D_I2, D_Set_36,SID,I1,I2) :=
 state_OrchestrationGoal(GOAL,1,D_I1,
    D_I2,D_Set_36,SID).
 iknows(pair(I1,I2))
 =>
 state_OrchestrationGoal(GOAL,2,I1,I2, D_Set_36,SID).
 iknows(apply(times,pair(apply(plus,pair(I1,I2)),
    apply(plus,pair(I1,I2))))).
 secret(apply(times,pair(I1,I2)),sec_prop,D_Set_36).
 contains(GOAL,D_Set_36)
```

The `Goal2Client` returns an ASLan file, where the specification above is replaced with the corresponding Client service.

Example 5

```
  step step_0001(ClientAgent_0000):=
       state_OrchestrationClient(ClientAgent_0000, 1).
       iknows(start)
    =>
       state_OrchestrationClient(ClientAgent_0000, 2).
       iknows(pair(var_Renamed_I1_0000_0000,
          var_Renamed_I2_0000_0000))

  step step_0000(ClientAgent_0000):=
       state_OrchestrationClient(ClientAgent_0000, 2).
       iknows(apply(times, pair(apply(plus,
          pair(var_Renamed_I1_0000_0000,
          var_Renamed_I2_0000_0000)), apply(plus,
          pair(var_Renamed_I1_0000_0000,
          var_Renamed_I2_0000_0000)))))
    =>
       state_OrchestrationClient(ClientAgent_0000, 3).
       iknows(finish_0000)
```

In its first transition `step_0001`, the Client sends a message that would be accepted by the Goal from Example 4 (the first dummy sending is not counted), and in its second transition Client expects to receive a message that has to be sent by the Goal if the Goal received a message sent by the Client. Here, constants `var_Renamed_I1_0000_0000` and `var_Renamed_I2_0000_0000` represent variables `I1` and `I2`, respectively.

This transformation of variables into constants leads to a correct result when we try to solve an orchestration problem thanks to the restriction on the atomicity of variables used in the Goal specification.

Please note that predicates related to security properties defined for Example 4 are ignored when building a corresponding Client (for example the predicate `secret(apply(times,pair(I1,I2)),sec_prop,D_Set_36)`). They will be integrated to the resulting specification thanks to the integrator part of `Trace2ASLan` tool.

3.4.3 CL-AtSe

This is a key component of the generator part of the orchestrator. Given an ASLan specification, it generates a trace — the sequence of communication events (send/receive) between a mediator (played by the intruder) and the available services (that can be used in the composition) — such that a special attack state is reached and an LTL formula is satisfied (only simple formulas equivalent to non-reachability of attack states are allowed for now but extensions are in progress; these are enough for orchestration). A detailed description of the tool can be found in § 4.1 and [33].

3.4.4 `Trace2ASLan`

We now focus on the *mediator*, i.e. the service that exposes the orchestration functionality to the outside world.

The component `Trace2ASLan` takes as input:

- the attack trace returned by CL-AtSe, describing the mediator, and
- the initial ASLan input (and possibly some of the auxiliary files produced by `Goal2Client`),

and returns as output the initial ASLan specification augmented with the ASLan specification of the mediator. This component is also responsible for feeding the validator with the initially proposed validation problem augmented with the freshly generated mediator (so that the validation problem takes also into account the presence of the mediator).

Moreover, the derived ASLan specification for the mediator is augmented with tests for checking its input messages as thoroughly as possible. Let us consider the following toy example to better illustrate this property:

Example 6

```
s initially knows: {m}

c -> s: scrypt(k,m)
c -> s: k
s -> c: pair(m,n)
```

This trace describes the communication between a service s (object of the study) and a client c. The service s first receives the encryption of a certain message m it already knows with a symmetric key k it does not know. Then, the service s receives the key k, generates a new nonce n and sends it paired with m.

Given the provided initial knowledge, an implementation of the service s that checks its input as thoroughly as possible should test, at the second reception step, if the first received message is exactly equal to the encryption of the already known message m with the freshly received key k. This check can be extracted from the attack trace (by identifying similar message parts in the attack trace).

The component also provides in a separate file an operational specification of the mediator, which is defined in the following terms:

- For each reception activity, it provides the set of necessary tests to be checked against the message being received before accepting it and updating current knowledge.
- For each sending activity, it provides the set of operations to be performed on elements in the current knowledge in order to construct the message to be sent.

We have chosen to store the result of these computations separately in view of its future use when generating WS-standards compliant implementations (in BPEL/WSDL style) for the generated mediators or for their security wrappers (see § 3.4.5 for more details).

In the remainder of this section, we explain how the `Trace2ASLan` component performs the previous tasks. First, the tool starts from a given initial knowledge (a set of ground terms extracted from the ASLan input) and proceeds by treating each communication activity appearing in the CL-AtSe trace as follows:

Reception case: Compute the reachable submessages of the message being received by the mediator, using the keys that are in its current knowledge. Denoting the Dolev-Yao operation o and the known messages l, r such that $m = o(l, r)$, for or each of these submessages m we do:

- If m is not already in the current knowledge, add it to the current knowledge and attach the operation that permits one to reach it;
- otherwise, m is already known by the mediator. Hence there exists a list of relations $o_i(l_i, r_i) = m$ for $i \in I$ where o_i is a Dolev-Yao operation and l_i, r_i are in the current mediator knowledge. The tool produces an equality test $o(l, r) = o_1(l_1, r_1)$ to be checked at this step by the mediator.

Sending case: Specify how the message to be sent can be built from the current knowledge. This is possible here since the CL-AtSe attack trace witnesses the fact that the intruder was able to build all sent messages from his knowledge.

In a second phase, the tool starts generating the ASLan specification describing the orchestration. The generated ASLan specification corresponds to a prudent implementation of the orchestration in the sense of [22]. The tool proceeds as follows:

Building the transition system: The local state of the mediator is basically the set of submessages it can reach at some step. This includes its initial knowledge, the received messages and the nonces available to it at this step. If the message to be sent requires some nonce generation, the rule is also prefixed with the corresponding existential quantifications.

Signature and types section: Most variables occurring in the generated ASLan specification can be related to equivalent ones in the ASLan input. This mapping can be established by analyzing the names appearing in the CL-AtSe trace. Thus the types for these variables can be easily inferred. The remaining variables receive fresh values (e.g. nonces) generated by the mediator, and if needed their types can be inferred from their position in the sent term (for instance, if a nonce appears for the first time in a sent message in the position of a symmetric key, it will be assigned the `symmetric_key` type). After typing all the constants and variables appearing in the transition system, the signature of the `state_OrchestrationGoal` fact (the state fact describing the mediator, according to the conventions) can then be defined consistently.

The generated ASLan rules for Example 6 are as follows:

```
step step_1(SID,M,Dummy_X,X,Dummy_K,Dummy_N) :=
 state_OrchestrationGoal(SID,1,M,Dummy_X,Dummy_K,
        Dummy_N).
 iknows(X)
  =>
 state_OrchestrationGoal(SID,2,M,X,Dummy_K,Dummy_N)

step step_2(SID,M,X,Dummy_K,K,Dummy_N,N) :=
 state_OrchestrationGoal(SID,2,M,X,Dummy_K,Dummy_N).
 iknows(K)
 &equal(X,scrypt(K,M))
  =[exists N]=>
 state_OrchestrationGoal(SID,4,M,X,K,N).
 iknows(pair(M,N))
```

At this point the tool outputs an independent ASLan specification, describing the mediator. This output is merged with the initial ASLan input before the entire specification of the orchestration is submitted to the Validator.

Note that the tool delivers an intermediary result after the first phase that can be used to generate an implementation of the mediator, if one is given an implementation of the basic Dolev-Yao operations. This implementation also checks the input messages as thoroughly as possible, as explained in the following subsection (for more details, see [22]).

3.4.5 Preliminary steps for a Wrapper implementation

The main purpose of the *Trust and Security Wrapper* is to enforce the trust and security policy attached to the mediator, during the execution of the orchestration.

Figure 6 shows a toy mediator service, its Wrapper and a client service. The policy of the mediator says it should only receive messages that are encrypted with the public key k and digitally sign the messages it sends using the private key $inv(k)$. The Wrapper acts according to the policy, applying the required cryptographic operators to inbound and outbound messages, while the mediator focuses on different tasks (here, pairing of messages).

If we assume that all the available services and their Wrappers are already implemented using WS-standards, then the platform can also provide an implementation in terms of WS-standards of the mediator and of its Wrapper. For WS-standards compliance, the mediator implementation should comprise at least the following files: its WSDL specification, its WS-SecurityPolicy

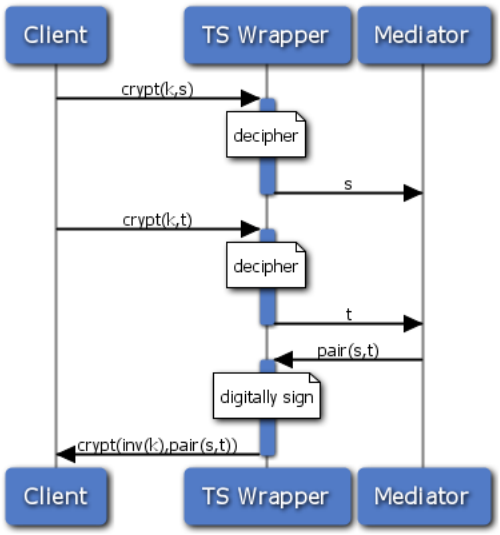

Figure 6: Trust and Security Wrapper

specification and its BPEL specification.

The intermediate results (see § 3.4.4) of `Trace2ASLan` should allow us in principle to derive automatically:

- A BPEL executable specification of the mediator service that can be executed under a BPEL Engine, like Ode [3]. Note that this implementation also takes into account the previously generated set of tests to be checked against input messages for each step.
- A WSDL specification describing the interface of the mediator in terms of operations where inbound and outbound messages abide to given XML schemas.
- A WS-SecurityPolicy file, given which a BPEL engine such as Ode can use its associated security module, Rampart [2], to apply the trust and security wrapper functions:
  - checking whether the received messages abide by the given policies (i.e. some parts are encrypted, digitally signed and/or time-stamped as specified in the policy), and extracting the parameters needed to apply the mediator operations;
  - performing symmetric operations on sent messages: intercepting the message sent by the mediator and possibly applying some cryptographic operations to the message (as specified in the policy) before forwarding it to the original recipient.

3.5 A toy example

Suppose that we want to create a service that accepts two numbers and returns the square of their sum. The target service is not supposed to carry out the multiplication and addition operation, but rather to rely on available services for this: the multiplier (which multiplies two given values) and the adder (which computes the additions of two given numbers).

First, we must specify the available services.

Available services:

Adder: receives two values a, b, and sends $a + b$.

