Distributed Orchestration of Web Services under Security Constraints[⋆]

Tigran Avanesov[1], Yannick Chevalier[2], Mohammed Anis Mekki[1], Michaël Rusinowitch[1], and Mathieu Turuani[1]

[1] INRIA Nancy Grand Est,
615 allée du jardin botanique,54000 Vandœuvre-lès-Nancy, France
{avanesot, mekkimoh, rusi, turuani}@loria.fr
[2] Université de Toulouse,
118 route de Narbonne, F-31062 Toulouse, France
ychevali@irit.fr

Abstract. We present a novel approach to automated distributed orchestration of Web services tied with security policies. The construction of an orchestration complying with the policies is based on the resolution of deducibility constraint systems and has been implemented for the non-distributed case as part of the AVANTSSAR Validation Platform. The tool has been successfully experimented on several case-studies from industry and academia.

Keywords: Web services, automatic composition, security, distributed orchestration, formal methods

1 Introduction

Composability, one of the basic principles and design-objectives of *Service-oriented Architecture (SOA)* expresses the need for providing simple scenarios where already available services can be reused to derive new added-value services. SOA in its SOAP Web services incarnation based on XML messaging and relying on a rich stack of related standards provides a flexible yet highly interoperable solution to describe and implement a variety of e-business scenarios involving different services possibly bound to complex security policies. Therefore automated solutions should be considered for composition to realize scalability since the composed service can be very complex either to discover or even to describe, especially if some security constraints are to be respected.

Mainly two approaches to Web service composition have been considered, namely orchestration and choreography [25]. In the former a unique business process, called a *Mediator*, aggregates the existing services, while in the latter each service is responsible for implementing its part of the composed service.

We present in this paper a scalable Web service composition approach relying on the notion of *partner* corresponding to an organization. Each partner

⋆ This work is supported by FP7 AVANTSSAR [1] and FP7 NESSoS [2] projects.

in a composition implements its own part of the orchestration. In this setting standard orchestration is a special case in which only one partner is involved, whereas choreography is another case in which there is one partner per available service. Several related "distributed orchestration" notions have been advocated for in the literature (e.g. [4]). However in inter-organizational business processes it is crucial to protect sensitive data of each organization providing a component service in some orchestration, and our main motivation is to advance the state of the art by taking into account the security policies while computing an orchestration.

Contributions First, we present a formal framework to model Web services, their security policies and their intercommunication. We consider a rich structure for Web services' messages including ciphered texts to ensure non-disclosure policies. We show that in this setting the distributed orchestration problem appears to be non trivial even for linear workflows. Second, we propose an algorithm to solve the distributed orchestration problem in this setting. Our decidability result relies on advanced symbolic constraint solving techniques. Finally the paper reports a freely available prototype implementation of the automatic Web services composition approach for the non-distributed case. This prototype is one of the few automatic tools (like [29,12]) that are able to orchestrate Web services.

Related work A common approach to *static* [8] (or *syntactic* [27]) Web services compositions is Orchestration. Web services orchestration [24] deals with a central entity called *orchestrator* or *mediator* that operates as a glue between the Client and the community of available services.

Most works on service composition rely on a *behavior model* [11], i.e. where services are considered as stateful and used by the Client according to some scenario. The behavior is usually presented as a Kripke structure, where transitions are labeled with services' operations [22]. Here we will give a small overview of related works.

A composition based on conversations (where a conversation is a sequence of messages exchanged by peers) was presented in [9]. The available services are represented by a finite set of message classes, a finite set of abstract peers (services) and a finite set of directed channels. Each channel consists of two endpoints and a set of message classes allowed to be sent over this channel. The composition problem amounts then to find a Mealy machine for each peer such that the set of possible conversations that can be seen by an external observer is equivalent to some given conversation specification.

Another work on the same lines is [7]. The authors proposed a way to synthesize and verify cryptographic protocols given a set of multiparty sessions that represent possible conversations between the participants. Given multiparty sessions represented as directed graphs whose nodes are labeled with roles and edges labeled with message descriptor and two sets of typed variables: the first states the variables in which a role from the source node can write, and the second — from which the destination role can read. They first build the projection for each role and then reinforce it by adding security layer in order to guarantee integrity and secrecy properties. Moreover, a prototype of the compiler implementing this approach generates interfaces and implementations for the generated protocols in ML.

Note that these approaches aim to *generate* implementations of the participants with given behavior, while in this paper we focus on how to *compose* automatically existing ones in order to synthesize a service that is able to satisfy a given client.

In the *Roman Model* [11], the available services are specified as finite state machines. In [22] the author extends the service model from deterministic [6] to non-deterministic state machines allowing shared memory. Moreover, the approach allows one to find a *finite orchestrator generator* that can derive a set of all possible orchestrations. Note that in all these works services accept only data from a finite domain. This has motivated an extension called *COLOMBO* [5] of Web services composition which deals with messages from infinite domains. The composition problem is equivalent to finding a *Mediator* service which uses messages to interact with the available services and the Client such that the overall behavior of the mediated system faithfully simulates the behavior of given goal service. However, even in this model the Mediator is not able to handle the cryptographic primitives (e.g. decrypt an encrypted message).

In many cases a single entity (device, organization) is not able to orchestrate the Web services due to the lack of resources (e.g. absence of data requested by available services) or because of access limitations: some services are limited to a protected private network. But if partner organizations are involved in the orchestration, every party can contribute to satisfy client requests. In *distributed* or *decentralized orchestration* (e.g. [23] and [22,26] for other alternative approaches), each partner can invoke his available services and also communicate with other partners. In this way a Mediator is distributed between the partners, but still we have a dedicated one to communicate with the Client. However even in cooperation mode sensitive data should not be propagated beyond the organizational border (a company will not share secrets with partners). This is why communication between partners must be restricted. We will show below how distributed orchestration is still possible in such constrained setting. For the non-distributed case and without implementation some initial ideas were presented in [14].

Paper organization In Section 2 we introduce some basic notions and present an example of non-distributed Web services orchestration; Section 3 starts with an example of distributed orchestration, then our formal model is explained, and ends with presenting our method for solving distributed orchestration problems. Section 4 reports our implementation for the non-distributed case.

2 The AVANTSSAR Approach

In this section we present an approach used in *AVANTSSAR Orchestrator*, a tool for automatic orchestration of Web services.

2.1 Web service model

Web services can be described at two levels: *(i)* The *profile*, a precise description of the interface exposed by a service in terms of a set of operations it provides, their corresponding in-bound and out-bound message patterns and possibly their security policies. From the point of view of standards, this information corresponds to the *WSDL* part of the service specification. *(ii)* The *behavior*, or the use-case scenario of the interface exposed by the service, e.g. a sequencing in the calls of the operations provided by the service. This is typically covered by the *BPEL* part of the service specification.

We use first-order terms to describe the profile of a Web service as they capture an interesting fragment of the *XML Schema* and *WS-SecurityPolicy*. On the other hand, for the behavior part of a Web service we use a transition system capturing its workflow logic. For the sake of simplicity we will assume only linear workflows (i.e. sequential workflows without branching) in this section.

We consider an abstraction of Web services as sequences of *actions*, where an action is either receiving or sending of a message pattern. We write $?r$ for a reception of message r, and $!s$ for an emission of message s. We call a finite sequence of actions a *strand* [17]. To simplify we consider only *normal strands*, i.e. starting from a ? action, ending by a ! action and alternating ? with ! actions. They can be viewed as *synchronous* Web services, that is where each request imply an immediate reply.

