Noname manuscript No.
(will be inserted by the editor)

A formal framework for secure and complying services

Davide Basile · Pierpaolo Degano ·
Gian-Luigi Ferrari

the date of receipt and acceptance should be inserted later

Abstract Internet is offering a variety of services, that are assembled to accomplish requests made by clients. While serving a request, security of the communications and of the data exchanged among services is crucial. Since communications occur along specific channels, it is equally important to guarantee that the interactions between a client and a server never get blocked because either cannot access a selected channel. We address here both these problems, from a formal point of view. A static analysis is presented, guaranteeing that a composition of a client and of possibly nested services respects both security policies for access control, and compliance between clients and servers.

Keywords formal methods, service contracts, security, compliance, QoS

1 Introduction

The opportunities of exploiting distributed services are becoming an imperative for all organizations and, at the same time, new programming techniques are transforming the ways distributed software architectures are designed and implemented. Distributed applications nowadays are built by assembling together computational facilities and resources offered by (possibly) untrusted providers. E.g. in Service-Oriented Computing [19], the basic building blocks are independent software components called *services*, equipped with suitable interfaces describing (roughly) the offered computational facilities; standard communication protocols (e.g. SOAP over HTTP) take care of the interactions among parties. Another illustrative example of effective distributed services exploitation is Cloud computing [3].

Software architectures for distributed services require several operational tools to be in place and effective. In these computing environments, managing security issues is rather complex, since security controls are needed to handle access to

This work has been partially supported by the MIUR PRIN project *Security Horizons*

Davide Basile, Pierpaolo Degano Gian-Luigi Ferrari
Dipartimento di Informatica, Università di Pisa, Italy
E-mail: basile,degano,giangi@di.unipi.it

services and resources, as well as to create suitable and reliable services on demand. E.g. identity management and authorization are vital to set-up and deploy services on-the-fly. Moreover, when services are made available through third parties is crucial to ensure that clients and services interact correctly, which in turn implies verifying the correctness of their behavioural obligations. *Service Contracts* are the standard mechanisms to describe the external observable behaviour of a service, as well as the responsibility for security. Service contracts can be used for guaranteeing that all the services are capable of successfully terminating their tasks (progress property) without rising any security exceptions.

The management of service contracts is typically tackled by service providers through the notion of *Service-Level Agreement* (SLA), which clients must accept before using the service. SLAs are documents that specify the level(s) of service being sold in plain language terms. Recently, so called *contract-oriented design* has been introduced as a suitable methodology where the interaction between clients and services are specified and regulated by formal entities, named *contracts*. Their formal nature permits to unambiguously represent the obligations, helps developing mechanical systems for service management, and also supports the automatic analysis and verification of the contracts themselves. In the literature several proposals addressed and investigated contracts, exploiting a variety of techniques [13, 15,17,16,11,4,1]. Security issues have been recently amalgamated within contract based design by [9], that develops a calculus where behavioural contracts may be violated by dishonest participants after they have been agreed upon.

We outline here a formal theory of contracts that supports verification of *service compliance* and of *security policies* enforcing access control over resources. Services are compliant when their interactive behaviour eventually progresses, i.e. all the service invocations are guaranteed to be eventually served. Services are secure when data are exchanged and accessed according to specific rules, called *policies*.

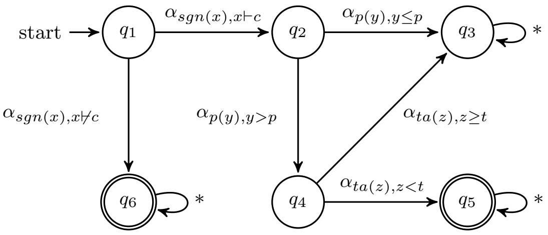
Our starting point is the language-based methodology supporting static analysis of security policies developed in [6]. Its main ingredients are: local policies, call-by-contract invocation, type and effect systems, model checking and secure orchestration. A program is type-checked and a safe over-approximation of its behaviour is extracted, recording both the actions that may affect security and the policies that must be enforced at run-time. Then, the approximation is model checked against the active security policies, that are formally expressed as safety policies. If the test is passed, then the program will never raise security exceptions at run-time. A *plan* orchestrates the execution of a service-based application, by associating the sequence of run-time service requests with a corresponding sequence of selected services. A main result shows how to construct a plan that guarantees that no executions will abort because of some action attempting to violate security. Note that it is not sufficient checking each service in isolation because the composition of multiple services may introduce tricky security bugs. This methodology has been extended in [18] to also deal with quantitative aspects of service contracts, typically the rates at which the different activities are performed.