```
step step_0 (A,SID,I,J) :=
 state_Adder(A,1,dummy_msg,dummy_msg,SID).
 iknows(pair(I,J))
 =>
 state_Adder(A,2,I,J,SID).
 iknows(apply(plus,pair(I,J)))
```

Its state-fact in the initial state is:

```
  state_Adder(a,1,dummy_msg,dummy_msg,3)
```

Multiplier: receives two values a, b, and sends $a * b$.

```
step step_1 (M,SID,I,J) :=
 state_Multiplier(M,1,dummy_msg,dummy_msg,SID).
 iknows(pair(I,J))
 =>
 state_Multiplier(M,2,I,J,SID).
 iknows(apply(times,pair(I,J)))
```

Its state-fact in the initial state is:

```
  state_Multiplier(m,1,dummy_msg,dummy_msg,4)
```

Then, as there are two options for specifying the input, we have to choose between the partial Goal description and the Client description.

Target services:

Client: has (sends) a, b; wants (expects) $(a + b) * (a + b)$.

Goal: receives a, b; returns $(a + b) * (a + b)$.

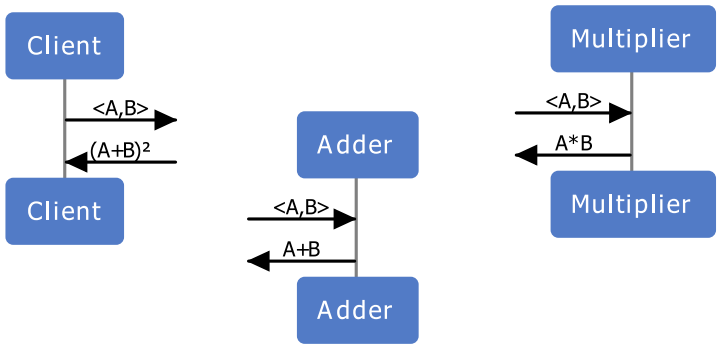

Figure 7: Toy example: available services and the Client

3.5.1 Client case

If the modeler chose to specify the behavior of the Client to define an orchestration problem, then the Client specification is:

```
step step_2 (C,SID,I1,I2) :=
 state_OrchestrationClient(C,1,dummy_nonce,
    dummy_nonce,SID).
 iknows(start_orchestration)
=[exists I2,I1]=>
 state_OrchestrationClient(C,2,I1,I2, SID).
 iknows(pair(I1,I2)).
 keepsecret(apply(times,pair(I1,I2)), secret_value_id)

step step_3 (C,I1,I2,SID) :=
 state_OrchestrationClient(C,2,I1,I2, SID).
 iknows(apply(times,pair(apply(plus,pair(I1,I2)),
    apply(plus,pair(I1,I2)))))
 =>
 state_OrchestrationClient(C,3,I1,I2, SID).
 iknows(end_orchestration)
```

Its initial state is:

```
 state_OrchestrationClient(c,1,dummy_nonce,
    dummy_nonce,5).
```

At the moment of writing this deliverable, the tool prototype required to use dummy send and receive if no payload message is to be sent (`iknows` should be always present on both sides of a service's transition). Here, `iknows(start_orchestration)` is the case.

The last `iknows(end_orchestration)` is a mark of the final Client transition (see § 3.2.2 and Example 3); the corresponding attack state is defined

as:

```
attack_state orchestrationFinalState (ASGoal) :=
 iknows(end_orchestration)
```

The fact `keepsecret(apply(times,pair(I1,I2)), secret_value_id)` is given as an example of security property to be verified. It is not related to an orchestration problem, but to validation. The corresponding attack state is defined as follows:

```
attack_state to_validate(MGoal) :=
 iknows(MGoal).
 keepsecret(MGoal,secret_value_id)
```

Conforming to requirements listed in § 2, we should also define an initial knowledge of the Goal service:

```
 state_OrchestrationGoal(g,start_orchestration)
```

3.5.2 Goal case

The other option is to partially define the Goal service, i.e. specify a part related to the communications with the Client:

```
 step step_2 (GOAL,SID,I1,I2) :=
   state_OrchestrationGoal(GOAL,1,
      dummy_msg,dummy_msg,SID).
   iknows(pair(I1,I2))
 =>
   state_OrchestrationGoal(GOAL,2,I1,I2, SID).
   iknows(apply(times,pair(apply(plus,pair(I1,I2)),
      apply(plus,pair(I1,I2))))).
   keepsecret(apply(times,pair(I1,I2)),secret_value_id)
```

Here, we also added a secrecy property to be verified by the Validator. The corresponding attack state is:

```
 section goals:

 attack_state to_validate (MGoal,ASGoal) :=
 iknows(MGoal).
 keepsecret(MGoal,secret_value_id).
```

The initial state includes