The execution of consecutive receive-send actions $?r.!s$ in a normal strand together with the corresponding send and receive actions of the caller is called an *invocation* of a synchronous service. We express message patterns (like s and r) as first-order terms. More precisely, we consider an infinite set of free constants $\mathcal{C}$ and an infinite set of variables $\mathcal{X}$. Given a signature $\mathcal{F}$ (*i.e.* a set of function symbols with arities) we denote by $\mathrm{T}(\mathcal{F},\mathcal{X})$ the set of terms over $\mathcal{F} \cup \mathcal{C} \cup \mathcal{X}$ defined recursively as follows: *(i)* $\mathcal{C} \cup \mathcal{X} \subseteq \mathrm{T}(\mathcal{F},\mathcal{X})$ *(ii)* $\forall f \in \mathcal{F}\ \forall t_1,\ldots,t_n \in \mathrm{T}(\mathcal{F},\mathcal{X})$ implies $f(t_1,\ldots,t_n) \in \mathrm{T}(\mathcal{F},\mathcal{X})$, where n is the arity of f. Given a term t we denote by $\mathrm{Sub}\,(t)$ the set of its subterms defined recursively as follows: *(i)* $t \in \mathrm{Sub}\,(t)$ *(ii)* $t = f(t_1,\ldots,t_n)$ implies $\bigcup_{i=1,\ldots,n} \mathrm{Sub}\,(t_i) \subseteq \mathrm{Sub}\,(t)$. We denote by $\mathrm{Vars}\,(t)$ the set of variables occurring in term t, i.e. $\mathrm{Vars}\,(t) = \mathrm{Sub}\,(t) \cap \mathcal{X}$. The set of ground terms (or messages) over $\mathcal{F}$ is denoted by $\mathrm{T}(\mathcal{F})$ and defined as $\mathrm{T}(\mathcal{F}) = \{t \in \mathrm{T}(\mathcal{F},\mathcal{X}) : \mathrm{Vars}\,(t) = \emptyset\}$. A substitution σ is a mapping from $\mathcal{X}$ to $\mathrm{T}(\mathcal{F},\mathcal{X})$; a ground substitution is a mapping from $\mathcal{X}$ to $\mathrm{T}(\mathcal{F})$. The application of a substitution σ to a term t (resp. a set of terms E) is denoted $t\sigma$ (resp. $E\sigma$) and is equal to the term t (resp. E) where all variables x have been replaced by the term $x\sigma$. The list of predefined symbols and their meanings are given in Table 1.

Security policies (e.g. WS-SecurityPolicy standard's integrity or confidentiality assertions) are expressed in message patterns. Consider for instance a service that receives any value X and returns the result of function f on this value. This service is specified by the normal strand $?X.!f(X)$. Now if the security policy requires that incoming messages should be encrypted with the public key K_S

Table 1. Predefined functional symbols

Term	Description
$\text{enc}(t_1, t_2)$	t_1 encrypted with symmetric key t_2
$\text{aenc}(t_1, t_2)$	t_1 encrypted with asymmetric public key t_2
$\text{pair}(t_1, t_2)$	t_1 concatenated with t_2
$\text{priv}(t_2)$	corresponding private key for public key t_2
$\text{sig}(t_1, \text{priv}(t_2))$	signature of message t_1 with key $\text{priv}(t_2)$

Table 2. Dolev-Yao deduction rules

Composition rules	Decomposition rules
$t_1, t_2 \to \text{enc}(t_1, t_2)$	$\text{enc}(t_1, t_2), t_2 \to t_1$
$t_1, t_2 \to \text{aenc}(t_1, t_2)$	$\text{aenc}(t_1, t_2), \text{priv}(t_2) \to t_1$
$t_1, t_2 \to \text{pair}(t_1, t_2)$	$\text{pair}(t_1, t_2) \to t_1$
$t_1, \text{priv}(t_2) \to \text{sig}(t_1, \text{priv}(t_2))$	$\text{pair}(t_1, t_2) \to t_2$

of that service then the corresponding strand associated to the service will be $?\,\text{aenc}(X, K_S)\,.\,!f(X)$ instead.

2.2 Web service orchestration

Following [6,5] we call a *Mediator* a service that adapts and dispatches requests from a client to the community of available services. We state the orchestration problem as follows: given a set of available services (represented as strands), a client (also represented as a strand) and an initial knowledge of a *Mediator* (a service to be composed), one must find a feasible (with regard to elementary Dolev-Yao operations, see Table 2, that the service can execute internally) communication between the Mediator, Client and available services, such that all requests of the Client are satisfied by the Mediator service.

Orchestration example Assume there is a demand on translating texts from French to English. It is known that the texts were obtained by automatically recognized hand-written documents and thus contain some misspells. The client's need is to send a text in French and receive back an English translation of the preliminary corrected text, i.e. the client specification is: $!t.\,?en(corr(t))$.

The available services are

1. SpellChecker: a service that corrects spelling (e.g. using the semantically closest word from list of possible options). Its model: $?T.\,!\,\text{pair}(corr(T), n(T))$, where T is a text, $corr(T)$ is the corrected text, $n(T)$ is a number of corrections done.
2. Translator: a service producing an automatic translation of given text from French to English. Its security policy requires all incoming messages to be encrypted with its public key. The specification is: $?\,\text{aenc}(M, K_{tr})\,.\,!en(M)$, where M is a text, $en(M)$ is a translation of M into English and K_{tr} is a public key of the Translator.

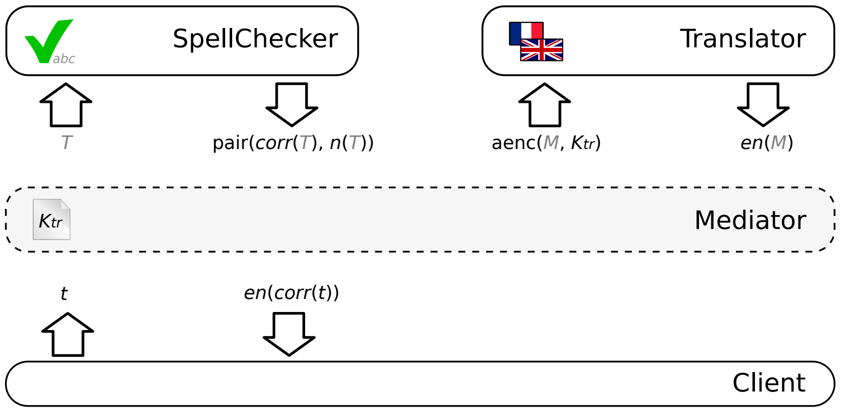


Fig. 1. Illustration for the orchestration example

We assume that the Mediator initially knows only a public key of the Translator K_{tr}. The question is how to satisfy the Client's request? The problem is illustrated in Fig. 1.