Here we extend this approach to check security and service compliance at the same time. Our first contribution is to extend the abstraction of service behaviour, called *history expressions*, with suitable communication facilities to model the interactive behaviour of services, including possibly nested service sessions in a multiparty fashion. In particular, we extend history expressions to include communications along channels, and internal/external choice for combining the notions of security of resource accesses and progress of interactions. Our second contribution is sharpening the verification phase. We prove that service compliance is a safety property: when it holds, all the involved parties are able to successfully complete their interactions without getting stuck. Reducing service compliance to a safety property makes it efficiently model-checkable. Finally, we extract from a history expression all the *viable* plans for serving a request, i.e. those orchestrations that successfully drive secure and compliant executions. Adopting a valid plan guarantees that the involved services never go wrong at run-time, in that they are capable of successfully accomplishing their tasks, raising no security exceptions. Thus, no run-time monitor is needed, to control the execution of the network of services.

The paper is organised as follows. The next section intuitively presents our formal machineries, the problems we address, and our goals through an illustrative example. Compliance, its reduction to a safety property and the technicalities needed to model-check it are in Section 3. In Section 4, we summarise our results and future work. Lack of space prevents us to present in details the technical treatment that is at the web page: `www.di.unipi.it/user/basile/papers/pact2013full.pdf`.

2 Motivating Example

To illustrate our approach and help intuition, we consider a simple cloud-based scenario. We take into account *federated cloud services* [14] where a broker is responsible for collecting the clients' requests and for sending the information to the multiple cloud providers. The broker has also granted the rights to negotiate the service contracts with the providers on the behalf of the clients: and to distribute and coordinate clients' requests across the multiple cloud providers. Finally, formalising and guaranteeing security of federated cloud services is a major challenge [12]. When a client submits its service requests to the broker, it also specifies the constraints on the required quality of service. In our approach, this negotiation is managed by issuing the policy $\varphi_{(c,p,t)}$. Roughly, policies are regular properties over the execution progress of service requests (*execution histories*), which are specified through the so-called usage automata [7], a sort of parametric finite state automata. As an example the policy φ displayed below will be used in our running example to determine the possible bad behaviour, discuss later on.



The policy parameters are the configuration profile c, the pricing p and the workload threshold t. Since in the cloud federation each provider comprises multiple computing services, the information stored in c is used to identify the computing facilities according to the need of the clients. The configuration profile can also contain information about the required security constraints to model the fact that cloud architectures have well-defined security policies and enforcement mechanisms in place. E.g., the profile could include the requirement to provide a level of isolation over the virtualization infrastructure. The pricing information is used to decide how service requests are charged. E.g., pricing can be based on the submission time, it could be fixed or depending on the availability of certain resources. The last parameter t specifies the minimum amount of work expected to be done. E.g., in a data center this value could be the amount of data processing performed in a given time. The parameters of the service contract are crucial to identify and supply the actual demand of computing resources on the cloud.

When a cloud provider accepts the service requests, it signs the contracts, i.e. it supplies the information x about the profile of the available resources issuing the *event* $\alpha_{sgn(x)}$. The cloud provider also publishes its pricing and workload information by issuing the events $\alpha_{p(y)}, \alpha_{t(z)}$. If the profile of the available resources does not match the given client profile (formally expressed by the guard $x \not\vdash c$) the policy is violated and the final state q_6 is reached. Note that the forbidden traces belong then to the language *accepted* by the automaton, as prescribed by the so called "default-accept" approach. A violation of the policy also occurs if the pricing of the cloud provider does not fulfill client's requirements and the provided workload is lower then t. In this case the policy is violated and the final offending state q_5 is reached. Finally, note that the policy allows to have a pricing different from the given one whenever a higher workload performance is ensured.