```
  state_OrchestrationGoal(g,1,dummy_msg,dummy_msg,5)
```

to initialize the Goal service.

4 The Validator

The Validator takes as input an orchestration and a security goal formally specified in ASLan v.1, and automatically checks whether the orchestration meets the security goal. If this is the case, then the ASLan v.1 specification of the validated orchestration is given as output, otherwise a counterexample is sent back to the Orchestrator.

Currently, the functionality of the Validator is supported by the following back-ends: CL-AtSe (developed and maintained by INRIA), OFMC (developed and maintained by IBM), and SATMC (developed and maintained by UGDIST).

4.1 CL-AtSe

CL-AtSe [33, 7] is a Constraint Logic based Attack Searcher for security protocols and services. The main idea in CL-AtSe consists in running the protocol or set of services in all possible ways by representing families of traces with positive or negative constraints on the intruder knowledge, on variable values, on sets, etc. Thus, each run of a service step consists in adding new constraints on the current intruder and environment state, reducing these constraints down to a normalized form for which satisfiability is easily decidable, and decide whether some security property has been violated up to this point. CL-AtSe does not limit the service in any way except for bounding the maximal number of times a service can be iterated, in the case such an iteration is allowed in the specification. Otherwise, the analysis might be non-terminating on secure services and only heuristics, approximations, or restrictions on the input language could lift this limitation.

If a security property of the input specification is violated then CL-AtSe outputs a warning (UNSAFE), some details about the analysis (e.g. whether the considered model is a typed or an untyped one), the property that was violated (secrecy, for instance), statistics on the number of explored states, and, finally, an ATTACK TRACE that gives a detailed account of the attack scenario. If no attack was found then similar information is provided (but the ATTACK TRACE).

Simplifications and optimizations: CL-AtSe implements modules to simplify and optimize the input specification statically before analysis. The simplification module aims at reducing the search space without truly changing the structure of the problem to be analyzed. This option is activated by default because the computations involved are quite small compared to the expected gain on the analysis time. The optimization module, on the other

hand, tries to change the structure of the input specification to pre-process a significant part of the branching that should be done during analysis, so that many dead-ends can be found and removed statically. This option is not activated by default since, theoretically, in the worst case it might be equivalent to a complete analysis, especially for specifications with weak message structure (e.g. a large number of encryptions with no headers, etc.). However, when the specification is adapted the gain during analysis can be huge. It can be activated with the "-opt" option.

Support for ASLan: CL-AtSe reads any specification written in ASLan v.1 by default. The only current limitations are: no Horn clauses and limited support for LTL. We are continuously working on the tool, and we expect to lift these two restrictions in a near future. Currently, security goals are rewritten into attack states, for which support in the tool is complete. Significant LTL support for properties based on future time will be added soon. However, full support for Horn clauses will require approximation methods that are currently under development and implementation.

4.2 OFMC

The Open-source Fixedpoint Model Checker OFMC [15,31,30,20] consists of two modules: the classical module performs verification for a bounded number of transitions of honest agents (similar to SATMC) using a constraint-based representation of the intruder behavior. The novel fixedpoint module works on an over-approximation of the search space that is specified by a set of Horn-clauses using abstract interpretation techniques and counter-example based refinement of abstractions. While the over-approximation can introduce "false positives" (attacks that do not work in the reference model), it allows for verification without restricting the number of steps.

Figure 8 shows the architecture of the OFMC tool-suite in detail. It supports specifications written in one of two input languages: ASLan v.1 as the standard input language of AVANTSSAR, and a a novel language AnB based on Alice-and-Bob notation with special support for algebraic properties of operators, pseudonymous secure channel notation, and zero-knowledge proofs [28, 29, 9]. We will present the AnB language in more detail in an future deliverable as part of the industrially suited specification languages we have been developing.

The standard procedure is to check an ASLan specification in parallel in both the classical and the fixedpoint modules to exploit the complementary features of these modules. The classical module is limited to a bounded number of steps of honest agents. When an attack (a violation of the goals)

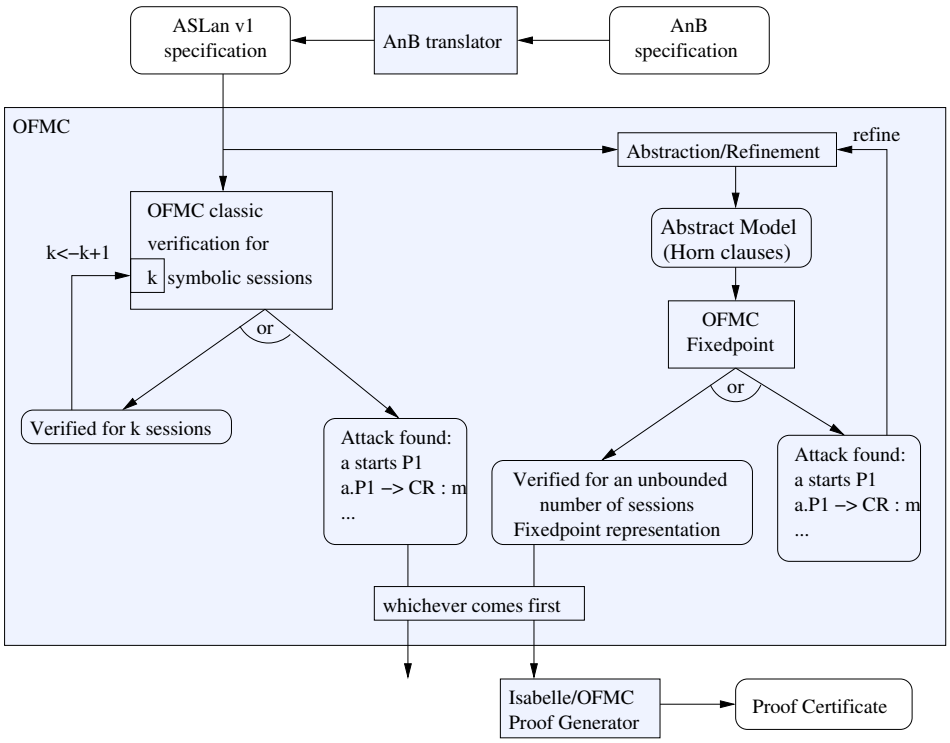

Figure 8: The architecture of the OFMC tool suite.

is found, it is immediately reported to the user. When no attack is found, however, the bound on the steps is increased and the analysis is repeated. Thus, for a correct service (or protocol), the classical module will never terminate but repeatedly report to the user the bound under which the system has been verified.

The fixedpoint module of OFMC [20, 27] over-approximates the search space in two regards, an abstraction of control and data, following the line of static analysis and abstract interpretation approaches such as [17, 18, 19, 26]. OFMC starts with a very coarse abstraction and refines this abstraction automatically whenever the verification with the fixedpoint module fails, i.e. when the fixedpoint contains a potential attack. Note that attacks found can be "false positives" caused by the over-approximation of the model, and it is thus reasonable to try the verification with a refined model.

Running both modules in parallel, OFMC stops as soon as the classic module has found an attack or the fixedpoint module has verified the specification, so as soon as there is a definitive result. Otherwise, OFMC can just report the bounded verification results and the potential attacks that the fixedpoint module has found.

In the case of a positive result, we can use the computed fixedpoint to automatically generate a proof certificate for the Isabelle interactive theorem prover. The automatic proof generator OFMC/Isabelle was developed in collaboration by IBM and SAP [20]. The idea is to gain a high reliability, since after this step the correctness of the verification result does no longer depend on the correctness of OFMC and the correct use of abstractions. Rather, it only relies on two things:

- the correctness of the small Isabelle core that checks the proof generated by OFMC/Isabelle, and
- that the original ASLan specification (without over-approximations) indeed faithfully models the system and properties that are to be verified.

Support for ASLan: The classic module supports the entire ASLan standard with three restrictions. First, for LTL we currently allow only formulae of the form $G\ \phi$ for a propositional formula ϕ, so that a violation is defined by a state satisfying $\neg\phi$. We plan to extend OFMC to support arbitrary safety properties. Second, we support only quantifier-free Horn-clauses (i.e. the variables of the head are a subset of the variables of the body). Third, some features of the typing system (sub-typing, composed types) are not yet supported by OFMC and handled with an over-approximation (which can lead to the detection of ill-typed attacks).

The fixedpoint module is rather new and currently supports only ASLan specifications that contain no negative conditions in transition rules and can only check for secrecy and weak authentication properties. Part of the ongoing work of WP3 and WP4 is to extend the scope of this method.

4.3 SATMC

SATMC is a state-of-the-art model checker that has been key to the discovery of serious flaws in security protocols and services [4, 5]. At the core of SATMC lies a procedure that, given a security problem, automatically generates a propositional formula whose satisfying assignments (if any) correspond to counterexamples on the security problem of length bounded by some integer k. Intuitively the formula represents all the possible evolutions of the transition system described by the security problem up to depth k. Finding attacks (of length k) on the service therefore boils down to solving propositional satisfiability problems. SATMC relies on state-of-the-art SAT solvers for this task, which can handle propositional satisfiability problems

with hundreds of thousands of variables and clauses or more. SATMC can be instructed to perform an iterative deepening on the number of steps, i.e. on k. As soon as a satisfiable formula is found, the corresponding model is translated back into a partial-order plan. As in AI planning, a *partial-order plan* is a partially ordered (multi)-set of rules that leads the system from the initial state to the goal state. (A partial-order plan concisely represents a set of attacks on the service.) More details on SATMC can be found in [4, 6].

Support for ASLan: SATMC provides almost complete support to the ASLan v.1 language. We have extended SATMC to support model checking of LTL formulae as described in [16]. Currently all LTL future operators are supported, while LTL past operators are not supported yet. We have also extended the SAT encoding technique for supporting the Horn clauses.

Table 1: Services currently available

Host	URL	Service
UNIVR	http://regis.sci.univr.it/avantssar	Validator
INRIA	http://avantssar.loria.fr	Validator, Orchestrator
UGDIST	http://ai-lab.it/avantssar	Validator, Platform

5 The AVANTSSAR Platform as a Service

The AVANTSSAR Platform is implemented as a service-oriented architecture (SOA), where each component service is offered as a Web Service. The services currently available are reported in Table 1. The table shows the name of the partner hosting the service, and the URL where the service (and its WSDL interface) can be accessed. As shown in the table, we provide three different instances of the Validator service (one for each host). For each instance, the functionality of the Validator is supported by a different back-end, among the ones described in § 4.

Invoking the services. In order to test the services, we have devised clients for invoking each service. The requests can be sent online in a web-page format on http://ai-lab.it/avantssar, which then builds the SOAP request to invoke each service. Alternatively, a simple python function can be defined to invoke the services. The following code can be used for invoking the orchestrator service, and can be easily adapted for the other services:

```
from SOAPpy import SOAPProxy

def orchestrate(spec):
    url = 'http://avantssar.loria.fr' +
       '/AVPLoriaOrc/services/AVPLoriaOrcSOAP'
    namespace = 'http://www.avantssar.eu/AVPLoriaOrc/'
    server = SOAPProxy(url)
    return server._ns(namespace).Orchestrate(orcInput =
       spec)['outASLAN']
```

The function `orchestrate` takes as input a string containing the ASLan specification of the orchestration problem and it returns a string containing an orchestration, if one is possible, and an error message otherwise.

5.1 The Platform

The platform service is hosted by UGDIST. It is deployed as a web service implemented in PHP5, by using the WSO2 Web Services Framework for PHP (WSO2 WSF/PHP) [34], an open source, enterprise grade, PHP extension for providing and consuming Web Services in PHP. The framework provides base communication functionality in SOAP, XML, and other message formats carried over various transports including HTTP, SMTP, XMPP and TCP. SOAP and HTTP are the standards used for the current Web Services implementation.

5.1.1 Interface

The current version of the platform does not include the connectors layer. The services that carry out the translation from application-level specification languages into the ASLan v.1 language (and vice versa) will be integrated in the last version of the platform. Thus, the current input/output of the platform service is an ASLan specification.

Input: A string in the ASLan v.1 format of an orchestration problem.

Output: A string that either describes the solution of the orchestration problem or contains a keyword indicating the impossibility to find a valid solution (and possibly some more information).

Given an ASLan specification of an orchestration problem as input, the workflow representing the platform service behavior comprises the following set of tasks:

(a) invocation of the Orchestrator service in order to obtain an ASLan specification of the orchestration satisfying the functional goals;

(b) parallel invocation of all the available Validator services, in order to check the security of the services orchestration;

(c) return as output of the first valid response obtained by one of the Validator services.

As mentioned in § 3.2, the automatic backtracking of the Orchestrator tool is not supported yet. If the validation phase proves the insecurity of the resulting service, then there is no means to produce another orchestrated service. In the next release of the platform we will integrate the backtracking phase: steps (a) and (b) will be repeated until a secure services orchestration is found, if it exists, or a warning will be returned otherwise.

Note that for some case studies the orchestration phase is not required. If this is the case, the platform service neglects step (a) and invokes directly the validators as in step (b).

5.2 The Orchestrator

The orchestrator service is hosted by INRIA. The tool is deployed as a Web Service implemented in Java. The Eclipse integrated development environment has been used to generate the WSDL file describing the interface of the orchestrator service and then the provided Axis2 [1] plugin has been used to generate skeleton java classes for the service. The skeleton class files have been eventually updated with the code corresponding the treatment performed by the service before deploying it under an Apache Tomcat Server.

5.2.1 Interface

The interface of the orchestrator service is as follows:

Input: A string containing the ASLan v.1 specification of an orchestration problem.

Output:

- A string containing the type of returned value (erroneous or regular execution).
- A string containing the error message (if any) or the resulting ASLan specification including the composed Goal service and the Client.

Note that, when the automatic backtracking of the Orchestrator tool will be supported, we will consider another string as input, containing either a counterexample if the solution of a previous orchestration problem does not satisfy the security goals, or a void message otherwise.

5.3 The Validator

We provide three different instances of the validator service, each of them leveraging a different back-end, presented in § 4. In particular, OFMC, CL-AtSe, and SATMC are hosted by UNIVR, INRIA, and UGDIST, respectively.

5.3.1 Interface

The common interface for the validator services is as follows:

Input: A string containing the solution of the orchestration problem in ASLan v.1.

Output: A string containing either a counterexample if the solution of the orchestration problem does not satisfy the security goals, or a void message otherwise.

6 Experimental Results

We have tested the current state of the AVANTSSAR Platform using four of the case studies proposed in Deliverable D5.1 ("Problem Cases and their Trust and Security Requirements" [10]):

- Digital Contract Signing,
- Public Bidding,
- Car Registration,
- Loan Origination Process.

In addition to the validation phase, which is part of all case studies, DCS, PB and CRP also include an orchestration phase. Furthermore, CRP and LOP involve access control policies, specified as Horn clauses in addition to the transition rules of the individual services.

Modeling, orchestration and validation of the case studies have been very instrumental in assessing the usability of the platform and refining the goals for the final year of the project. More specifically:

Translators Three of the case studies have been initially modeled in a high-level language before being translated into ASLan v.1. The workflow of Digital Contract Signing has been modeled in BPEL, whereas Public Bidding and Car Registration have been modeled in ASLan v.2. This has helped us to validate the output of the translator, and make explicit some of the assumptions of the back-end model checking tools. It has also validated the design decisions underlying ASLan v.1 and enabled minor additions increasing the ease of modeling. The case studies have also confirmed that the high-level description style of ASLan v.2 leads to a high number of transition rules in the ASLan v.1 translation. This is an issue that we had anticipated, resulting in the semantics for step compression defined in Deliverable D2.1 ("Requirements for modelling and ASLan v.1" [8]) and implemented in the back-end tools.

Orchestration The orchestrator has proved stable, requiring only a few modifications with respect to its initial deployment. Testing the orchestrator on widely different case studies has allowed us to quickly remove some initial limitations related to assumptions on the structure of the composed services. Both client-oriented and goal-oriented versions of the orchestration problem are supported, with relatively minor differences in the specification

style. The main desired feature remains the addition of Horn clauses, which would extend the support for policies from the validation phase also to the orchestration phase. Moreover, a desired enhancement arising from the use of the orchestrator is the definition of a potential common trace output format. Currently, the orchestrator relies on the trace facility of one specific back-end, CL-ATSE. The agreement on a common trace format would make the orchestrator back-end independent, allowing for the employment of the back-end that would arrive at the fastest and/or most efficient solution, depending on the characteristics of the input problem.

Validation The case studies have prompted extensive testing and enhancement of back-ends. The use of ASLan v.1 has led to re-standardizing on a common input language, since some of the AVISPA tools had diverged in their use of slightly different variants and modeling styles for the IF language. The case studies have also exposed differing implicit assumptions of individual tools, and prompted a unification of supported features. In particular, the issue of standard and customizable ASLan v.1 preludes and/or a clear specification of common built-in tool facilities have been identified as an issue for the next platform release. As the service case studies have a complexity sometimes significantly higher than the typical protocols of the AVISPA toolset, and a different structure (in particular, the large number of roles) some of the back-end features have been adapted and reengineered for better efficiency for the analysis of services.

6.1 Digital Contract Signing

First, we briefly recall the case study; a full description can be found in Deliverable D5.1 [10] and will appear in Deliverables D5.2 and D5.3 together with its formalisation and validation analysis.

The Digital Contract Signing case study represents a contract signing procedure carried out by two partners. Both have a secured access to the web service called Business Portal (BP), which has a contract to be signed. After the first signer signs the contract, the BP needs to verify its signature, check that the corresponding certificate was not revoked, and timestamp the signed document. Then, if the checks are successful, the second signer signs the contract. The BP carries out the same checks as for the first signature, and, if everything is alright, the signed contract is archived for long-term conservation and the signers are notified about the successful storing.

The processing of proof elements (like signature, timestamp) is delegated to another service, called Security Server (SS), which is to be built. During

the modeling of the case study, we abstracted from the signers and represented both by one entity of BP. We chose SS to be our target (Goal), so the BP will play a Client role.

6.1.1 Available services

We defined the following services, which can be used by the orchestrator to generate an orchestration:

- TimeStamper: puts a timestamp.
- PKI: verifies whether the certificate has been revoked.
- Archiver: stores the given information.

As all internal abilities of the service to be built must be represented by available services, the additional service AssertionProvider was provided:

- it verifies the signature and generates the corresponding assertion representing the correctness of the signature;
- it generates a non-revocation assertion if the certificate has not been revoked; and
- it generates an assertion representing the fact that the message passed through TimeStamper, if an input message was timestamped.

For every available service, we defined as many different initial states as intended to be used. For example, to specify the fact that TimeStamper cannot be used more than twice, in the initial state we have:

```
...
      state_timestamper(ts, kts, md5, 1, dummy_nonce,
         dummy_msg, 3).
      state_timestamper(ts, kts, md5, 1, dummy_nonce,
         dummy_msg, 7).
...
```

where the last integer parameter denotes the session identifier of the service. The signature of this state fact is

```
state_timestamper: agent * public_key * hash_func *
   nat * text * message * nat -> fact
```

6.1.2 Client case

One of the possible inputs that we tried is to define the Client service (in this case it is a behavior of the BP). To make the orchestrated service perform the signature check, timestamp etc., we defined the Client in such way that it expects assertions that can be produced only if the signature is correct, the signed contract is timestamped, etc. In this way, we encoded some information that is required for a resulting service into the Client. For example (where the names of some constants were changed to shorter ones to increase readability):

```
step step_11(SSCl, PKI, TS, SS, BP, ARC, KPKI, KTS,
   KSSARC, KBPSS, Hash, SID):=
        state_OrchestrationClient(SSCl, PKI, TS, SS,
           BP, ARC, KPKI, KTS, KSSARC, KBPSS, Hash, 5,
           SID).
        iknows(scrypt(KBPSS, pair(cX1, pair(cX3,
           pair(cX4, apply(tsOK, pair(cX2, pair(cX3,
           pair(cX4, pair(signaturePolicy1,
           crypt(inv(cX4), pair(cX2, pair(cX3,
           pair(cX4, signaturePolicy1))))))))))))
    =>
        state_OrchestrationClient(SSCl, PKI, TS, SS,
           BP, ARC, KPKI, KTS, KSSARC, KBPSS, Hash, 6,
           SID).
        iknows(scrypt(KBPSS, pair(cX1, pair(cX3, cX4))))
```

Here, an assertion is encoded into the message. The part of input message that represents the timestamp assertion is:

```
apply(tsOK, pair(cX2, pair(cX3, pair(cX4,
   pair(signaturePolicy1, crypt(inv(cX4), pair(cX2,
   pair(cX3, pair(cX4, signaturePolicy1))))))))
```

Output In the output specification, we obtained the Goal service, consisting of 23 transitions.

6.1.3 Goal case

We also tried to specify the same orchestration problem by partially defining the Goal service. The difference in modeling between Client and Goal cases is not considerable. The requirements for internal checks are still modeled by assertions being a part of communication messages.

Output In the output specification, we obtained the new Goal service, consisting of 21 transitions. The Goal case needs two transitions less, which can be explained as follows: the first transition of the resulting Goal in the Client case corresponds to the dummy `start` to be sent, and the last transition of the resulting Goal in the Client case corresponds to the dummy value (`finish`) sent by the Client to determine the last transition.

6.1.4 Security properties to validate

When we attempted to define validation goals proposed in the case study, we faced a number of problems. For instance, one of the properties to be validated is an authentication between BP and SS (Client and Goal in our case), but only one of them can appear in the input of the orchestration problem, so that it became impossible to define this property before obtaining the orchestration.

Another property is the secrecy of a message sent by the mediator to the Archiver. The problem in this case is that the message to be secret will be only constructed after the orchestration problem is solved (as the message sent to the Archiver is not in the input, it is one to be found by the orchestrator). Instead, we defined another, stronger property denoted by P_1: any message received by the Archiver should stay secret. Note, that the messages received by the Archiver are encrypted with a shared key between the Archiver and SS.

The problem described above stems from the fact that some security properties can be specified in the input problem, but some others are dependent upon the specific messages that are exchanged by the mediator. For the latter case, we believe the alternative is either to insert between the orchestration and validation phases a step in which a human user interacts with the platform to add security properties concerning the mediator, or to change the input format to allow both Goal and Client to appear (maybe partially) in order to be able to define the security properties to be validated. We conjecture that the tools can be extended to deal with this as in [21].

For the meantime, we have decided to test the tools by adding security properties that were not specified by OpenTrust (the proposer of the case study) in order to check first whether the tool-chain is correctly handling the input files, and second whether the orchestration is amenable to automated analysis by the tools.

As the defined security property P_1 seems to be resistant to the presence of an intruder, we have tested another input specification with an additional secrecy property—the contract should stay secret—which cannot be valid even with a passive intruder. An attack has been reported immediately.

6.1.5 Conclusions

The DCS case study is a service orchestration problem that has been proposed by OpenTrust. The platform has been able to solve the problem and generate a Security Server. By changing the available services we can now easily explore alternative implementations of the Security Server manually designed by OpenTrust.

The input specification is quite large but the orchestrator prototype coped with the task for both input styles and returned the result in a very reasonable time.

To demonstrate the tool-chain as a whole (orchestrator and validator) on one example, we specified a security property that can be violated: any message received by the Archiver should stay secret.This showed that the tool can take as input an orchestration problem with a security property to validate, solves the orchestration problem, and then rejects the result in the validation phase. In the next tool version, we will allow some automatic backtracking to attempt new orchestrations in this situation.

6.2 Public Bidding

In this section, we give a formal specification of the *Public Bidding* case study in ASLan v.2. The case study has been described in full detail in Deliverable D5.1 [10]. Here we aim at validation as well as orchestration for the public bidding model.

In the specification, we model the *Bidding Portal* service, along with three types of *users* of the service, namely *Bidder*, *Bid Manager* and *Technical Committee*. With respect to the original specification of [10], we opt for a more concise model where the *Security Server* and its subcomponents are abstracted away. The Security Server tasks are carried out by the Bidding Portal.

We recapitulate the workflow of the case study, organized in four sequential phases:

1. **Publication Phase:** Before the call for tender is opened to the public, the Bid Manager uploads the tender specifications (i.e. an uninterpreted document) and the name of the Technical Committee into the Bidding Portal.

2. **Submission Phase:** Bidders retrieve the application form, containing the tender specifications, from the Bidding Portal. Then, while the submission phase is still open, they submit their applications, consisting of

a financial part and a technical part. The submission timer, determining the end of the submission phase, is modeled as a non-deterministic process executed in parallel with the main Bidding Portal process.

```
entity SubmissionTimer{
  body{
    submission_phase_open;
    retract submission_phase_open;
  }
}
```

3. **Evaluation Phase:** After completion of the submission phase, the Bidding Portal sends the set L_1 of the applicants to the Bid Manager. The Bid Manager at this stage consults its policy file and decides which of the applicants are *eligible* to participate. We model this decision step using static policies for the Bid Manager:

```
is_eligible (agent) : fact;
...
is_eligible(alice);
```

The eligible applicants are thus part of the Bid Manager specification.

Then the Bid Manager sends the set of the eligible applicants L_2 to the Bidding Portal. Obviously L_2 is a found by removing zero or more elements from L_1. The Bidding Portal at this stage sends the technical proposals of the eligible applicants to the Technical Committee.