2.3 Reduction of Orchestration to Protocol State Reachability

The example shows that an orchestration problem can be reduced to checking whether there is a reachable state where the Client has received a reply on his request. Our idea is to adopt techniques from security protocol analysis, and thus to open the possibility of reusing existing tools, in order to solve the orchestration problem expressed as a reachability problem. The Mediator is a new service (that we try to generate) that organizes the communications between the given services and the Client in such a way, that the Client reaches a final acceptable state. The Mediator is conceptually similar to the Intruder [16] of Dolev-Yao model of cryptographic protocols: the Dolev-Yao intruder tries to communicate with a honest agent, by playing some roles or faking some messages in order to reach an unsafe (or attack) state, e.g. when the honest agent sends a message from which the intruder is able to infer a secret key. We show some correspondence between entities of WS Orchestration and Protocol Analysis problem:

$$Services \left\{ \begin{array}{rcl} \text{Available service/Client} & \sim & \text{Protocol role} \\ \text{Mediator} & \sim & \text{Intruder} \\ \text{Final state} & \sim & \text{Attack state} \end{array} \right\} Protocols$$

Tool `CL-AtSe` [30,28] for finding attacks on protocols is placed at the core of AVANTSSAR Orchestrator. Let us survey the technique that underlies CL-AtSe on the example of § 2.2. First, the tool selects a linear order on service invocations and client calls that is compatible with their individual workflow. Suppose the following sequence has been selected:

$$\left\{\begin{array}{lcr} \{K_{tr}, t\} & \rhd & T \\ \{K_{tr}, t, \mathsf{pair}\,(corr(T), n(T))\} & \rhd & \mathsf{aenc}\,(M, K_{tr}) \\ \{K_{tr}, t, \mathsf{pair}\,(corr(T), n(T))\,, en(M)\} & \rhd & en(corr(t)) \end{array}\right\} \begin{array}{r} (1) \\ (2) \\ (3) \end{array}$$

Fig. 2. A constraint system (T, M are variables) describing a possible orchestration.

1. Receive a request from the Client;
2. Invoke SpellChecker;
3. Invoke Translator;
4. Send a response to the Client.

Then, the tool generates a system of *deducibility constraints* from the specification of the services and the selected interleaving (see Fig. 2). A constraint is a couple denoted by $E \rhd t$, where E is a set of terms and t is a term. A ground substitution σ is a solution of the constraint $E \rhd t$, if $t\sigma \in \text{Der}\,(E\sigma)$, where $p \in \text{Der}\,(T)$ means that ground term p is *derivable* from set of ground terms T: formally $\text{Der}\,(T)$ is the smallest set T' containing T and such that for all $u, v \in T'$ and w such that $u, v \rightarrow w$ or $v \rightarrow w$ for some rule in Table 2 then $w \in T'$. Likewise, a ground substitution σ is a solution of a constraint system $\mathcal{S}$ iff σ is a solution of every constraint in $\mathcal{S}$.

Now we explain how the constraint system displayed in Fig. 2 is built. First the intruder receives t from the Client, i.e. his knowledge becomes $\{K_{tr}, t\}$. Second, in order to send a request to SpellChecker, the intruder must deduce some T from his knowledge (Constraint 1). Third, when he obtains the response, he adds it to his knowledge and tries to build a request acceptable by the Translator, i.e. that matches the expected pattern (Constraint 2). Finally, when he receives the response from the Translator, he tries to deduce the response for the Client (Constraint 3).

The constraint systems built by this procedure are *well-formed*, that is the sets on left-hand side of constraints are increasing (*knowledge monotonicity* property), and each variable appears first in the right-hand side of some constraint (*variable origination* property). A lot of work was done on the resolution of the constraints of this type (e.g [13,10]).

The unique solution of the constraint system in Fig. 2 is a substitution $\{T \mapsto t; M \mapsto corr(t)\}$, that can be interpreted as follows: first Mediator sends to the SpellChecker t (text received from the Client), then he receives a response: $\mathsf{pair}\,(corr(t), n(t))$. As the Mediator can decompose a pair he extracts the needed first part $(corr(t))$ and encrypt it with K_{tr}. The result is sent to the Translator that replies with $en(corr(t))$, the message expected by the Client. Thus, Mediator can forward it and complete.

We have demonstrated some abilities of the Mediator. In this simple example it *decomposed* a concatenated message to throw away a part that is not needed (here, number of corrections done) and *encrypted* a message with necessary public key in order to adapt it in such way that the resulting message will be accepted by the service, since the service's policy is satisfied.

In the next example we show how *several* mediators may collaborate. In order to solve the analogous problem in these settings, a technique for resolution of well-formed constraints is not enough. Since we have to take into account knowledges of several different composers working in parallel, the constraint systems obtained in this case will probably violate the knowledge monotonicity property, and thus will not be well-formed.

3 Distributed orchestration under security constraints

We extend the previous section and we show how to reduce the distributed orchestration problem to the resolution of deducibility constraints. The main difference is that the resulting constraints are not well-formed.

For the distributed orchestration we will consider multiple cooperating mediators, called *partners*. We distinguish one of them (still called Mediator) who communicates with the client, while all others do not. The partners are free to invoke available services, but the cooperation between them is restricted by communication patterns and *non-disclosure* policy conditioned by inter-organizational relationships (no sensitive data must be propagated to other organizations).

3.1 Distributed orchestration example

Suppose that the available services are not free and can serve only registered users (for the reason of simplicity we suppose that the Translator and the SpellChecker have a unique registered user each). Moreover, the Mediator has no credential to log in and use the Translator service, but it has an account for the SpellChecker (note that both Translator and SpellChecker are still reachable).

Fortunately, there is a partner who has an account for the Translator and can help to satisfy the client's requests (see Fig. 3), but does not want to reveal his credentials to the Mediator.

The corresponding specification of the Translator is:

$$?\,\mathrm{aenc}\left(\mathrm{pair}\left(\mathrm{pair}\left(usr_{tr}, pwd_{tr}\right), M\right), K_{tr}\right).!en(M),$$

where usr_{tr} is a login of the registered user and pwd_{tr} is the corresponding password; K_{tr} is a public key of the Translator.

And the corresponding specification of the SpellChecker is:

$$?\,\mathrm{aenc}\left(\mathrm{pair}\left(\mathrm{pair}\left(usr_{sc}, pwd_{sc}\right), T\right), K_{sc}\right).!\,\mathrm{pair}\left(corr(T), n(T)\right),$$

where usr_{sc} is a login of the registered user and pwd_{sc} is the corresponding password; K_{sc} is a public key of the SpellChecker.

We suppose that Mediator and Partner share a symmetric key k and all communications between them are encrypted with the shared key.

The problem is, again, to find a feasible communication scenario between all the parties, such that *(i)* all requests of the Client are satisfied and *(ii)* no partner can extract a sensitive data of another partner from a message received