We abstractly and formally specify this scenario in a suitable process calculus, that deeply differs from standard ones because policies are first class citizens of the calculus. Consider the following setting with two clients C_1, C_2, a cloud broker B and a cloud federation including computational service providers S_1, S_2, S_3, S_4.

$$\begin{aligned}
&C_1 = \texttt{open}_{1,\varphi(\{c1\},45,100)}\overline{Req}(Ack.\overline{Pay} + NoAv)\texttt{close}_{1,\varphi(\{c1\},45,100)} \\
&C_2 = \texttt{open}_{2,\varphi(\{c2\},40,70)})\overline{Req}(Ack.\overline{Pay} + NoAv)\texttt{close}_{2,\varphi(\{c2\},40,70)}) \\
&B = Req.\texttt{open}_{3,\emptyset}\overline{IdC}.(AgOk + UnA)\texttt{close}_{3,\emptyset}(\overline{Ack}.Pay \oplus \overline{NoAv}) \\
&S_1 = \alpha_{sgn(1)}.\alpha_{p(45)}.\alpha_{t(80)}.IdC(\overline{AgOk} \oplus \overline{UnA}) && S1 \notin c1, c2 \\
&S_2 = \alpha_{sgn(2)}.\alpha_{p(70)}.\alpha_{t(100)}.IdC(\overline{AgOk} \oplus \overline{UnA} \oplus \overline{Del}) \\
&S_3 = \alpha_{sgn(3)}.\alpha_{p(90)}.\alpha_{t(100)}.IdC(\overline{AgOk} \oplus \overline{UnA}) && S3 \notin c2 \\
&S_4 = \alpha_{sgn(4)}.\alpha_{p(50)}.\alpha_{t(90)}.IdC(\overline{AgOk} \oplus \overline{UnA})
\end{aligned}$$

The two clients only differ in the way they instantiate their policies. A client opens a session and sends his request to the broker, who must respect the policy φ. Sending the request is modelled by the action $\overline{Req}$, while receiving it is done through Req, the complementary action (for brevity and simplicity we omit data). The client is then willing to receive the confirmation of the policy agreement and to settle the payment ($Ack.\overline{Pay}$). The client is also ready to receive a negative message in the case where no resourses are available ($NoAv$). When either message is received, the session with the broker is closed. As discussed above, the broker receives the service request Req and then opens a session with the service providers. Here for simplicity we only model the interactions with the providers and not the actual deployment of the service. The broker sends the client Id and all the related data by issuing the event $\overline{IdC}$, and then waits for either the agreement or for the negative messages with ($AgOk + UnA$). Then the session is closed, and the response message is forwarded to the client. The cloud providers perform the events of signing and publishing the pricing and the workload; and they then interact with the broker. Here we assume a fixed pricing strategy and a basic description for the workload. Note that all services, except for S_2, have the *internal* choice $\overline{AgOk} \oplus \overline{UnA}$. This abstractly represents that the cloud providers can decide on their own which message to send depending on their state of affairs. Being purely non-deterministic, in our computational model the internal choice behaves differently than the external choice, e.g. $AgOk + UnA$, that is instead driven by the message received. Since the broker B is ready to receive each sent message, we say that the mentioned providers are *compliant* with B. Instead, the provider S_2 is *not compliant* with B. Indeed, the broker can also send the message $\overline{Del}$ (meaning that there will be available services later) but the broker cannot handle it, and therefore the interaction gets stuck. As far as security is concerned, assuming that S_1 does not match the configuration profile $c1$, then it turns out that the providers S_1 and S_4 violate the policy settled by C_1, since S_1 has not the required profile and S_4 does not respect the required workload. Finally, note also that the services S_1, S_3 do not satisfy the configuration parameters of C_2. Consider the following fragment of computation, i.e. a sequence of configurations χ and of transitions $\chi \xrightarrow{\gamma} \chi'$, where γ records either an event relevant to security or progress, or a communication made of two complementary actions (disregard for a while the indexes $\boldsymbol{\pi}, R$ of the arrows). A configuration is made of tuples $\eta, \ell : S$, put in parallel (through $\|$), where η is a sequence of events, and ℓ is the location of the service/client S.