```
  while (Eligible_applicants->contains(?Bidder)) {
    if (Applications->contains(Bidder,?,?,?TP))
      Eligible_technical_proposals->add(Bidder,TP);
    fi
    Eligible_applicants->remove(Bidder);
  }
```

Here, *Eligible_applicants* is the set of applicants that are deemed eligible by the Bid Manager (i.e. L_2 above). The Bidding Portal constructs the set *Eligible_technical_proposals* by collecting all the names of the eligible bidders along with their technical proposals. Note that here each application (collected in the set *Applications*) is a tuple containing the name of the bidder, the tender, the bidder's financial proposal, and the bidder's technical proposal.

The Technical Committee returns its evaluation on these technical proposals to the Bidding Portal. We leave this evaluation step unspecified.

```
eval (message) : message;
...
while (Eligible_technical_proposals
                        ->contains(?Bidder,?TP)) {
  Technical_reports->add(Bidder,eval(TP));
  Eligible_technical_proposals->remove(Bidder,TP);
}
```

Here, *eval* is an uninterpreted function, which receives a technical proposal and returns the Technical Committee's evaluation on the proposal.

Then, the Bid Manager receives the financial proposals and the Technical Committee's report, evaluating the quality of the technical proposals of the eligible applicants. Here the Bid Manager never receives the technical details of the applications, and the Technical Committee never receives the financial details of the applications. This separation of concerns is motivated in Deliverable D5.1. [10].

4. **Decision Phase:** Ultimately, the Bid Manager decides which Bidder wins the tender. In general, this decision depends on financial and technical details of the bids. The exact decision procedure is however beyond the scope of this case study. Therefore, in our specification, the decision is modeled by allowing the Bid Manager to choose non-deterministically one of the eligible bidders as the winner of the tender.

```
if(Eligible_applicants->contains(?WBidder))
  send(BP,WBidder);
fi
```

Here, *WBidder* is the name of the winner of the tender. The Bid Manager then communicates the identity of the winner to the Bidding Portal, who will subsequently publish the result.

A complete ASLan v.2 specification of the scenarios described above can be found in Appendix B. The specification is concise, and guides the reader with comments, whenever necessary.

This specification can be automatically translated to ASLan v.1. As the public bidding system contains various data manipulation procedures (such as set constructions, projections, etc.), the resulting ASLan v.1 model has almost 100 transition steps. This case study constitutes a very relevant test for the correctness of the translator and has been an opportunity to evaluate its features and add translation options.

For verification, we have modified the ASLan v.1 model (that is generated by the translator) for efficiency:

- process instances are created statically at start rather than in the steps of the environment entity,
- local variables that are used in only one transition step are removed from the permanent local state,
- step compression (cf. [13]) is performed for the generated model, collapsing transitions that involve only local processing.

All these steps could be performed automatically by an optimizing translator. We consider integrating some of these features in the next platform release.

On the resulting model (which is still complex, with some 50 transitions), we have verified that an honest eligible bidder can complete and win the bid, even in the presence of a malicious, i.e. ineligible, bidder.

```
attack_state compl_good(Actor) :=
   completed(alice)
```

This property is verified by CL-AtSe in 5 seconds, resulting in a 35-step execution trace, representing a successful execution scenario for the bidding process. OFMC also verifies functionality when no ineligible bidder is present, taking about 20 seconds to complete. To handle systems with increasing number and complexity of roles that appear in a service-oriented context, CL-AtSe has been partly re-engineered and optimized for this case, the results being especially apparent for this case study.

In terms of safety requirements, we have verified that the bid cannot be won by an ineligible bidder, represented in the model by Eve. For this purpose, we have rewritten most of the sets used in the model as compound message terms. CL-AtSe verifies the safety property in about 5 minutes.

```
attack_state win_bad(Actor) :=
   isWinner(eve)
```

As the public bidding case study has been the most complex one verified so far in the context of AVANTSSAR, we will investigate additional optimizations and state space reduction techniques to increase the size of systems that can be handled.

In addition to validation, we have also used the case study to test the orchestration platform. We have removed the Bidder entity from the model and added its knowledge and communication capabilities to the intruder, using the client model for orchestration. Starting from the 33-step trace leading to a winning bid, the orchestration platform is able to synthesize a 5-step goal service which reproduces the desired behavior of the bidder, hence successfully solving the orchestration problem.

6.3 Car Registration with access control policies

The Car Registration Process (CRP) Service was introduced in Deliverable D5.1 ("Problem Cases and their Trust and Security Requirements" [10]) as a case study from the Citizen and Service Portals family. It has been used in the project to discuss and prototype the ASLan language features: excerpts of its ASLan v.1 formalization were presented in Deliverable D2.1 [8] and a complete ASLan v.2 formalization was included in Deliverable D2.2 [13].

The main focus for the use of the CRP in testing the AVANTSSAR Platform has been the interplay of workflow and access control policies. The latter are expressed in the model using Horn clauses, for which support in the platform is evolving (fully supported in SATMC and under development in CL-AtSe and OFMC). The goal set for this case study was to orchestrate the workflow abstracting away from policies, and to validate the result taking full account of the policies.

The case study involves four agents: the citizen C, the car registration office RO and one of its employees E, and a central repository CR for customer documents. In the case study, the citizen is viewed as orchestration client, and the goal is to satisfy the registration request. The scenario we would like to obtain as the result of the orchestration is the following:

1. C sends a document (request for registration) to RO
2. RO receives it and stores in a local database of requests
3. E asks to obtain a citizen document to process
4. RO sends E the document of C from the database
5. E processes the request, and if successful asks CR to permanently store it in the central database
6. CR checks if E is entitled to store the processed request and, if so, stores it and returns a confirmation to E
7. E sends the document and status (approval or refusal) to RO
8. RO stores the processing outcome and informs the citizen

Some of the above interactions are governed by role-based access control policies:

- reading and writing documents from/to the local repository of the registration office is allowed for any agent who is an employee;
- writing the fully processed registration request to the central repository requires special permission, which can be given by the registration office head;

- the certification authority (CA) of the car registration office decides who has the role of registration office employee or head. This is expressed by signed certificates.

In the model, we have expressed these policies using rules inspired by the DKAL policy description language [25]. We employ both general rules related to certificates, knowledge and trust, and specific rules describing the local policy of the registration office. They are expressed as Horn clauses and thus evaluated locally in every state of the scenario execution.

```
%%%%%%%%%%% General DKAL-like rules %%%%%%%%%%%
% Anyone knows that anything signed with Certkey
% has been said by Certagent
  hc sign2said(Anyone, Anything, Certagent, Certkey) :=
    knows(Anyone, said(Certagent, Anything)) :-
      pubkey(Certagent, Certkey),
      iknows(pair(crypt(inv(Certkey),Anything),
                  Certagent))

% Anyone knows things said by Certagent if it is trusted
% trustFun0 trusts Certagent on Anyfun with any arg
  hc knowtrustsaid(Anyone, Certagent, Anyfun, Anyarg) :=
    knows(Anyone, apply(Anyfun, Anyarg)) :-
        knows(Anyone, said(Certagent, apply(Anyfun,
           Anyarg))),
        trustsFun0(Anyone, Certagent, Anyfun)
```

The first rule above relates sent certificates (represented here by intruder knowledge) to knowledge of a given agent. It is assumed that any agent can read the sent certificate, and thus knows that the certificate authority has issued it. The second rule is designed to be used with facts (represented here as functions) asserted by a trusted authority over certain agents or objects (here, generic arguments). A given agent `Anyone` trusts a given fact if he knows it has been stated by a certification agent who is trusted on that fact.

```
%%%%%%%%%%% Local policies %%%%%%%%%%%%%%%%%
% Anyone trusts certificate authority to assign roles
  hc trustsRole(Anyone) :=
    trustsFun0(Anyone, ca, hasrole) :- iknows(i)

% Anyone trusts head to issue canstore permissions,
% but must check role of agent (as in next rule)
  hc trustsStore(Anyone, Certagent) :=
    trustsFun(Anyone, Certagent, canstore) :-
```

```
        knows(Anyone, apply(hasrole, pair(Certagent,
           head)))
```

```
% An employee can store if certified by the head
  hc knowsstore(Anyone, Certagent, Someagent) :=
    knows(Anyone, apply(canstore, Someagent)) :-
        knows(Anyone, said(Certagent, apply(canstore,
           Someagent))),
        knows(Anyone, apply(hasrole, pair(Someagent,
           employee))),
        trustsFun(Anyone, Certagent, canstore)
```

The set of local policies has a rule stating that the certificate authority is trusted to assign any roles to any agents. Second, an agent in the role of registration office head is trusted (by anyone) to assign document storing permissions. The last rule is actually used by the central repository to check permission of an agent to store: the agent must be authorized by someone who is trusted on assigning storing permissions, and in addition must be an employee. An actual instance of usage in a protocol rule is given below.

```
%%% The central repository checks permission to store;
%%% if valid, it adds the document and sends an ack
step step_CR(CR,Kcr,CR_DB,SID,E,Ke,Doc) :=
  state_centrep(CR,1,Kcr,CR_DB,SID).
  iknows(crypt(Kcr,pair(crypt(inv(Ke),pair(accept,Doc)),
         E))).
  pubkey(E,Ke).
  knows(CR,apply(canstore,E))
  =>
  state_centrep(CR,1,Kcr,CR_DB,SID).
  pubkey(E,Ke).
  contains(pair(crypt(inv(Ke),pair(accept,Doc)),E),
                CR_DB).
  iknows(crypt(Ke,pair(crypt(inv(Kcr),pair(accept,Doc)),
         CR)))
```

All three back-ends find an orchestration scenario that leads to a successful registration. The trace found by OFMC is presented below in simplified form (collapsing send/receive actions visible to the intruder).