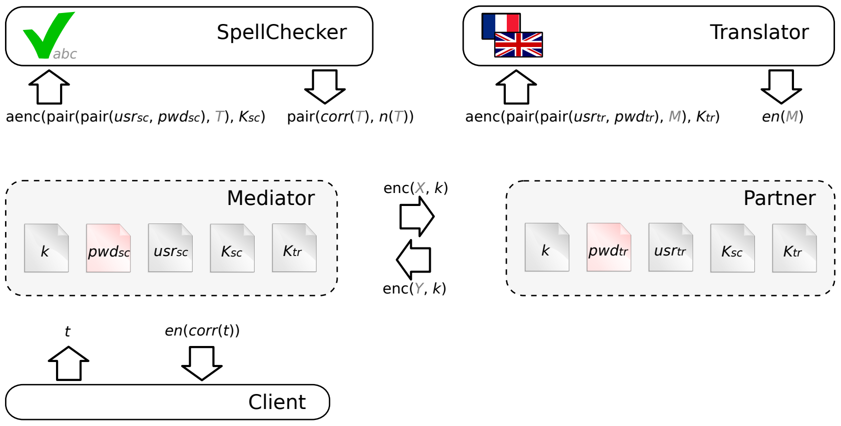


Fig. 3. Illustration for the distributed orchestration problem example

$$\left\{\begin{array}{ll}
\{K_{tr}, K_{sc}, usr_{sc}, pwd_{sc}, k, t\} \triangleright & \mathrm{aenc}\left(\mathrm{pair}\left(\mathrm{pair}\left(usr_{sc}, pwd_{sc}\right), T\right), K_{sc}\right) \\
\{K_{tr}, K_{sc}, usr_{sc}, pwd_{sc}, k, t, \mathrm{pair}\left(corr(T), n(T)\right)\} \triangleright & \mathrm{enc}\left(X, k\right) \\
\{K_{tr}, K_{sc}, usr_{tr}, pwd_{tr}, k, \mathrm{enc}\left(X, k\right)\} \triangleright & \mathrm{aenc}\left(\mathrm{pair}\left(\mathrm{pair}\left(usr_{tr}, pwd_{tr}\right), M\right), K_{tr}\right) \\
\{K_{tr}, K_{sc}, usr_{tr}, pwd_{tr}, k, \mathrm{enc}\left(X, k\right), en(M)\} \triangleright & \mathrm{enc}\left(Y, k\right) \\
\{K_{tr}, K_{sc}, usr_{sc}, pwd_{sc}, k, t, \mathrm{pair}\left(corr(T), n(T)\right), \mathrm{enc}\left(Y, k\right)\} \triangleright & en(corr(t))
\end{array}\right\}$$

Fig. 4. A constraint system (T, M, X, Y are variables) describing a possible distributed orchestration.

from the latter. In our example we can consider pwd_{sc} and pwd_{tr} as sensitive for the Mediator and Partner correspondingly. Later, in § 3.2 we will present a *non-disclosure condition* which is sufficient to ensure *(ii)*.

We can solve this problem if the number of *interactions* (i.e. invocations or send/receive pairs) is bounded. Suppose for simplicity that we can use every available service at most once, and that we allow one round of communication between the Mediator and the Partner (Mediator sends request and the Partner replies).

Again, as in previous example, we have to choose an interleaving of Client's and Partners' actions and invocations of available services (note that the number of all possible interleavings is finite).

Assume the selected interleaving is:

1. Client $\rightarrow$ Mediator
2. Mediator $\leftrightarrow$ SpellChecker
3. Mediator $\rightarrow$ Partner
4. Partner $\leftrightarrow$ Translator
5. Partner $\rightarrow$ Mediator
6. Mediator $\rightarrow$ Client

where $A \rightarrow B$ stands for "A sends and B receives", and $A \leftrightarrow B$ stands for "B is invoked by A". The corresponding constraint system is depicted in Fig. 4.

We can see that the obtained constraint system is not well-formed, and thus, the existing above mentioned solving methods (except [20]) are not applicable. The reason is that the property called knowledge monotonicity does not hold. This is due to the fact that there are multiple entities (partners) whose knowledge we must take into account, while in the case of non-distributed orchestration there exists the only entity (the Mediator) whose knowledge is expressed in LHS of the constraints.

Another remark: we cannot treat the constraint system in a modular manner, i.e. consider separately a subsystem of constraints for each partner, since the obtained constraint systems are not independent as they share variables. If we do not take into account this fact and solve them separately, then we will be probably unable to join the solutions, since they can be "incompatible", i.e. a solution for one subsystem implies one value for a variable, while a solution for another may imply another value for the same variable. Moreover, these subsystems are even not well-formed, since in general they will not satisfy the variable origination property.

One of the possible solutions, that can be automatically built (§ 3.3), of this constraint system is $\sigma = \{T \mapsto t, M \mapsto corr(t), X \mapsto corr(t), Y \mapsto en(corr(t))\}$. Mediator gets text t from the Client, sends it encrypted together with his login data to the SpellChecker, then from the reply he extracts the corrected version $corr(t)$ of the text, sends it to the Partner, who concatenates it together with his translator login/password and sends it encrypted to the Translator. Partner forwards the obtained response to the Mediator who returns it to the Client.

Note that neither $X\sigma$ nor $Y\sigma$ contains login information of the Mediator or the Partner.

3.2 Formal model

In this section we introduce a model for the distributed orchestration. Note that the standard (non-distributed) orchestration is the special case with the unique mediator. Informally, the distributed orchestration problem is stated as follows: given a community of the available services, a Client, a set of partners each with some initial knowledge and a set of communication channels between the partners, find a feasible communication scenario between partners, available services and the Client, such that the Client reaches its final state.

In this communication scenario the circulated messages have to be sent on a set of predefined channels and they have to follow some message patterns defined per each channel. Moreover some non-disclosure policies imposed on the communications are specified as a set of sensitive atomic data per each partner, which should not be extractable from messages sent to other partners.

To ensure the latter we will consider a stronger property: we don't allow any occurrence of sensitive data as a subterm of these messages. Indeed, as deduction rules in Table 2 do not produce new atoms, a partner is unable to extract any sensitive data from a message that does not contain it. In this way the partners are guaranteed not to directly reveal their confidential information to other ones, but this information still can be used to invoke the available services.

Note that in order to directly solve the problem with non-disclosure policies, one should consider more complex techniques that are able to cope with satisfiability of constraint systems that includes also a negation (term t should *not* be deducible from set of terms E). Unfortunately we are not aware of any such results.

Orchestration problem input We assume we are given:

- A set of available services $\mathbb{S} = \{S_1, \ldots, S_n\}$. Available service S_i represented by its name and a normal (for the sake of simplicity) strand, i.e. $S_i = \langle i, A_i \rangle$, where $A_i = ?r_1.!s_1. \ldots .?r_{e_i}.!s_{e_i}$.
- A client C. We can think of the client as a stand-alone available service $\langle 0, A_0 \rangle$, but A_0 is a strand which is not necessarily normal.
- A set of partners $\mathbb{P} = \{P_1, \ldots, P_k\}$ (P_1 is a Mediator) and for each partner P_i, a set of sensitive atoms N_i that he does not want to share with partners. Partner P_i is represented by its name i, his current knowledge K_i and a set of sensitive atomic values $N_i \subseteq \mathrm{Sub}\,(K_i)$, i.e. $P_i = \langle i, K_i, N_i \rangle$.
- A set of communication channels $\mathbb{C} = \{C_1, \ldots, C_u\}$ between partners. Communication channel C_i is a tuple $\langle i, j, p \rangle$, where i and j are names of partners P_i and P_j correspondingly and all messages sent from P_i to P_j must match pattern p. We assume that pattern p does not contain sensitive atoms as subterms, i.e. $\mathrm{Sub}\,(p) \cap N_i = \emptyset$.
- An upper bound on the number of interactions m.

We assume that the sets of variables used to describe each available service (and in the Client) are pairwise disjoint, i.e. $\mathrm{Vars}\,(S_i) \cap \mathrm{Vars}\,(S_j) = \emptyset$, if $i \neq j$ and for all i, $\mathrm{Vars}\,(S_i) \cap \mathrm{Vars}\,(C) = \emptyset$.

Execution model We define a *non-disclosure condition* (or *non-disclosure policy*) according to what we have already announced before: a sensitive atom of a partner never occurs as a subterm of a message emitted by him, but we will impose this policy *only* on messages emitted by one partner to another, while the communication with available services is free from this condition. We define a *non-disclosure condition* (or *non-disclosure policy*) $\mathcal{H}$ as a set of equations $\left\{\mathrm{Sub}\,(m_i) \cap N_{j_i} \stackrel{?}{=} \emptyset\right\}_{i=1,\ldots,l}$, where m_i is a term and N_{j_i} is a set of sensitive atoms. We will say that a ground substitution σ is a solution of (or satisfies) $\mathcal{H}$ iff for all $i = 1, \ldots, l$ an equality $\mathrm{Sub}\,(m_i\sigma) \cap N_{j_i} = \emptyset$ holds.