$$
\begin{array}{r}
\varepsilon, \ell_{c1} : C_1 \| \varepsilon, \ell_{c2} : C_2 \xrightarrow{op_{1,\varphi_1}}_{\boldsymbol{\pi},R} \\
\llparenthesis_{\varphi_1}, [\ell_{c1} : \overline{Req}.(Ack.\ldots)\mathtt{close}_{1,\varphi_1}, \ell_{br} : Br] \| \varepsilon, \ell_{c2} : C_2 \xrightarrow{\tau}_{\boldsymbol{\pi},R} \\
\llparenthesis_{\varphi_1}, [\ell_{c1} : (Ack.\ldots)\mathtt{close}_{1,\varphi_1}, \ell_{br} : \mathtt{open}_{3,\emptyset}\overline{IdC}\ldots] \| \varepsilon, \ell_{c2} : C_2 \xrightarrow{op_{3,\emptyset}}_{\boldsymbol{\pi},R} \\
\overbrace{\llparenthesis_{\varphi_1}, [\ell_{c1} \ldots, [\ell_{br} : \overline{IdC}\ldots, \ell_{s3} : \alpha_{sgn(3)} \cdots]]}^{P} \| \varepsilon, \ell_{c2} : C_2 \xrightarrow{op_{2,\varphi_2}}_{\boldsymbol{\pi},R} \\
P \| \overbrace{\llparenthesis_{\varphi_2}, [\ell_{c2} : \overline{Req}\ldots\mathtt{close}_{2,\varphi_2}, \ell_{br} : Br]}^{Q} \xrightarrow{\alpha_{sgn(3)}}_{\pi,R} \xrightarrow{\alpha_{p(90)}}_{\pi,R} \xrightarrow{\alpha_{ta(100)}}_{\boldsymbol{\pi},R} \\
\overbrace{\llparenthesis_{\varphi_1}\alpha_{sgn(3)}\alpha_{p(90)}\alpha_{ta(100)}}^{\eta}, [\ell_{c1} : \ldots, [\ell_{br} : \overline{IdC}\ldots, \ell_{s3} : Idc\ldots]] \| Q \xrightarrow{\tau}_{\pi,R} \xrightarrow{\tau}_{\boldsymbol{\pi},R} \\
\eta, [\ell_{c1} : \ldots, [\ell_{br} : \mathtt{close}_{3,\emptyset}\ldots, \ell_{s3} : \varepsilon]] \| Q \xrightarrow{cl_{3,\emptyset}}_{\boldsymbol{\pi},R} \\
\eta, [\ell_{c1} : (Ack.\overline{Pay} + NoAv)\mathtt{close}_{1,\varphi_1}, \ell_{br} : (\overline{Ack}.Pay \oplus \overline{NoAv})] \| Q \xrightarrow{\tau}_{\boldsymbol{\pi},R} \\
\eta, [\ell_{c1} : \mathtt{close}_{1,\varphi_1}, \ell_{br} : \varepsilon] \| Q \xrightarrow{cl_{1,\varphi_1}}_{\boldsymbol{\pi},R} \eta\rrparenthesis_{\varphi_1}, \ell_{c1} : \varepsilon \| \llparenthesis_{\varphi_2}, [\ell_{c2} : \overline{Req}, \ldots, \ell_{br} : B] \xrightarrow{\tau}_{\boldsymbol{\pi},R} \ \ldots
\end{array}
$$

Above, the starting configuration has the two clients, one at location ℓ_{c1}, the other at ℓ_{c2}. Both performed no actions, so their execution history is empty (ε). The first step opens ($op1$) session 1 between C_1 and B and registers in the history that the whole session, in particular B, is subject to the policy φ, duly instantiated (call it $\varphi_1 = \varphi_{(\{c1\},45,100)}$). Step 2 shows that the request of the client C_1 has been accepted by the broker, via a communication. Now a nested session is opened between B and S_3, step 3, and no policy is imposed over the called service S_3. Concurrently, C_2 can perform its service request in step 4, that registers policy $\varphi_2 = \varphi_{(\{c_2\},40,70)}$ as active. Note that the broker can replicate its code at will.