```
i -> (c,1): start_orchestration
(c,1) -> (ro,1): {{doc}_inv(kc).c}_kro
(e,1) -> (ro,1): {readDoc}_inv(ke).e
(ro,1) -> (e,1): {{{doc}_inv(kc).c}_inv(kro).ro}_ke
(e,1) -> (cr,1): {{accepted.doc}_inv(ke).e}_kcr
```

```
(cr,1) -> (e,1): {{accepted.doc}_inv(kcr).cr}_ke
(e,1) -> (ro,1): {{accepted.doc}_inv(ke).e}_kro
(ro,1) -> (c,1): {{accepted.doc}_inv(kro).ro}_kc
(c,1) -> i: end_orchestration
```

The synthesized goal orchestration has 10 steps, and is obtained starting from a similar trace produced by CL-AtSe. It shows that the system can indeed provide the required car registration service to the client, who in its initial transition sends the signed document to the registration office and expects an acknowledged accepted registration in return, after the service has performed the required steps.

For the resulting orchestrated system, we validate some required security properties, as stated in the description of Deliverable 5.1.

- Documents must be secret for anyone who cannot read the repository.

First, we verify that documents cannot be accessed by the intruder:

```
attack_state secret(Doc) :=
  iknows(doc)
```

The actual specification is more precise; to validate it, we define a set `readSet` of all agents who have access to the document while executing the registration; the fact `contains(E, readSet)` is added to all transitions of the agent that processes the document. Then we verify that any member of `readSet` must have a certificate of employee role (which gives read permission), by checking that the following attack state is unreachable:

```
attack_state confidential(Anyone,Doc) :=
  contains(Anyone,readSet).
  not(knows(ro,apply(hasrole,pair(Anyone,employee))))
```

We also verify a second property:

- Documents stored in the repository must be consistent, i.e. their signatures are correct.

For this, we check that it is impossible to have a stored document that is tagged with a client's identity, but the signature does not match:

```
attack_state badsig(C,Kc,Doc) :=
  contains(pair(crypt(inv(Kc),Doc),C),ro_db).
  not(pubkey(C,Kc))
```

All these attack states are immediately dismissed by CL-AtSe which verifies that the system is safe.

Concluding, we have evaluated the AVANTSSAR Platform on a case study where the workflow involves access control policies. With access control decisions encoded directly into the initial state, an execution trace representing the desired orchestration can be found by all three back-ends. With access control policies expressed over certificates sent among participants, the model can be successfully handled by SATMC, which supports Horn clauses. This feature will be extended to all back-ends of the platform.

6.4 Loan Origination Process

The loan origination process (LOP) has been described in Deliverable D5.1 [10] and an excerpt of its formalization in ASLan v.1 has been put forward in Deliverable D2.1 [8]. In Deliverable D2.2 [13] a more complete formalization of the LOP in ASLan v.2 has been presented. Thanks to the features offered by ASLan v.2, the specification focuses on the entities involved in the process and shows how to model the key aspects of the process in ASLan v.2.

However, as ASLan v.1 is the input format of the AVANTSSAR Platform, we consider in this deliverable a formalization in ASLan v.1 so as to focus on a relevant aspect of the LOP scenario: a separate specification of the workflow and the access control policy.

We recall that the LOP describes a bank's evaluation of a customer's request for a loan. As depicted in Figure 9, the workflow we consider starts with the input of the customer's data. Afterwards, a contract for the current customer is prepared while the customer's rating evaluation takes place concurrently. The rating enables the bank to determine whether the customer can be granted the requested loan. To this end, the execution may follow different paths: if the risk associated with the loan is low, i.e. if the internal rating is positive and the amount of the loan is not high, then an internal rating suffices; otherwise, the internal rating is followed by an external evaluation carried out by a Credit Bureau, a third-party financial institution. The loan request must then be approved by the bank. Subsequently, if the customer and the bank have reached an agreement, the contract is signed. Notice that the execution of a task may affect the state of the process. In particular:

- the task `inputCustomerData` may modify the state of the execution by issuing statements asserting if the customer is industrial and if the amount of the loan is high,
- the task `internalRating` may issue a statement asserting if the internal rating of the customer is positive, and

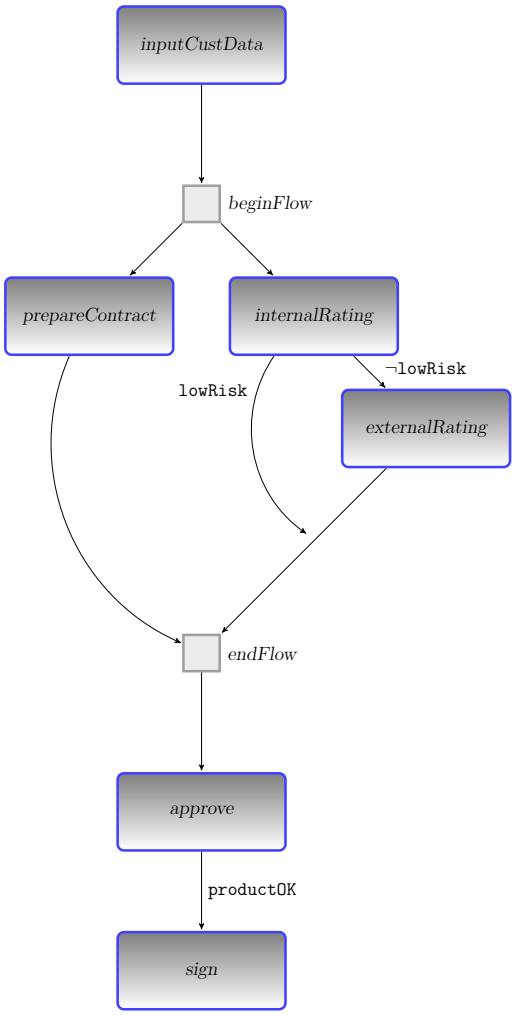

Figure 9: Workflow of the LOP

- the task `approve` may issue a statement asserting if the proposed product is suitable or not for the customer.

Finally, different statements may determine new process features, e.g., the `lowRisk` condition is used to denote a situation in which the internal rating is positive and the amount of the loan is not high. For the sake of simplicity, we do not model the process engine that listens for customer requests and initiates new LOP workflows, but we assume that an instance of the workflow is ready to be executed.

Whether a user can execute a task or not is managed by the access control policy. We consider an access control model based on RBAC [23] extended with delegation. According to the RBAC model, to perform a task a user

Table 2: Permission assignment for the LOP

Task	Role
`inputCustomerData`	`preprocessor`
`prepareContract`	`postprocessor`
`internalRating`	if (`industrialCustomer`) then `supervisor` else `postprocessor`
`externalRating`	`supervisor`
`approve`	if (`industrialCustomer`) then `manager` else `supervisor`
`sign`	if (`intRatingPositive`) then `manager` else `director`

must be assigned a role that is enabled to execute the task and the users must be also active in that role. The roles used in the current specification of the LOP are given in Table 2 together with the tasks they are enabled to execute. We consider delegation by means of *delegation rules* of the form

$$\langle PreConds, ARole, DRole, Task \rangle$$

where *ARole* and *DRole* are roles, *Task* is a task, and *PreConds* is a set of conditions that must hold for the delegation to be applicable. A delegation rule states that if *PreConds* holds then *ARole* can delegate *DRole* to execute *Task*. Notice that this is a "delegation of permission", as defined in [24], where the user being delegated acquires the permission to execute the task but is not obliged to perform it.

The scenario described slightly differs from the process shown in Deliverable D5.1 [10] so to focus on the workflow itself and on the access control policy managing the authorization of users to perform the tasks. In particular, we make the following extensions of the scenario described in Deliverable D5.1 [10]: we consider

- a workflow characterized by the parallel execution of several tasks and by conditional branches,
- non-deterministic effects of tasks possibly affecting the state of the process, and
- a dynamic access control policy: an RBAC model extended with delegation according to which the authorization to perform a task can change during the process execution as the state of the process evolves.

Table 3: Facts and their informal meaning

Fact	Meaning
`ua`(a, r)	user a is assigned to role r
`activated`(a, r)	user a is playing role r
`excludedOwner`(a, t)	user a cannot execute task t
`delegated`(a, r, t)	user a is delegated by a user in role r to perform task t
`pa`(r, t)	role r has the permission to perform task t
`canExecute`(a, r, t)	user a can execute task t by means of role r
`canDelegate`(r_1, r_2, t)	a user in role r_1 can delegate a user in role r_2 to perform task t
`executed`(a, r, t)	user a has executed task t obtaining the authorization from role r
`ready`(t)	task t is ready to be executed
`done`(t)	task t has been executed
`start`	the process is ready to be executed
`lowRisk`	the loan is requested in conditions of low risk
`highValue`	the loan amount is high
`lowValue`	the loan amount is low
`industrialCustomer`	the customer is industrial
`privateCustomer`	the customer is a private citizen
`intRatingPositive`	the customer's internal rating is positive
`intRatingNegative`	the customer's internal rating is negative
`productOK`	the customer and the bank agree on the contract

To illustrate this, we will use the ASLan facts summarized with their informal meaning in Table 3.

In the ASLan v.1 specification, the workflow is expressed by means of rules managing the facts `ready`(T) and `done`(T). A task is marked as `done` when it has been executed, and as `ready` when it is ready to be executed. For example, after the execution of `inputCustomerData` the fact that two tasks, i.e. `prepareContract` and `internalRating`, are ready to be executed can be specified as follows:

```
step prepareContractANDintRating :=
    done(inputCustomerData)
  =>
    ready(prepareContract).
    ready(internalRating)
```

The workflow is also characterized by conditional branches. As an example,

the task `externalRating` is performed after task `internalRating` has been done only if the risk is not low. This can be specified as follows:

```
step extRating :=
    done(internalRating).
    not(lowrisk)
  =>
    ready(externalRating)
```

The execution of tasks by an authorized user is expressed by the following ASLan v.1 rule:

```
step task_execution(T,A,R) :=
    ready(T).
    canExecute(A,R,T).
    not(excludedOwner(A,T))
  =>
    executed(A,R,T).
    done(T)
```

stating that a task `T` can be executed by a user `A` in role `R` if *(i)* `T` is ready, *(ii)* `A` can execute the task `T` by means of role `R` and *(iii)* `A` is not an excluded owner (i.e. a user not allowed to execute task `T`).

As already presented, the execution of a task may affect the state of the process by means of its non-deterministic effects. In this scenario, we consider the execution of tasks as an atomic operation, i.e. we do not model different phases of a task execution. Thus, the non-deterministic effects have to be managed together with the execution of tasks. The way in which we model the non-deterministic effects ensures that if these effects take place, this happens as soon as the task is executed. This can be expressed by adding an additional rule for the execution of tasks characterized by non-deterministic effects. As an example, the approval task modifies the state of the process by issuing a statement asserting if the proposed product is suitable or not for the customer. This can be expressed by adding the following rule to the one previously presented:

```
step task_approve_execution_OK(A,R) :=
    ready(approve).
    canExecute(A,R,approve).
    not(excludedOwner(A,approve))
  =>
    executed(A,R,approve).
    productOK.
    done(approve)
```

As a result, if the rule above is executed, then as effect the product is ok; if the first one is executed, then the product is not ok. Notice that the two rules are indeed mutually exclusive and non-deterministically one of the two is executed.

Rules can be used to specify delegation as well. The delegation rules previously described can be expressed in ASLan v.1 as follows:

```
step delegation(A1,R1,A2,R2,T) :=
    canDelegate(R1,R2,T)
    ua(A1,R1).
    ua(A2,R2)
  =>
    delegated(A2,R1,T)
```

where `canDelegate(R1,R2,T)` expresses the fact that role `R1` can delegate role `R2` to execute task `T`, i.e. `canDelegate(R1,R2,T)` holds if the conditions in *PreConds* hold. The conditions characterizing the delegation rules can be modeled in ASLan v.1 by using Horn clauses. As an example, the conditions of the delegation rule:

$\langle\{\texttt{pa(manager, approve)}, \texttt{intRatingPositive}\}, \texttt{manager}, \texttt{supervisor}, \texttt{approve}\rangle$

can be specified in ASLan v.1 by the following horn clause

```
hc delegation_approveMgrSup :=
  canDelegate(manager,supervisor,approve) :-
                                    pa(manager,approve),
                                    intRatingPositive
```

Notice that in the delegation rules we consider the conditions in *PreConds* to always include the fact `pa(r,t)`, i.e. *ARole* must have the permission to execute *Task* according to the permission assignment relation to be able to delegate the task.

In the ASLan v.1 specification, Horn clauses are also used to model the permission assignment relation and the authorization of users to perform tasks (i.e. `canExecute(A,R,T)`). As an example, the permission to perform the task `internalRating` is expressed by the following Horn clause:

```
hc pa_internalRating_notInd :=
    pa(postprocessor,internalRating) :- privateCustomer

hc pa_internalRating_Ind :=
    pa(supervisor,internalRating) :- industrialCustomer
```

Notice that the access control policy is dynamic as the permission can change over time according to the state of the process, e.g. the permission to perform

task `internalRating` is given to role `postprocessor` if the customer is private and to role `supervisor` if the customer is industrial. The authorization of users to perform tasks is expressed by the following Horn clauses:

```
hc rbac_ac (A,R,T) :=
    canExecute(A,R,T) :- ua(A,R), activated(A,R),
       pa(R,T)

hc delegation_ac (A,R,T) :=
    canExecute(A,R,T) :- delegated(A,R,T)
```

where the former Horn clause represents the classical RBAC model, while the latter expresses the fact that a user can execute a task by being delegated.

The security requirement we consider in the experiments is a Separation of Duty (SoD) property. SoD properties are used for internal control and are probably the most common application-level properties that business processes must comply with in order to mitigate business frauds. SoD amounts to requiring that some critical tasks are executed by different agents. This can be achieved by constraining the assignment of roles (*Static SoD*), their activation (*Dynamic SoD*) or even the execution of tasks [32]. In the experiments, we focus on an object-based Separation of Duty (ObjSoD) requirement according to which no user can perform all the tasks accessing the same object. Since `internalRating`, `externalRating`, and `approve` tasks access and deal with the rating of the customer, they form a set of critical tasks and the LOP is thus expected to meet the following ObjSoD property:

> If the process terminates successfully, then no single user has performed all the critical tasks (namely `internalRating`, `externalRating`, and `approve`).

More formally, we require that if the process terminates successfully, then for all users $a \in \mathcal{A}$ there exists a task $t \in T$, where $T = \{\texttt{internalRating}, \texttt{externalRating}, \texttt{approve}\}$, such that a does not perform t:

$$\mathbf{F}(\texttt{done}(\texttt{sign})) \Rightarrow \bigwedge_{a \in \mathcal{A}} \bigvee_{t \in T} \mathbf{G}\neg\texttt{executed}(a, t). \tag{1}$$

To check if the property is ensured, we verify if there exists an execution path leading to a state where the property is violated. Thus we check if a state in which the task `sign` is done and a user `A` has performed all the tasks `internalRating`,`externalRating`, and `approve` can be reached. This can be specified in ASLan v.1 by the following attack state in the goal section:

```
section goals:
```

```
attack_state objssod(A,R,R1,R2) :=
   executed(A,R,internalRating).
   executed(A,R1,externalRating).
   executed(A,R2,approve).
   done(sign)
```

We consider a scenario characterized by the following users: `david` acting as the director, `mary` and `mark`, playing the role manager, `susan` and `stefan` as supervisors, and `peter`, who can act both as preprocessing clerk and as postprocessing clerk. Thus the initial state contains facts expressing the presented assignment of users to roles.

```
initial_state init1 :=
  start.
  ua(david,director).
  ua(mary,manager).
  ua(mark,manager).
  ua(stefan,supervisor).
  ua(susan,supervisor).
  ua(peter,postproc).
  ua(peter,preproc).
  notisIndustrial.
  nothighValue.
  notintRatingOK
```

Besides the facts representing the user assignment, there are

- `start`, i.e. the workflow is ready to be executed, and
- `privateCustomer`, `lowValue`, `intRatingNegative` which initialize the loan process stating that the customer is private, the value of the loan is low and the rating is negative by default, i.e. the values used in the process unless some tasks modify them.

We have analyzed the presented specification of the LOP in ASLan v.1 with respect to the ObjSoD property by using the AVANTSSAR platform. In particular, it is the validator tool with its back-end SATMC that performed the analysis. As already presented, the ObjSoD property is specified in the ASLan v.1 model as an attack state. If the attack state is reached then the property is violated. Thus, the tool checks if there exists an execution path leading to the attack state, i.e. if it is possible to violate the property. If this is the case, the tool returns a counterexample witnessing the property, i.e a partial-order plan (PPOP) of ASLan rules leading from the initial state to the attack state.

The tool found the counterexample of Figure 10. It is easy to see that the length of the counterexample is 9, i.e. the attack state is reached with a bound k of 9. More in detail, each step represents the execution of a set of ASLan rules, while

```
GOALS: [objssod(stefan,supervisor,supervisor,manager)]
```

reports the name of the attack state reached and shows the user and roles responsible for the violation. As an example,

```
Step 9:[task_execution(sign,mary,director)]
```

represents the execution of the ASLan rule `task_execution(T,A,R)` for task `sign`, user `mary`, and role `director` at the last time step, i.e. `mary` executes tasks `sign` through role `director`. The counterexample shows that the user responsible for the violation of the property is `stefan`, who executes `internalRating` and `externalRating` as `supervisor`, but can nevertheless execute `approve` because a manager, `mark`, delegates him to approve the loan and this leads to the violation. By inspecting the intermediate states of the trace it is easy to conclude that the violation occurs if the customer is industrial, the internal rating is positive, the value of the loan is high, and hence the risk is high. Indeed, in this scenario the permission assignment relation in Table 2 allows a `supervisor` to perform `internalRating` and `externalRating` and the delegation rule previously presented to be executed.

To avoid this violation, we constrained the applicability of the delegation rule by conjoining its applicability condition with the literal `lowValue` to obtain a more restrictive policy in case the value of the loan is high. The validator does not find any counterexamples to the new specification.

```
GOALS: [objssod(stefan,supervisor,supervisor,manager)]