We present a configuration as a tuple $\langle \{S_1, S_2, \ldots, S_n\}, C, \{P_1, \ldots, P_k\}, \mathcal{S}, \mathcal{H} \rangle$, i.e. set of available services, client, set of partners, constraint system and non-disclosure condition to be satisfied. We define a set of transitions in Fig. 5 that allow to pass from one configuration to another.

Transition 4 expresses that Partner $P_i = \langle i, K_i, N_i \rangle$ can invoke available service $S_j = \langle j, A_j \rangle$, iff he is able to build a message (ground term) that is compatible with the expected pattern. The reply of S_j will become a part of the partner's knowledge. Similarly for the message exchange of the Mediator P_1

$$\frac{\left\langle \left\{ \left\langle j, ?r.\,!s.\,A'_j \right\rangle \right\} \cup S', C, \{\langle i, K_i, N_i\rangle\} \cup P', \mathcal{S}, \mathcal{H} \right\rangle}{\left\langle \left\{ \left\langle j, A'_j \right\rangle \right\} \cup S', C, \{\langle i, K_i \cup \{s\}, N_i\rangle\} \cup P', \mathcal{S} \cup \{K_i \rhd r\}, \mathcal{H} \right\rangle} \tag{4}$$

$$\frac{\langle S, \langle 0, !s.\,A'\rangle, \{\langle 1, K_1, N_1\rangle\} \cup P', \mathcal{S}, \mathcal{H}\rangle}{\langle S, \langle 0, A'\rangle, \{\langle 1, K_1 \cup \{s\}, N_1\rangle\} \cup P', \mathcal{S}, \mathcal{H}\rangle} \tag{5}$$

$$\frac{\langle S, \langle 0, ?r.\,A'\rangle, \{\langle 1, K_1, N_1\rangle\} \cup P', \mathcal{S}, \mathcal{H}\rangle}{\langle S, \langle 0, A'\rangle, \{\langle 1, K_1, N_1\rangle\} \cup P', \mathcal{S} \cup \{K_1 \rhd r\}, \mathcal{H}\rangle} \tag{6}$$

$$\frac{\langle S, C, \{\langle i, K_i, N_i\rangle, \langle j, K_j, N_j\rangle\} \cup P', \mathcal{S}, \mathcal{H}\rangle \qquad [\text{if } \langle i, j, p\rangle \in \mathbb{C};\ q = \text{refresh}(p)]}{\langle S, C, \{\langle i, K_i, N_i\rangle, \langle j, K_j \cup \{q\}, N_j\rangle\} \cup P', \mathcal{S} \cup \{K_i \rhd q\}, \mathcal{H} \cup \{\text{Sub}(q) \cap N_i \overset{?}{=} \emptyset\}\rangle} \tag{7}$$

where refresh(t) is a term equal to t where all variables are replaced with fresh ones.

Fig. 5. Transition system

and the Client C, except that the Client can initiate a sending (5,6). A partner $P_i = \langle i, K_i, N_i\rangle$ can send a message to a partner P_j, iff there exists a channel $C_{ij} = \langle i, j, p\rangle \in \mathbb{C}$ between them such that partner P_i can build a message compatible with pattern p and this message will not contain sensitive data from N_i as a subterm (7). Note that in this setting services cannot be reused second time, but we are free to add several instances for services to problem input.

A sequence of length l of configurations starting with initial one $\langle \mathbb{S}, C, \mathbb{P}, \emptyset, \emptyset\rangle$ and obtained by applying transitions from Fig. 5 is called *symbolic execution* SE of length l.

An *execution* E is a pair $\langle SE, \sigma\rangle$ of a symbolic execution SE and ground substitution σ, such that for the last configuration $\left\langle \mathbb{S}^l, C^l, \mathbb{P}^l, \mathcal{S}^l, \mathcal{H}^l \right\rangle$ of SE the $\mathcal{S}^l$ and $\mathcal{H}^l$ are satisfied by σ. We can see that an execution E defines message flow, and thus the sequence of actions performed by every Partner.

Distributed orchestration problem statement Given a problem input as described above is there an execution $E = \langle SE, \sigma\rangle$ of length $l \le m$ such that at the end the sequence of actions of the Client is empty, i.e. the last configuration of SE is $\left\langle \mathbb{S}^l, \langle 0, \emptyset\rangle, \mathbb{P}^l, \mathcal{S}^l, \mathcal{H}^l \right\rangle$?

3.3 Solving the distributed orchestration problem

We reduce the distributed orchestration problem to the satisfiability of a deducibility constraint system under non-disclosure of sensitive data condition and then discuss a decision procedure under the hypothesis of bounded number of interactions.

Since one can build finitely many different symbolic executions for a fixed problem input, we can guess a symbolic execution with its final configuration $\left\langle \mathbb{S}^l, C^l, \mathbb{P}^l, \mathcal{S}^l, \mathcal{H}^l \right\rangle$ where the Client has no more actions to perform (i.e. $C^l = \langle 0, \emptyset\rangle$). And then, building the desired execution is equivalent to finding such a ground substitution σ that satisfies both $\mathcal{S}^l$ and $\mathcal{H}^l$.

We refer to [3] for the technique for solving constraint systems within Dolev-Yao deduction system. Under non-restrictive assumption that in the problem input there exists at least one atomic value which is not sensitive to any of partners, and the assumptions stated in § 3.2 hold, we can easily adapt the mentioned technique in such way that the following theorem holds:

Theorem 1. *Satisfiability of deducibility constraint system within Dolev-Yao deduction system under non-disclosure condition is in NP.*

And thus,

Corollary 1 *The problem of distributed orchestration is decidable and in NP.*

Note also that having a desired execution $E = \langle SE, \sigma \rangle$, and thus a sequence of actions performed by the Partners as well as their initial knowledges, we can extract a *prudent implementation* of the Partners as services (see details in [15]).

4 Implementation

We report only an implementation for standard (non-distributed) orchestration introduced in Section 2. As mentioned above, being a special case of the distributed orchestration, standard orchestration can be handled using simpler constraint solving procedure since in that case one has only to handle well-formed constraints. To this end we use `CL-AtSe` tool [30]. We are not aware of tools that can be reused to implement the distributed orchestration.

4.1 Input and Output Language

The available services, the Client and the resulting Mediator are described in *ASLan* language [1], a formal language for specifying trust and security properties of services, their associated policies, and their composition into service architectures. A service's behavior in ASLan is defined as a set of transitions, its initial state and possibly a finite set of Horn clauses typically used to define an authorization logic; and messages to be exchanged (with applied security policies) as first order terms.

For example, a TimeStamper service can be represented as the following transition:

```
step step_0(TS,Dummy_Time,Dummy_M,Time,M):=
  state_timestamper(TS,1,Dummy_Time,Dummy_M).
  iknows(M)
  =[exists Time]=>
  state_timestamper(TS,1,Time,M).
  iknows(pair(M, pair(Time, crypt(inv(kts), pair(apply(md5,
      M), Time)))))
```

where `state_timestamper` represents a *state* of `TimeStamper` service. The message exchange is modeled by `iknows` *facts*. The transition means that if the `TimeStamper` being in state `state_timestamper(TS,1,Dummy_Time,Dummy_M)` receives message `M`, he will generate a new Time value (`=[exists Time]=>`), pass to state `state_timestamper(TS,1,Time,M)` and reply with message `pair(M, pair(Time, crypt(inv(kts), pair(apply(md5, M), Time))))`. In other words, this service receives a message, stores it (encoded in his state) and reply with a time stamp of the received message. Note that ASLan supports functional symbols equivalent to ones from Table 1.