The two parallel sessions can evolve concurrently. For simplicity, we proceed with service S_3, that signs, shows the pricing and its workload threshold (all displayed in the same line). The broker is ready to send the client's data to S_3, and to receive back an answer, say "no services are available" (S_3 is now ε, because it has no further activities to do). Session 3 is then closed through $cl3$ in step 10; the broker resumes its conversation with the client C_1, and forwards the non-availability message in step 11. The next steps close the session numbered 1 and the security framing implementing and enforcing the policy φ_1. The last transition continues the session involving the second client. The index R of the arrows shows that the transitions depend on the service providers constituting the cloud federation.

The index $\boldsymbol{\pi}$ is *plan.* i.e. a vector of functions mapping requests to services. The plan π_1 for the first client maps the request 1, originated by $\mathtt{open}_1$ of the client, to ℓ_{br}, and the request 3 from $\mathtt{open}_3$ of the broker, to ℓ_{s3}. We call π_1 *valid*, because it drives a computation where both the security constraints and compliance of clients/services are guaranteed.

Suppose now that the plan π_2 for the second client maps request 2 to ℓ_{br} and request 3 (from the second instance of the broker) to ℓ_{s2}. Since S_2 does not comply with B, at run-time a communication involving the action $\overline{Del}$ cannot occur because the broker has no action Del. Our assumption that the service can decide what to send on its own is violated. For this reason, we say that this plan is *not valid.* Finally, consider a plan that maps request 3 to ℓ_{s3}, that this time is compliant with the broker. However S_3 does not satisfy the configuration profile settled by C_2, and so a policy violation occurs; also this plan is not valid.

3 Model Checking Validity and Service Compliance

In our approach, a service S is identified by its interface, i.e. a type τ that makes some of the service functionality available across a network. Also each interface is annotated by the *history expression* H that over-approximates the possible run-time behaviour of S. We do not detail here the type and effect system used to associate this information with a service. A theorem guarantees that H is safe: when a service with interface τ and annotation H is run, it will generate one of the execution histories denoted by H. Hereafter we will often identify the service with its abstract behaviour H. The syntax of History Expressions follows.

$$H ::= \varepsilon \mid h \mid \mu h.H \mid (\sum_{i\in I} a_i.H_i) \mid (\bigoplus_{i\in I} \overline{a}_i.H_i) \mid \alpha \mid H\cdot H \mid \mathtt{open}_{r,\varphi}\, H,\ \mathtt{close}_{r,\varphi} \mid \varphi(\!|H|\!)$$

Intuitively, ε does nothing (thus $\varepsilon\cdot H \equiv H \equiv H\cdot\varepsilon$). Recursion is $\mu h.H$, restricted to be tail-recursive and guarded by communication actions $\bar{a}$ or a. Events α can occur, if they do not violate any active policy; they represent accesses to resources (see [8] for richer actions). The expression $H_1\cdot H_2$ is sequential composition. An expression sends/receives on a channel messages, indexed by a set I. To stress that the non-deterministic choice of the output $\overline{a}_i$ is up to the sender only (internal choice), we use $\oplus$, while the external choice only involves inputs a_i and is denoted by Σ. Security framing $\varphi(\!|H|\!)$ enforces the policy φ while H is running, (sometimes we equivalently write $(\!|_{\varphi}\cdot H\cdot|\!)_{\varphi}$ for $\varphi(\!|H|\!)$). A policy is a (sort of) finite state automaton that accepts those strings of access events that violate it, in the default-accept paradigm. Entering a security framing corresponds to calling a monitor checking that all the histories, i.e. the sequence of events previously fired, must respect the policy, in the style of *history-dependent* approaches to security [2]. A service is engaged in a session with another through $\mathtt{open}_{r,\varphi} H \mathtt{close}_{r,\varphi}$, where r is a unique identifier and φ is the policy to be enforced while the responding service is active. The operational semantics of is given by the transition system inductively defined by the following rules (we omit the standard rules for $H_1 \cdot H_2$ and $\mu h.H$):