Step 0:[inputCustomerData,
        activate(peter,preproc)]
Step 1:[activate(peter,postproc),
        task_inputCustData_execution_hValInd(peter,preproc),
        delegation(david,mary,director,manager,sign)]
Step 2:[prepareContractANDintRating,
        activate(stefan,supervisor)]
Step 3:[task_intRating_execution_Pos(stefan,supervisor),
        task_execution(prepareContract,peter,postproc)]
Step 4:[extRating]
Step 5:[task_execution(externalRating,stefan,supervisor)]
Step 6:[approve_notlowRisk,
        delegation(mark,stefan,manager,supervisor,approve)]
Step 7:[task_approve_execution_OK(stefan,manager)]
Step 8:[sign]
Step 9:[task_execution(sign,mary,director)]
```

Figure 10: Counterexample to ObjSoD

7 Conclusion

We have presented a prototypical implementation of the AVANTSSAR Platform, which we have implemented as a service-oriented architecture. We have described the two main components of the platform, the Orchestrator and the Validator, as well as the service orchestration problem, the common interface we use for specifying problems for the Orchestrator.

In order to test the current functionalities of the platform, in this deliverable we have also described the experimental results obtained by running the platform against a selection of problem cases taken from Deliverable D5.1 [10], We have given here excerpts of the case studies, while the complete formalization of the problem cases will be the objective of WP 5.2. The assessment of the AVANTSSAR Platform (within WP 5.4) against the validation problems will provide valuable feedback on the usability, effectiveness, and efficiency of the tools we have been developing in WP 4. The improvement of the reasoning techniques developed and automated in WP 3 and WP 4, as result of the assessment phase, will be integrated in the final version of the platform, which we will present in D4.2.

This deliverable mainly focused on the logical level on which the platform operates. The definition of the connectors to the application level will be explicitly addressed in the Industry Migration workpackage WP 6, which takes current industrial best practice languages and models into account. The

outcome of this work will be integrated in the final version of the platform.

As future work, we plan to fix the current limitations of the platform that we have described in this deliverable. In particular, we will integrate into the Orchestrator the function for generating the *Wrappers*.

References

[1] Apache Software Foundation. Apache Axis2 - Web Services Engine. http://ws.apache.org/axis2/, 2004.

[2] Apache Software Foundation. Apache Rampart - Axis2 Security Module. http://ws.apache.org/rampart/, 2005.

[3] Apache Software Foundation. Apache Ode - Orchestration Director Engine. http://ode.apache.org/, 2006.

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

[5] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. T. 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.

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

[7] C. Arora and M. Turuani. Validating integrity for the ephemerizer's protocol with cl-atse. In *Formal to Practical Security: Papers Issued from the 2005-2008 French-Japanese Collaboration*, pages 21–32, Berlin, Heidelberg, 2009. Springer-Verlag.

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

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

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

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

[12] AVANTSSAR. Deliverable 6.2.2: Industrial language requirements. Available at http://www.avantssar.eu, 2008.

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

[14] P. Balbiani, F. Cheikh, and G. Feuillade. Composition of interactive web services based on controller synthesis. *Services - Part I, 2008. IEEE Congress on*, pages 521–528, July 2008.

[15] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[16] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of TACAS'99*, LNCS 1579, pages 193–207. Springer-Verlag, 1999.

[17] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *csfw01*, pages 82–96. ieeecoso, 2001.

[18] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proceedings of CSFW'03*, pages 126–140. IEEE Computer Society Press, 2003.

[19] Y. Boichut, P.-C. Heam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay Technique to Automatically Verify Security Protocols. In *Proceedings of Automated Verification of Infinite States Systems (AVIS'04)*, ENTCS, to appear.

[20] A. Brucker and S. Mödersheim. Integrating automated and interactive protocol verification. In *Proceedings of FAST 2009*, Lecture Notes in Computer Science. Springer-Verlag, 2009 (To appear). Extended version available as IBM Research Report RZ3750.

[21] Y. Chevalier, M. Mekki, and M. Rusinowitch. Orchestration under security constraints. In *Sixth International Workshop on Formal Aspects in Security and Trust (FAST2009) Eindhoven, the Netherlands, November 5-6, 2009*, 2009.

[22] Y. Chevalier and M. Rusinowitch. Compiling and securing cryptographic protocols. *CoRR*, abs/0910.5099, 2010. to appear.

[23] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992. http://citeseer.ist.psu.edu/ferraiolo92rolebased.html.

[24] P. Giorgini, F. Massacci, and J. Mylopoulos. Modeling security requirements through ownership, permission and delegation. In *Proc. of RE'05*, pages 167–176. IEEE Press, 2005.

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

[26] S. M. Hansen, J. Skriver, and H. R. Nielson. Using static analysis to validate the SAML single sign-on protocol. In *WITS '05: Proceedings of the 2005 workshop on Issues in the theory of security*, pages 27–40, New York, NY, USA, 2005. ACM Press.

[27] S. Mödersheim. On the Relationships between Models in Protocol Verification. *Journal of Information and Computation*, 206(2–4):291–311, 2008. http://dx.doi.org/10.1016/j.ic.2007.07.006.

[28] S. Mödersheim. Algebraic Properties in Alice and Bob Notation. In *Proceedings of Ares 2009*, pages 433–440. IEEE Computer Society Press, 2009. DOI:http://doi.ieeecomputersociety.org/10.1109/ARES.2009.95. An extended version is available as Technical Report no. RZ3709, IBM Zurich Research Lab, 2008, domino.research.ibm.com/library/cyberdig.nsf.

[29] S. Mödersheim and L. Viganò. Secure Pseudonymous Channels. Technical Report RZ3724, IBM Zurich Research Lab, 2008. domino.research.ibm.com/library/cyberdig.nsf.

[30] S. Mödersheim and L. Viganò. The Open-source Fixed-point Model Checker for Symbolic Analysis of Security Protocols. In *Fosad 2007-2008-2009*, volume 5705 of *LNCS*, pages 166–194. Springer-Verlag, 2009.

[31] S. Mödersheim, L. Viganò, and D. Basin. Constraint Differentiation: Search-Space Reduction for the Constraint-Based Analysis of Security Protocols. *Journal of Computer Security*, 2009 (to appear).

[32] A. Schaad, V. Lotz, and K. Sohr. A model-checking approach to analysing organisational controls in a loan origination process. In *SACMAT 2006,11th ACM Symposium on Access Control Models and Technologies, Lake Tahoe, California, USA, June 7-9, 2006, Proceedings*, pages 139–149. ACM, 2006.

[33] M. Turuani. The CL-Atse Protocol Analyser. In *Term Rewriting and Applications (Proceedings of RTA'06)*, LNCS 4098, pages 277–286, 2006.

[34] WSO2. Web Services Framework for PHP. http://wso2.org/projects/wsf/php, 2006.

A Input Format Syntax

Here a list of points to be respected for specification of orchestration problem is presented.

- for a state of any service, use a fact with a name starting with `state_` followed by a service name, e.g. `state_Service1`;
- use `state_OrchestrationGoal`(<list of parameters>) as a predicate representing the state of a service to be orchestrated;
- use `state_OrchestrationClient`(<list of parameters>) as a predicate representing the state of the Client service;
- predicates `state_OrchestrationGoal` and `state_OrchestrationClient` are not allowed to appear in the same transition; we need separate transitions for these two key services;
- to specify constraints for the orchestration in the form of an LTL formula, use the goal identifier `orchestrationConstraint`;
- to specify a state representing successful orchestration, use the `attack_state orchestrationFinalState`;
- terms used as parameters of fact predicates `state_OrchestrationGoal` and `state_OrchestrationClient` placed in the initial state are considered as an initial knowledge of the Goal and Client services, respectively. Not more than one initial state of the Goal and not more than one of the Client should be specified;
- sending and receiving of messages are modeled by `iknows` facts: the `iknows` in the LHS of a transition stands for receiving a message, while in the RHS of a transition it stands for sending a message;
- the application of new message-level function symbols are modeled via `apply(new_function_name, argument)` instead of `new_function_name(argument)`, where `new_function_name` should be of `function` or `hash_func` type. Note, that it does not concern the facts, but only messages (terms appeared inside `iknows`).
- Note also, that a unique initial state is required. More than one initial state is allowed for the validation of security properties but for the orchestration it has no meaning.

- All the other goals (LTL formulas with name different from `orchestrationConstraint` and attack states with name different from `orchestrationFinalState`) are considered as security properties to be validated.

B The Public Bidding Case Study in ASLan v.2

```
entity Environment {

 symbols

  %% Messages

  applicants_download_request : message;
  technical_proposals_download_request : message;
  reports_download_request : message;
  winner_request : message;

  initiate(agent,agent,message) : message;
  submission_request (agent) : message;
  application(agent,message,message,message) : message;
  eval (message) : message;

  %% Participants

  bp, bm, tc, alice, eve : agent;

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

 entity BidManager(Actor, TC, BP: agent) {

  symbols
   Tender : message;

   Applicants : (agent) set;
   Eligible_applicants : (agent) set;
   Financial_reports : (agent,message,message) set;

   Bidder : agent;
   WBidder: agent;

   is_eligible (agent) : fact;

  body {
```

```
   %% Eligible bidders
   is_eligible(alice);

   Tender := fresh();
   %% Publication phase
   send(BP, initiate(Actor,TC,Tender));

   %% Evaluation phase
   send(BP, applicants_download_request);
   receive(BP, ?Applicants);


   while (Applicants->contains(?Bidder)) {
     if (is_eligible(Bidder))
       Eligible_applicants->add(Bidder);
     fi

    Applicants->remove(Bidder);
   }
   send(BP,Eligible_applicants);

   %% Decision phase
   send(BP,reports_download_request);
   receive(BP,?Financial_reports);

   %% Choice abstracted away
   %% In general WBidder is a function of Financial_reports
   if(Eligible_applicants->contains(?WBidder))
     send(BP,WBidder);
   fi
  }
 }

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

 entity Bidder(Actor, BP: agent) {

  symbols
   Tender : message;
```

```
   TP : message;
   FP : message;
   Winner : agent;
   completed(agent) : fact;

  body {

   %% Submission phase
   send(BP, submission_request(Actor));
   receive(BP,?Tender);
   FP := fresh();
   TP := fresh();
   send(BP,application(Actor,Tender,FP,TP));


   %% Decision phase
   send(BP,winner_request);
   receive(BP,?Winner);
   completed(Actor);
  }
 }

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

 entity TechnicalCommittee(Actor, BP: agent) {

  symbols
   Eligible_technical_proposals : (agent,message) set;
   Technical_reports : (agent,message) set;

   TP : message;
   Bidder: agent;

  body {


   %% Evaluation phase
   send(BP,technical_proposals_download_request);
   receive(BP,?Eligible_technical_proposals);

   while (Eligible_technical_proposals->contains(?Bidder,?TP)) {
```

```
      Technical_reports->add(Bidder,eval(TP));
      Eligible_technical_proposals->remove(Bidder,TP);
    }

    send(BP,Technical_reports);

  }
}

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

entity BidPortal(Actor, BM, TC : agent) {

  symbols
   submission_phase_open : fact;
   announcement_phase_on : fact;
   isWinner (agent) : fact;

   Tender : message;
   Applications : (agent,message,message,message) set;
   Applicants : (agent) set;
   Eligible_applicants : (agent) set;
   Eligible_technical_proposals : (agent,message) set;
   Technical_reports : (agent,message) set;
   Financial_reports : (agent,message,message) set;

   Bidder : agent;
   Winner : agent;
   Report : message;
   FP : message;
   TP : message;

  entity SubmissionTimer {

   body {

    submission_phase_open;
    retract submission_phase_open;
   }
  }
```

```
body{

 %% Publication phase
    receive(BM,initiate(BM,TC,?Tender));


 new SubmissionTimer;

 while (submission_phase_open) {
  select {
   on receive(?Bidder,submission_request(Bidder)) : {
    send(Bidder,Tender);
   }
   on (receive(?Bidder,application(Bidder,Tender,?FP,?TP))
    & !(Applications->contains(Bidder,Tender,FP,TP))) : {
    Applications->add(Bidder,Tender,FP,TP);
    Applicants->add(Bidder);
   }
  }
 }

 %% Evaluation phase
 receive(BM, applicants_download_request);
 send(BM,Applicants);
 receive(BM,?Eligible_applicants);

 receive(TC, technical_proposals_download_request);
 while (Eligible_applicants->contains(?Bidder)) {
   if (Applications->contains(Bidder,?,?,?TP))
     Eligible_technical_proposals->add(Bidder,TP);
   fi

   Eligible_applicants->remove(Bidder);
 }

 send(TC,Eligible_technical_proposals);
 receive(TC,?Technical_reports);

%% Decision phase
 receive(BM,reports_download_request);
```

```
   while (Technical_reports->contains(?Bidder,?Report)) {
     if (Applications->contains(Bidder,?,?FP,?))
       Financial_reports->add(Bidder,FP,Report);
     fi

     Technical_reports->remove(Bidder,Report);
   }
   send(BM,Financial_reports);
   receive(BM,?Winner);
   isWinner(Winner);

   %% Announcement
   announcement_phase_on;
   while (announcement_phase_on) {
    receive(?Bidder,winner_request);
    send(Bidder,Winner);
   }
 }
}


 body {

  new BidPortal(bp,bm,tc);
  new BidManager(bm,tc,bp);
  new TechnicalCommittee(tc,bp);
  new Bidder(alice,bp);
  new Bidder(eve,bp);
 }
}
```