The global state of the transition system is given by a set of facts that are true. A transition can fire, if it has an instance (where all the variables are replaced with ground terms) such that all the facts of its left-hand side (LHS) are true. As a result, the facts of the LHS are removed from the current state and the facts of the RHS are added. The only exception is the `iknows` facts which are considered as persistent.

Since we are looking for an orchestration, all the messages emitted by the services or the Client or received by them should come to/from the Mediator. Thus, every `iknows` fact (as it models the communication) once produced by the transition comes to the knowledge of the Mediator. And vice versa, `iknows` facts that are consumed by the services or the Client should be produced by the Mediator.

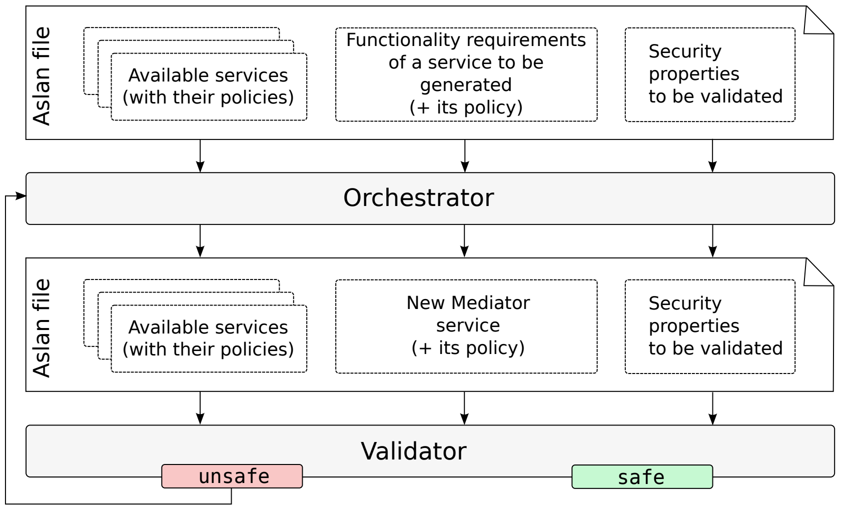
4.2 AVANTSSAR Platform

The Orchestrator is part of the AVANTSSAR Platform [1], an automated toolset for validating trust and security aspects of service-oriented architectures (see Fig. 6). The overall objective of the platform is to generate from an orchestration problem a solution that meets some given security requirements (e.g., secrecy, authentication).

The AVANTSSAR platform also offers the possibility to check whether an orchestrated service (e.g. generated by the Orchestrator tool) is vulnerable to an active Dolev-Yao intruder. For that the modeler inputs some security properties, e.g. some message confidentiality requirement, before starting the orchestration generation. Some automatic validation tools (like [30] and others, see [1]) will check whether the property is satisfied by the generated orchestration. The Orchestrator solves the orchestration problem, and then the solution (with the properties to be validated) are transferred to the AVANTSSAR Validator.

If the specification meets the validation goals then the orchestration solution is considered as safe *w.r.t.* the user's requirements, otherwise a verification report including the violation proof is returned. In the latter case the Orchestrator is able to backtrack and try an alternative solution. This is illustrated in Fig. 6 by the returning-arrow from the Validator to the Orchestrator.

To summarize the platform performs as follows: (*i*) it generates an orchestration (a Mediator); (*ii*) it verifies security properties on it; (*iii*) if the orchestration is vulnerable, it generates a new Mediator and jumps to (*ii*).



Aslan file
Available services (with their policies)
Functionality requirements of a service to be generated (+ its policy)
Security properties to be validated
Orchestrator
Aslan file
Available services (with their policies)
New Mediator service (+ its policy)
Security properties to be validated
Validator
unsafe
safe

Fig. 6. The AVANTSSAR Validation platform

Table 3. AVANTSSAR Orchestrator benchmark

Input problem				Mediator generation	
Case study	Number of available services	Number of transitions	Number of Horn Clauses	Number of generated transitions	Running time
CRP	4	25	17	17	4.1 sec.
DCS	3	18	0	22	4.6 sec.
PB	2	20	2	4	1.7 sec.

5 Conclusion

The automatic composition of Web services is a challenging task on the understanding that all the aspects of Web services have to be taken into account. We presented a novel approach for automatic distributed orchestration of Web services under security constraints generalizing the non-distributed case as well as tackling the Web services choreography.

Whereas there are a lot of theoretical works on the automatic composition of Web services, only a few are ended with the implementation. We described a tool that allows one to automatically orchestrate Web services taking into account in the same time their workflow, messaging and security.

We have successfully tested the tool on several industrial case studies (see Table 3), like *Digital Contract Signing* (DCS) and *Public Bidding* (PB) which are originated from commercial products of the OpenTrust company and *Car Registration Process* (CRP), a case study proposed by Siemens AG.

The Orchestrator is deployed and available at http://avantssar.loria.fr/OrchestratorWI/. It was implemented in OCaml and Java and its source contains of more than 20'000 lines of code.

References

1. Automated Validation of Trust and Security of Service-Oriented Architectures, AVANTSSAR project. http://www.avantssar.eu.
2. Network of Excellence on Engineering Secure Future Internet Software Services and Systems, NESSoS project. http://www.nessos-project.eu.
3. T. Avanesov, Y. Chevalier, M. Rusinowitch, and M. Turuani. Satisfiability of General Intruder Constraints with and without a Set Constructor. Research Report RR-7276, INRIA, 05 2010. http://hal.inria.fr/inria-00480632/en/.
4. Luciano Baresi, Andrea Maurino, and Stefano Modafferi. Towards distributed bpel orchestrations. *ECEASST*, 3, 2006.
5. D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic Composition of Transition-based semantic Web Services with Messaging. In *Proc. 31st Int. Conf. Very Large Data Bases, VLDB 2005*, pages 613–624, 2005.
6. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of e-Services that export their Behavior. In *Proc. 1st Int. Conf. on Service Oriented Computing, ICSOC 2003*, volume 2910, 2003.
7. K. Bhargavan, R. Corin, P.M. Deniélou, C. Fournet, and J.J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 124–140. IEEE, 2009.
8. A. Bucchiarone and S. Gnesi. A survey on services composition languages and models. In *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 51–63, 2006.
9. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.
10. S. Bursuc, S. Delaune, and H. Comon-Lundh. Deducibility constraints. In *ASIAN'09*, volume 5913 of *LNCS*, pages 24–38, Seoul, Korea, Dec. 2009. Springer.
11. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Massimo Mecella, and Fabio Patrizi. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.
12. J. Camara, J.A. Martin, G. Salaun, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. Itaca: An integrated toolbox for the automatic composition and adaptation of web services. *ICSE '09*, 0:627–630, 2009.
13. Y. Chevalier, R. Küsters, M. Rusinowitch, and M. Turuani. An np decision procedure for protocol insecurity with xor. *Theo. Comp. Sci.*, 338(1-3):247–274, 2005.
14. Y. Chevalier, M.A. Mekki, and M. Rusinowitch. Automatic composition of services with security policies. In *Proceedings of the 2008 IEEE Congress on Services - Part I*, SERVICES '08, pages 529–537, Washington, DC, USA, 2008. IEEE.
15. Y. Chevalier and M. Rusinowitch. Compiling and securing cryptographic protocols. *Inf. Process. Lett.*, 110(3):116–122, 2010.
16. D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.
17. F.J.T. Fabrega, J.C. Herzog, and J.D. Guttman. Strand spaces: why is a security protocol correct? In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 160–171, May 1998.
18. Joshua D. Guttman, F. Javier Thayer, Jay A. Carlson, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Trust management in strand spaces: A rely-guarantee method. In *In Proc. of the European Symposium on Programming (ESOP '04), LNCS*, pages 325–339. Springer-Verlag, 2004.