$$\alpha \overset{\alpha}{\twoheadrightarrow} \varepsilon \quad (\textsc{Act}) \qquad \bigoplus_{i\in I} \overline{a}_i.H_i \overset{\overline{a}_i}{\twoheadrightarrow} H_i \quad (\textsc{I-Choice}) \qquad \sum_{i\in I} a_i.H_i \overset{a_i}{\twoheadrightarrow} H_i \quad (\textsc{E-Choice})$$

$$\varphi\llparenthesis H \rrparenthesis \overset{\llparenthesis_\varphi}{\twoheadrightarrow} H\cdot \rrparenthesis_\varphi \quad (\textsc{P-Open}) \qquad \mathtt{open}_{r,\varphi}.H.\mathtt{close}_{r,\varphi} \overset{open_{r,\varphi}}{\twoheadrightarrow} H.\mathtt{close}_{r,\varphi} \quad (\textsc{S-Open})$$

A network N is composed of the parallel composition of clients H, each hosted at a location $\ell \in \mathsf{Loc}$, and of sessions S involving a client and a service. Services are published in a global trusted repository $R = \{\ell_j : H_j \mid j \in J\}$, and they are always available for joining sessions. The syntax of networks follows, with that of plans $\boldsymbol{\pi}$, i.e. vectors of functions π'_i assigning a service ℓ to each request r, written $r[\ell]$.

$$N ::= N \| N \mid S \qquad\qquad S ::= \ell : H \mid [S, S]$$

The operational semantics of networks follows (Φ stores the active policies). For lack of space we omit that for syncronization (on complementary actions) and that driving the network to proceed when a component does.

$$\frac{H \overset{open_{r,\varphi}}{\twoheadrightarrow} H' \quad r[\ell_j] \in \pi \quad \{\ell_j : H_j\} \subseteq R \quad \models \eta\llparenthesis_\varphi}{\eta, \ell_i : H \xrightarrow{open_{r,\varphi}}_{\pi,R} \eta\llparenthesis_\varphi, [\ell_i : H', \ell_j : H_j]} \; (\textsc{Open})$$

$$\frac{H \overset{close_{r,\varphi}}{\twoheadrightarrow} H'}{\eta, [\ell_i : H, \ell_j : H''_j] \xrightarrow{close_{r,\varphi}}_{\pi,R} \eta\eta', \ell_i : H'} \; \eta' = \Phi(H''_j)\rrparenthesis_\varphi \quad (\textsc{Close})$$

$$\frac{H \overset{\gamma}{\twoheadrightarrow} H' \quad \models \eta\gamma}{\eta, \ell_i : H \xrightarrow{\gamma}_{\pi,R} \eta\gamma, \ell_i : H'} \; (\textsc{Access}) \qquad \frac{\eta, S \xrightarrow{\lambda}_{\pi,R} \eta', S' \quad \models \eta'}{\eta, [S, S''] \xrightarrow{\lambda}_{\pi,R} \eta', [S', S'']} \; (\textsc{Session})$$

The first property of our verification methodology concerns secure executions. Formally this is done by checking *validity* of histories. The idea is to suitably assemble the history expression H associated to a network of services, and recording in a plan for H which service to invoke for each request, so obtaining the pair $\hat{H}, \pi$. Note that $\hat{H}$ may be *non-valid*, even if the composing selected services are valid, each in isolation. Indeed, the impact on the execution history of selecting a service H_r for a request r is not confined to the execution of H_r, but it spans over the whole execution, because the active security policies are to be enforced on the whole execution history, as our approach is history-dependent. The validity of the composed service $\hat{H}$ depends thus on the global orchestration, i.e. on the plan π.

