Service Interaction Contracts as Security Policies

Davide Basile

Dipartimento di Informatica, Università di Pisa, Italy
basile@di.unipi.it
http://www.di.unipi.it/~basile/

Abstract. We outline a methodology to check the compliance of service orchestration with respect to service contracts. The contract of a service is expressed by a finite state automaton, while the traces of the orchestration form a context-free language. A contract asserts the security policy controlling resource accesses, including accesses to the communication channels, so making manifest also the client-service interactions. The key idea of the methodology is controlling both access control and compliance by an appropriate model-checking technique. Our approach naturally deals with multi-party contracts.

Keywords: service contracts, compliance, model checking usages

1 Introduction

In Service-Oriented Computing (SOC), applications are built by combining different distributed components, called services. Standard communication protocols are used for the interaction between the parties. Service composition depends on which information about the services is made public. Security issues can make more complex service composition, since a service can impose constraints on the interactions it can hold. Also, the descriptions of services lack semantic information. Behavioral contracts have been introduced to describe the external observable behavior of a service, and can be used for guaranteeing progress. This property ensures that the whole system will never get stuck, i.e. all the components are able to successfully terminate their tasks. We consider two different paradigms for describing contracts. The contracts of Castagna et al. [3, 4] take the form of CCS processes and permit to describe if the interaction between two parties terminates or gets stuck, and when a service can be replaced with a more general one. Instead Bartoletti et al. [1, 2] introduce a core calculus for services that extends the λ-calculus with primitives for composing services in a call-by-contract fashion under security properties. They develop a static technique for extracting the abstract behaviour of a service (called History Expression) that must obey the security policies. An orchestration machinery constructs a plan, i.e. a binding between requests and services guaranteeing that the security properties are always satisfied.
We extends History Expressions to include channel communications and internal/external choice for combining the notions of security access and progress of

interactions, so merging and enriching the above surveyed approaches. We prove that compliance between client and server is a safety property. The main novelty of our approach lies in exploiting standard techniques of model checking for controlling compliance of behavioural contracts. Also differently from [1–3] we manage both multi-party contracts and sessions. Due to lack of space, we cannot compare in more detail the vaste existing literature in this field, and refer the interested reader to [6].

2 Programming Model and Verification

History Expressions (HE) are a suitable process calculus through which we abstractly describe services. Beside the standard operations of process calculi, namely I/O operations, prefixing, concatenation, guarded (tail) recursion, a History Expression H contain access events α, and two non-deterministic choice operators. The external choice $\sum_{i\in I} a_i.H_i$ proceeds according to a value received on channel a_i from the external environment; while internal choice $\bigoplus_{i\in I} \overline{a}_i.H_i$ describes a service that internally decides whether to continue with one of the summands $\overline{a}_i.H_i$. Additionally, HE have the framing $\varphi(\!|H|\!)$ specifying that the policy φ must be enforced in H. A policy is an FSA that recognizes strings of access events. An example of safety policy φ is "never perform write actions(α_{write}) after read actions(α_{read})". A trace that violates this policy is $\alpha_{read}\alpha_{write}$. Finally, HE describe the opening and closing of a session by the expression $\texttt{open}_{r,\varphi} H \texttt{close}_r$, where r represents the unique identifier of the request (i.e. a point in the abstract syntax tree of H) and φ is the policy that the responding service must obey. Inside the session, two services can synchronize on I/O actions. Two services are *compliant* if for every output action the other party is ready to perform the corresponding input action. Every services is published at a location ℓ. An orchestrator statically creates a plan π which is a binding between the request r and the location of the service choosen for opening the session. Only two services are involved in a session. Nested sessions are possible, since a service involved in a session can open a new session with another service. A plan π is valid if the two services are compliant and the server does not violates the policy φ. Finally a network N is the parallel composition of different services and sessions.
The operational semantic is defined by a transition system. The configurations of a network have the form $R \rhd N$, where R is a set of services and N is the active network, i.e. the active clients and services. The set R is partitioned into two parts: $(1)\{\ell_i : H_i\}^{?}_{i\in 1\ldots k}$ is the set of stand-by available services that can be invoked with a $\texttt{open}_{r,\varphi}$ operation and $(2)\{\ell_i : H_i\}^{!}_{i\in 1\ldots k'}$ is the set of busy services, which are involved in sessions. To help intuition, we work out the following running example. Consider the services:

$$H_1 = a \cdot (\texttt{open}_{2,\varphi_2}\, \overline{d}.(e+f)\ \texttt{close}_2) \cdot (\overline{b} \oplus \overline{c}) \cdot d \quad H_2 = \beta \cdot d.(\overline{e} \oplus \overline{f}) \cdot \alpha$$
$$H_3 = a.\overline{g} \quad H_4 = \texttt{open}_{1,\varphi_1}\, \overline{a}.(b+c)\ \texttt{close}_1 \quad H_5 = \texttt{open}_{3,\varphi_3}\, \overline{a}.g\ \texttt{close}_3$$