19. J.A. Martín, Martinelli. F., and E. Pimentel. Synthesis of secure adaptors. In *JLAP (to appear)*, 2011.
20. Laurent Mazaré. *Computational Soundness of Symbolic Models for Cryptographic Protocols*. PhD thesis, Institut National Polytechnique de Grenoble, October 2006.
21. OASIS. Security Assertion Markup Language (SAML) v2.0. Available at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`, April 2005.
22. F. Patrizi. An introduction to simulation-based techniques for automated service composition. In *YR-SOC 2009, Pisa, Italy*, volume 2 of *EPTCS*, pages 37–49, June 2009.
23. G. Pedraza and J. Estublier. Distributed orchestration versus choreography: The focas approach. In *ICSP '09*, pages 75–86, Berlin, Heidelberg, 2009. Springer-Verlag.
24. C. Peltz. Web Services Orchestration, 2003. HP white paper.
25. C. Peltz. Web services orchestration and choreography. *Computer*, 36:46–52, 2003.
26. S. Quinton, I. Ben-Hafaiedh, and S. Graf. From orchestration to choreography: Memoryless and distributed orchestrators. In *Proc. of FLACOS'09*, 2009.
27. M. ter Beek, A. Bucchiarone, and S. Gnesi. Web service composition approaches: From industrial standards to formal methods. *Internet and Web Applications and Services, 2007. ICIW '07. Second International Conference on*, page 15, 2007.
28. The AVISPA Project. `www.avispa-project.org/`.
29. M. Trainotti, M. Pistore, G. Calabrese, Zacco G., G. Lucchese, F. Barbon, P. Bertoli, and P. Traverso. ASTRO: Supporting Composition and Execution of Web Services. In *ICSOC 2005*, LNCS 3826, pages 495–501. Springer Verlag, 2005. URL of the ASTRO project: `http://sra.itc.it/projects/astro/`.
30. M. Turuani. The CL-Atse Protocol Analyser. In *Proceedings of RTA'06*, LNCS 4098, pages 277–286, 2006.

A Tool architecture

The Orchestratortakes 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 Mediator[3]. The Orchestrator output is a file that contains the ASLan specification of the Mediator and the community of available Web services.

Figure 7 depicts the architecture of the Orchestrator. Sub-tools `Client2Client` and `Goal2Client` are used to preprocess the input for `CL-AtSe` (as it was originally created to tackle protocol insecurity problem). `Client2Client` assumes the presence of the Client specification, while `Goal2Client` assumes that a partial specification of the Mediator is given from which it produces a generic Client specification; it also produces some additional information (like a table of variables renaming that was done during generation of the Client) used later by the `Trace2ASLan`.

Then `CL-AtSe` is called. It is a key component of the Orchestrator. Given an ASLan specification, it generates a trace — the sequence of communication

[3] The input also contains validation goals which do not concern the Orchestrator. See § 4.2 for details.

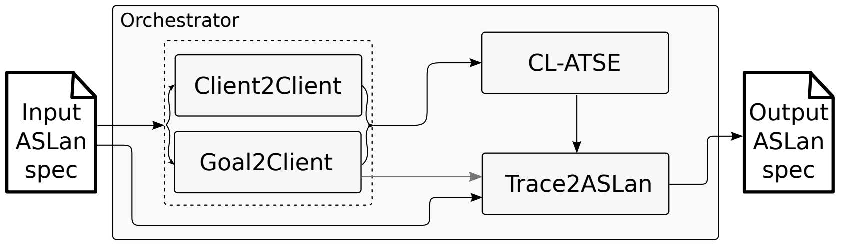


Fig. 7. Tool architecture

events (send/receive) between a Mediator (played by the intruder) and the available services (that can be used in the composition) — such that a final state of the Client is reached. A detailed description of the tool can be found in [30] and [1].

Finally, `Trace2ASLan` takes as input *(i)* the trace returned by `CL-AtSe`, describing the Mediator, *(ii)* the initial ASLan specification, and possibly *(iii)* auxiliary files produced by `Goal2Client`. It returns the initial ASLan specification augmented with the ASLan specification of the Mediator (and possibly a putative Client). The former is extracted from the trace which is a sequence of communication events in the form "A sends to B message m" and corresponds to a *prudent implementation* of the Mediator which informally means that it checks its input as thoroughly as possible (for a formal definition, see [15]).

B Digital Contract Signing case study

We describe in the following an experiment we had with the Orchestrator tool on the Digital Contract Signing case study (DCS). DCS describes two parties that have secure access to a trusted third party Web site, a Business Portal (BP), in order to digitally sign a contract. First, BP generates an electronic document corresponding to the terms of agreement between the two parties. Then, the first party accesses BP using a Web browser, views the contract and signs it using a digital certificate. BP verifies the generated signature and stores it. The second party, in turn, connects to the BP Web site, checks the status of the existing signature and then co-signs the contract after viewing it. Once the signatures have been completely verified by the business portal, the signers are notified. Then, the contract is archived for long-term conservation. The BP's internal system is Web service enabled. It delegates the processing of proof elements (signatures, signed documents, timestamps) to a Security Server (SS) using SOAP messages.

Three available trusted services are in the disposal of SS: a Time Stamper (TS), a Public Key Infrastructure (PKI), that returns information about the validity of a given certificate and an Archiver (ARC), an external storage facility.

The orchestration problem here is to generate a Mediator that emulates SS: satisfy BP's requests while relying on the community of available services