In order to ascertain the validity of history expressions we resort to model checking. Because of the possible nesting of security framings, validity of history expressions is a non-regular property, so standard model checking techniques can not be directly applied. In [6], a semantic-preserving transformation is presented, that removes the context-free aspects due to policy nesting: it suffices recording the opening of policies, and removing those already opened and their corresponding closures, in a stack-like fashion. In this way, (standard) model checking is efficiently feasible through specially-tailored finite state automata. A generalisation of this technique has been applied here to the extended definition of history expressions.

We now introduce the notion of service compliance. Given the service request $\mathtt{open}_{r,\varphi} H_1 \mathtt{close}_{r,\varphi}$ and the service H_2, we say that H_1 and H_2 are compliant if for every possible internal action of a party, the other can perform the corresponding co-action. Note that compliance does not require the service to terminate, because the client can terminate whenever all its operations have been completed.

We now outline our model-checking technique for verifying service compliance. The key idea is to reduce compliance to a safety property. First we manipulate the syntactic structure of a history expression to identify and pick up all the requests, i.e. the subterms $\mathtt{open}_{r,\varphi} H_1 \mathtt{close}_{r,\varphi}$. Then, to check compliance of a request r against a service H_2, we compute the projection of H_1 and H_2 on their communication actions. This projection removes from H_1 and H_2 all the resource access events α and policy opening and closing $\llparenthesis_{\varphi}, \rrparenthesis_{\varphi}$, as well as all the *inner* service requests, i.e. the subterms $\mathtt{open}_{r',\varphi'} \ldots \mathtt{close}_{r',\varphi}$ occurring inside H_1 and H_2.

Formally, the projection function of an history expression is much alike a behavioural contract as defined in [16]; so we call *contracts* these projected history expressions and we write them $H^!$. More precisely, the projection function produces a subset of those contracts, since in our history expressions the internal choice is always guarded by output actions and the external choice is always guarded by input actions. Finally, we only have guarded tail recursion. Because of the last restriction, it turns out that the transition system of $H^!$ is *finite state*; in other words there only is a finite number of expressions that are reachable from $H^!$ through the transitions defined by the operational semantics of history expressions in isolation.

We introduce now the notion of *product automaton* of two contracts. The product automaton models the behaviour of contracts composition where final states represent stuck configurations: these states are reached whenever the two contracts are not compliant. Let H_1' and H_2' be history expressions. The product automaton of $H_1 = (H_1')^!$ and $H_2 = (H_2')^!$ is $H_1 \otimes H_2 = \langle S_1 \times S_2, \{\tau\}, \delta, \langle H_1, H_2 \rangle, F \rangle$, where

- S_1 and S_2 are the states of H_1 and H_2, with $\langle H_1, H_2 \rangle$ the initial state
- $\{\tau\}$ is the alphabet, representing pairs of complementary action $\langle a, co(a) \rangle$
- the transition function δ is:
  $\delta = \{(\langle H_1, H_2 \rangle, \tau, \langle H_1', H_2' \rangle) | H_1 \overset{a}{\twoheadrightarrow} H_1' \wedge H_2 \overset{co(a)}{\twoheadrightarrow} H_2' \wedge \langle H_1, H_2 \rangle \notin F\}$
- the final states are $F = \{\langle H_1, H_2 \rangle | H_1 \neq \varepsilon \wedge (\neg(\exists \overline{a}.(H_1 \overset{\overline{a}}{\twoheadrightarrow} H_1' \vee H_2 \overset{\overline{a}}{\twoheadrightarrow} H_2')) \vee \neg((\forall H_1 \overset{\overline{a}}{\twoheadrightarrow} H_1', \exists H_2 \overset{a}{\twoheadrightarrow} H_2') \wedge (\forall H_2 \overset{\overline{a}}{\twoheadrightarrow} H_2', \exists H_1 \overset{a}{\twoheadrightarrow} H_1'))))\}$.