We can see that H_2 performs the access events α, β. Let φ_2 say "never β after α", while the actual definitions of φ_1 and φ_3 is immaterial. By abuse of notation,

when it is clear from the context, we indentify a service with the location where it is running. Let the initial configuration of the network be:

$$\{\ell_1 : H_1, \ell_2 : H_2, \ell_3 : H_3\}^? \cup \{\}^! \rhd \ell_4 : H_4 \| \ell_5 : H_5$$

Assume that the orchestrator plan is of the form $\pi = \bigcup_{i\in\{1,2,3\}}(r_i, \ell_i)$. We can see that all the services are compliant and ℓ_2 respects the policy φ_2. Suppose now that ℓ_4 fires the $\mathtt{open}_{1,\varphi_1}$ operation, we have:

$$\{\ell_2 : H_2, \ell_3 : H_3\}^? \cup \{\ell_1 : H_1\}^! \rhd [\ell_4 : \overline{a}.(b + c)\ \mathtt{close}_1, \ell_1 : \varphi_1(\!|H_1|\!)] \| \ell_5 : H_5$$

Now ℓ_1 is engaged in the session with the service ℓ_4, because $\pi(r_1) = \ell_1$. The service ℓ_1 is marked busy in R and its behaviour is checked against φ_1. The service ℓ_5 opens a new session, and the resulting configuration becames:

$$\{\ell_2 : H_2\}^? \cup \{\ell_1 : H_1, \ell_3 : H_3\}^! \rhd [\ell_4 : \ldots, \ell_1 : \ldots] \| [\ell_5 : \overline{a}.g\ \mathtt{close}_3, \ell_3 : \varphi_3(\!|H_3|\!)]$$

There are two parallel sessions. The services ℓ_5 and ℓ_3 will synchronize on the channels a and g so that ℓ_5 closes the session and terminates, restoring ℓ_3 to its initial state as an available service. Then ℓ_1 and ℓ_4 synchronize on channel a:

$$\{\ell_2 : H_2, \ell_3 : H_3\}^? \cup \{\ell_1 : H_1\}^! \rhd [\ell_4 : (b + c)\ \mathtt{close}_1, \ell_1 : \mathtt{open}_{2,\varphi_2} \ldots]$$

The service in ℓ_3 turns back to available service. Now ℓ_1 opens a new session with ℓ_2 while ℓ_4 is waiting:

$$\{\ell_3 : H_3\}^? \cup \{\ell_1 : H_1, \ell_2 : H_2\}^! \rhd [\ell_4 : \ldots, [\ell_1 : \ldots, \ell_2 : \varphi_2(\!|H_2|\!)]]$$

This is a nested session: ℓ_1 and ℓ_2 synchronize, note that φ_2 is respected. Eventually ℓ_1 closes the session:

$$\{\ell_3 : H_3, \ell_2 : H_2\}^? \cup \{\ell_1 : H_1\}^! \rhd [\ell_4 : (b + c)\ \mathtt{close}_1, \ell_1 : (\overline{b} \oplus \overline{c}) \cdot d]$$

Here ℓ_4 will receive the input on channel b or c and it will close the session. We can see that ℓ_1 could receive another message on channel d, but since the session is closed it gets back to its initial state. The final configuration is:

$$\{\ell_1 : H_1, \ell_2 : H_2, \ell_3 : H_3\}^? \cup \{\}^! \rhd \epsilon$$

For generating a valid plan we perform three steps. The first two find the compliant services for each request and check if the selected service respects the policy φ imposed by the client. For checking compliance of a given client with a sub-term of the form $\mathtt{open}_{r,\varphi} H_1 \mathtt{close}_r$ and an available service $\ell_2 : H_2$; we calculate a projection of H_1 and H_2 on their communication actions; operationally we remove all the policies φ, all the access events α and all the sub-terms $\mathtt{open}_{r',\varphi'} \ldots \mathtt{close}_{r'}$ nested in H_1 and H_2. Then, we make the product automaton $\mathcal{A}$ of the resulting transition systems. We only have finitely many states in $\mathcal{A}$. We fully characterize compliance of services by checking in each state that the client has not terminated and for all the possible output actions that a service is ready to fire, the other party is ready to perform the corresponding input action. We also check that at least one of the two services can perform one output. It turns out that compliance is an invariant property: a subset of the safety properties [5]. Now H_1 and H_2 are compliant if and only if the language of the product automaton $\mathcal{A}$ is empty: no final states exist in which the above condition do not hold. We also have to check if the choosen service respects the policy φ. For doing so we