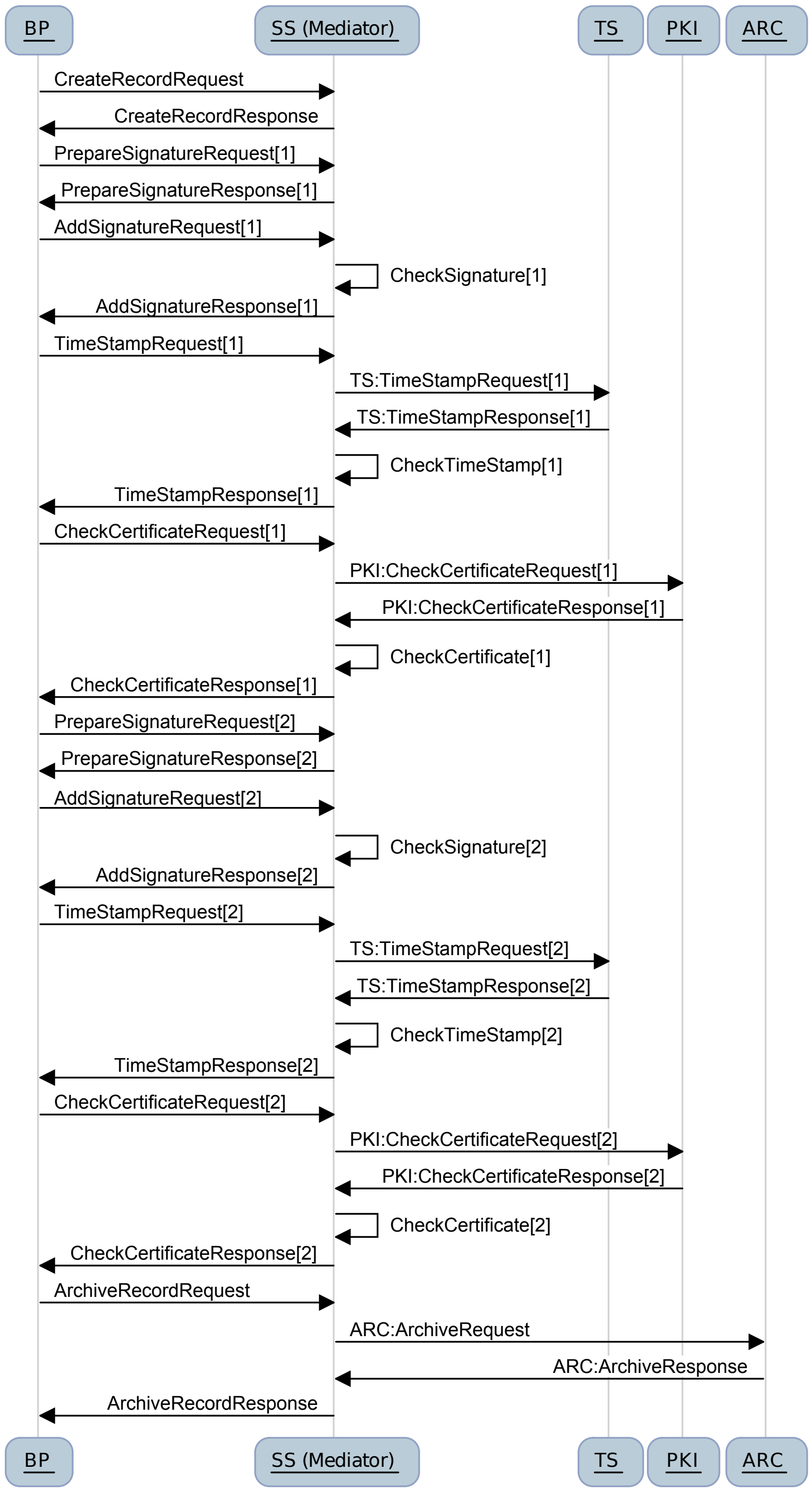


Fig. 8. A Mediator service for DCS orchestration problem

(namely TS, PKI and ARC). Figure 8 represents the solution generated by the tool.

Indeed, the generated Mediator service (SS) expects an initialization message to start the digital signature procedure and acknowledge the reception. Then BP invokes SS to get the signature policy for the first signer. Then BP transfers the contract signed by the first signer to SS which should check the signature and produce an assertion about its validity that BP expects back as a response. Then SS is asked to time stamp the signature and provide the corresponding assertion. To obtain the time stamp, SS must invoke the trusted TS. After this SS is asked to check whether the certificate used by the signer was not revoked, and if it was not, to return the corresponding assertion. Indeed, SS contacts PKI and tries to derive the needed assertion from its response. If SS succeeds the first round of the signature procedure is successfully ended. Similar steps are needed to collect the second signer's data before SS is asked by BP to archive the documents and proofs collected during the signature procedure. Therefore SS invokes ARC with the appropriate message, the latter acknowledges the reception and finally SS calls back BP to signal the successful end of the signature procedure.

Assertions produced by SS are claims made by some issuer and stating some property for the parameters they transport. An equivalent in the Web Service standards stack is SAML [21] assertions, which we simply model using first-order terms. The presence of an assertion in some received message by BP represent an additional constraint to the orchestration problem since SS will have to provide it. In this case-study one of the assumptions was that BP trusts SS as issuer for the assertions it required for example about the validity of one signer's certificate. To produce this assertions SS have to contact an internal service: the Assertions Provider (AP) which permits to provide the good assertion only if a positive answer about the validity of the certificate is given by a trusted third-party (here PKI). AP plays a role similar to the *trust engine* in *rely-guarantee* method introduced in [18]. Note that the calls to AP are abstracted in Fig. 8 by the returning arrows to SS.

We underline here the expressiveness of assertions for the considered orchestration problem, since they can describe for example the need to use only certain schema for the time stamps, or only PKI's offering the *Online Certificate Status Protocol (OCSP)* versus those using the classical *Certificate Revocation List (CRL)*. This can be easily done by tuning the AP service behavior to match the wanted expectancies.

C Short proof recall of Theorem 1

Proof (Short recall). Suppose a constraint system $\mathcal{S}$ is satisfiable with substitution σ that also satisfies non-disclosure condition $\mathcal{H}$. By applying some transformation on σ we can obtain another solution δ of $\mathcal{S}$ such that for any variable x, $x\delta$ is bounded by $P(\text{size}(\mathcal{S}))$, where P is a polynomial. And since the check $t \in \text{Der}(E)$ for ground t and set of ground terms E is polynomial, we obtain an NP procedure for the satisfiability of constraint systems.

We also note that the used transformation does not change atoms of a substitution if they appear in $\mathcal{S}$, but only ones that are not by replacing them with some fixed atom α. And as we assumed an existence of an atom a_0 which is not sensitive by any of Partners, we can put $\alpha = a_0$ and thus, δ will also satisfy $\mathcal{H}$ (under assumption we gave in § 3.2 concerning communication channels).

D Some message sequence charts

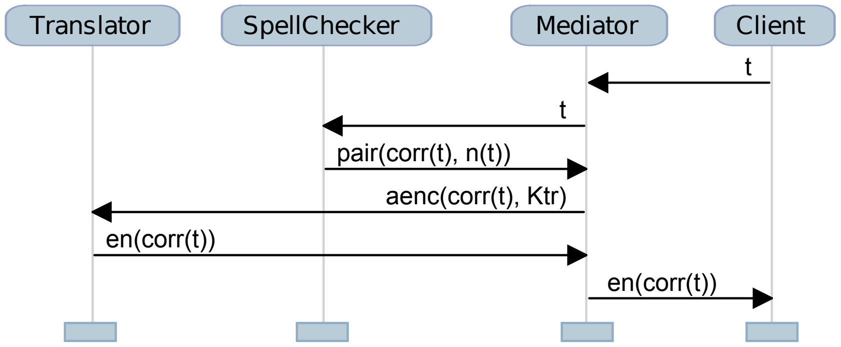


Fig. 9. Solution for the orchestration example (§ 2.2)

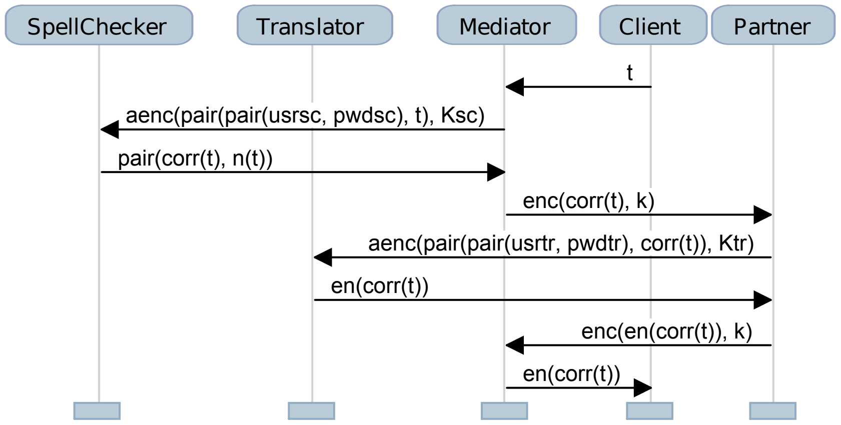


Fig. 10. Solution for the distributed orchestration example (§ 3.1)