Intuitively, H_1 and H_2 are compliant if and only if the language of the product automaton $\mathcal{A}$ is empty. This basically corresponds to the model-checking technique of invariant properties (a subset of the safety properties) [5].

4 Verifying Services Secure and Unfailing

We have all the means to statically verify whether a network of services will evolve with neither security nor compliance violations. Given a federation of services R and a vector of clients, the verification is as follows: (i) compute through a type and effect system the history expressions over approximating the dynamic behaviours of services and clients; (ii) pick up one service at a time and generate a valid plan π_H for it; and (iii) for each request $\mathtt{open}_{r,\varphi} H_1 \mathtt{close}_{r,\varphi}$ in the composed service check H_1 compliant with H_2, with $\pi_H(r) \in R$. If all these steps succeed, switch off any run-time monitor, and live happily: nothing bad will happen. This result relies on suitable extensions of the methodology proposed in [6], and on a careful definition of service sessions, possibly nested, and of compliance. Since compliance is a safety property, mechanical verification via standard model-checking is feasible.

Our main result stating the soundness of our verification technique follows:

Theorem 1 *Secure and Compliant Services = Model Checking Validity & Compliance*

We established a novel connection between the worlds of service contracts and of security. We plan to extend our approach for modeling more carefully the availability of services, that now can unboundedly replicate themselves. We also wish to extend our verification methodology to include quantitative information in the security policies, along the lines of [18]. We would like also to modify the model checker and related tools of [10] so to completely mechanise our proposal.

References

1. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty contracts: Agreeing and implementing interorganizational processes. Comput. J. **53**(1), 90–106 (2010)
2. Abadi, M., Fournet, C.: Access control based on execution history. In: Network and Distributed System Security Symposium, NDSS 2003. The Internet Society (2003)
3. Armbrust, M., et al: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
4. Artikis, A., Sergot, M.J., Pitt, J.V.: Specifying norm-governed computational societies. ACM Trans. Comput. Log. **10**(1) (2009)
5. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
6. Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and verifying service composition. Journal of Computer Security **17**(5), 799–837 (2009)
7. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Model checking usage policies To appear in *Math. Stuct. Comp. Sci.*, abridged version in TGC 2008, vol 5474 LNCS (2009)
8. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. **31**(6) (2009)
9. Bartoletti, M., Tuosto, E., Zunino, R.: On the realizability of contracts in dishonest systems. In: M. Sirjani (ed.) COORDINATION, *LNCS*, vol. 7274, pp. 245–260. Springer (2012)
10. Bartoletti, M., Zunino, R.: LocUsT: a tool for checking usage policies. Tech. Rep. TR-08-07, Dip. Informatica, Univ. Pisa (2008)
11. Bartoletti, M., Zunino, R.: A calculus of contracting processes. In: LICS, pp. 332–341. IEEE Computer Society (2010)
12. Bernsmed, K., Jaatun, M.G., Meland, P.H., Undheim, A.: Security SLAs for federated cloud services. In: ARES, pp. 202–209. IEEE (2011)
13. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: CONCUR 2010 - Concurrency Theory, 21th International Conference, *Lecture Notes in Computer Science*, vol. 6269, pp. 162–176. Springer (2010)
14. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Comp. Syst. **25**(6) (2009)
15. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for web services. In: M. Bravetti, M. Núñez, G. Zavattaro (eds.) WS-FM, *LNCS*, vol. 4184, pp. 148–162. Springer (2006)
16. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. **31**(5) (2009)
17. Castagna, G., Padovani, L.: Contracts for mobile processes. In: M. Bravetti, G. Zavattaro (eds.) CONCUR, *LNCS*, vol. 5710, pp. 211–228. Springer (2009)
18. Degano, P., Ferrari, G.L., Mezzetti, G.: On quantitative security policies. In: V. Malyshkin (ed.) PaCT, *LNCS*, vol. 6873, pp. 23–39. Springer (2011)
19. Papazouglou, M., Georgakopoulos, D.: Special issue on service oriented computing. Communications of the ACM **46**(10) (2003)