discard all the communication actions (possibly transforming $\oplus$ in $+$). Finally, an important property is that a History Expression H is valid under a policy φ if and only if $[\![H]\!] \cap [\![\varphi]\!] = \emptyset$, i.e. if and only if the languages $[\![H]\!]$ of the traces of access events of H and $[\![\varphi]\!]$ of the offending traces of φ do not intersect. Since $[\![H]\!]$ is context-free and $[\![\varphi]\!]$ is regular, and emptiness of a context-free language is decidable, so is our problem. Indeed several algorithms and tools show this approach feasible. The following example explains how to resolve the request $\texttt{open}_{2,\varphi_2}$ occuring in H_1 of our running example. The projection of H_2 on his communication actions give raise to the service $d.(\overline{e} \oplus \overline{f})$. The product automaton $\overline{d}.(e+f) \otimes d.(\overline{e} \oplus \overline{f})$ has three states: $\{\langle \overline{d}.(e+f), d.(\overline{e} \oplus \overline{f})\rangle, \langle e+f, \overline{e} \oplus \overline{f}\rangle, \langle \epsilon, \epsilon\rangle\}$. The set of final states is empty: no state satisfies the conditions described above. Recall that φ_2 says "never β after α", the projection of H_2 on the access events give raise to the service $\beta \cdot \alpha$, we have $[\![\varphi_2]\!] = \{\Sigma^* \alpha \Sigma^* \beta \Sigma^*\}$ and $[\![\beta \cdot \alpha]\!] = \{\beta \cdot \alpha\}$ and the intersection is trivially the empty language, therefore the two services are compliant and $(r_2, \ell_2) \in \pi$. Proceeding in this way, we generate the set of compliant services for each request. Finally the third last step ensures that a service never gets stuck while is waiting to opening a session. First we generate a "locally" viable plan for each service. One of the compliant services for each request is selected such that it will never be the case that two nested request r_1, r_2 are resolved by the same service. For example, the following network has no "locally" viable plan:
$\{\ell_2 : \overline{a}.(b+c+d+e)\}^? \triangleright \ell_1 : \texttt{open}_{1,\_} a.\texttt{open}_{2,\_} a.(\overline{b} \oplus \overline{c})\texttt{close}_2(\overline{b} \oplus \overline{c} \oplus \overline{d})\texttt{close}_1$
Indeed the service at ℓ_2 is compliant with both the request r_1 and r_2. The plan $\pi = \{(r_1, \ell_2), (r_2, \ell_2)\}$ is not locally viable for ℓ_1: the service at ℓ_2 will never be available to open the session r_2 since it is still involved in r_1.
The algorithm for generating a locally viable plan for a service at every iteration picks the outermost *open/close* subterm and selects one of the services compliant with that request. Then it removes from the selected service the set of compliant services of the nested *open/close* subterms. All the locally viable plans are merged into a global plan. The tecnique we adopt consists in building the state graph of the network and in checking that there are no cycles where a service is able to open a session and it will never do it. Finally we plan to implementing the algorithms outlined above in one of the existing model-checker.

References

1. Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, Roberto Zunino: Call-by-Contract for Service Discovery, Orchestration and Recovery. *LNCS: 232-261.*
2. Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari: Planning and verifying service composition. *Journal of Computer Security 17(5): 799-837 (2009)*
3. Giuseppe Castagna, Niel Gesbert, Luca Padovani: A Theory of Contracts for Web Services. *ACM TOPLAS*, 31(5), 2009.
4. Giuseppe Castagna and Luca Padovani: Contracts for mobile processes. *LNCS: 211-228* 2009.
5. C. Baier and J.-P. Katoen: Principles of Model Checking. *MIT Press*, 2008.
6. Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, Roberto Zunino: Local policies for resource usage analysis. *(TOPLAS)* 31(6) (2009).