Modular Plans for Secure Service Composition

Gabriele Costa, Pierpaola Degano, Fabio Martinelli

Modular Plans for Secure Service Composition *

Gabriele Costa[1,2], Pierpaolo Degano[2], and Fabio Martinelli[1]

[1] Istituto di Informatica e Telematica – Consiglio Nazionale delle Ricerche
[2] Dipartimento di Informatica – Università di Pisa

Abstract. Service Oriented Computing (SOC) is a programming paradigm aiming at characterising Service Networks. Services are entities waiting for clients requests and they often result from the composition of many services.
We address here the problem of statically guaranteeing security of open services, i.e. services with unknown components. Security constraints are expressed by local policies that service components must obey.
We present here a type and effect system that safely over-approximates, in the form of history expressions, the possible run-time behaviour of open services, collecting partial information on the behaviours of their components. From a history expression, we then extract a plan that drives executions that never rise security violations.
Finally, we show how partial plans satisfying security requirements can be put together to obtain a safe orchestration plan.

1 Introduction

In the last decade, *history-based security* [1] has emerged as a powerful approach to guarantee that program do not misbehave. Histories, i.e. execution traces, are sequences of relevant actions generated by running programs. Hence, a *security policy* consists in a distinction between accepted and rejected traces. A program is considered secure if all its executions obey the given security policies. Many techniques use histories for deciding whether a program is secure or not. Mainly, two distinct methods exist: *run-time enforcement* and *static verification* (e.g. see [20,10,19,15]).

Briefly, policy enforcement consists in applying a monitor to the running program. Whenever the monitor target tries to perform an action, the current trace is checked against the enforced policy. A security violation, i.e. an attempt to extend the actual trace to a forbidden one, activates some emergency operation, e.g. the program termination. Instead, static verification aims to prove program security regardless of any possible execution scenario. Roughly, one proves that the program can only produce policy-compliant traces by only looking at the program text. Some authors use a mixed approach (e.g. [17]) that combines a static check with a dynamic one in case the first fails. Often, security policies

* Work partially supported by EU-funded project "SENSORIA", by EU-funded project "CONSEQUENCE" and by EU-funded project "CONNECT".

are expressed as global safety properties of systems and in this case execution monitors rely on finite state automata for detecting illegal traces.

Recently, [3] proposed a more flexible approach in which so-called *local policies* only guard portions of code and have a local scope. Originally, local policies have been coupled with an extension on the λ-calculus used to specify services [4]. Then, also Java has been extended [2] with local policies showing the feasibility of this approach on a real-world language.

In this paper, we use *right-bounded* local policies, a variant of those defined in [3]; for brevity we will omit in the following "right-bounded" whenever unambiguous. The main difference between the two versions is that in the original one an active policy is checked on the whole past execution history, while in our proposal it is checked on the history produced since the policy is activated (see Tab. 2). In the next section we shall introduce only intuitively *usage automata* that are used to specify local policies. Usage automata are a variant of nondeterministic finite state automata, parameterized over system resources. More details and the formal definition are in [6,7].

The approach can be summarized as follows. The specification of a system already contains the local policies to be obeyed. Compliance is then established by model-checking a suitable, safe abstraction of the system behaviour (for the λ-calculus this is done through a type and effect system, for Java through a control flow analysis).

Systems may behave differently according to how requests are resolved into services. The association between requests and services, called *plan*, drives the execution at run-time. If the plan determines an abstract behaviour passing the model-checking phase (the plan is viable) then the original code can be run securely with no monitoring.

The above approach only deals with closed networks, i.e. with services whose components are fully specified a priori. However, services are built incrementally and components may appear and disappear dynamically. It is important then to also analyse *open* networks, i.e. networks having unspecified participants. As a matter of fact, service-oriented paradigms aim to guarantee compositional properties and should be independent from the actual implementation of (possibly unknown) parties. Moreover, closed networks orchestrated by a global, composition plan require to be completely reorganized whenever a service falls or a new one rises.

Here, we extend the results of [4,5] on open networks. In particular, we will single out *partial* plans that only involve parts of the known network and that however can be safely adopted within any operating context. We also show how these partial plans can be combined together, along with the composition of the services they come from. As expected, composability of viable plans is subject to some constraints, and we also outline a possible way to efficiently check when these constraints are satisfied.

The paper is structured as follows. Section 2 introduces our motivating example. To carry on our investigation in a pure framework, we use here an extension of λ^{req}[4], the syntax and semantics of which are in Section 3. Section 4 intro-

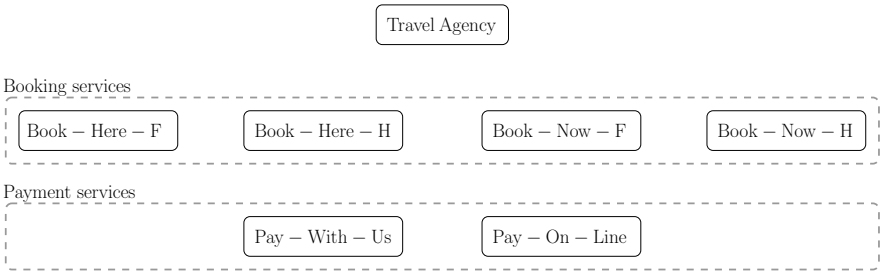
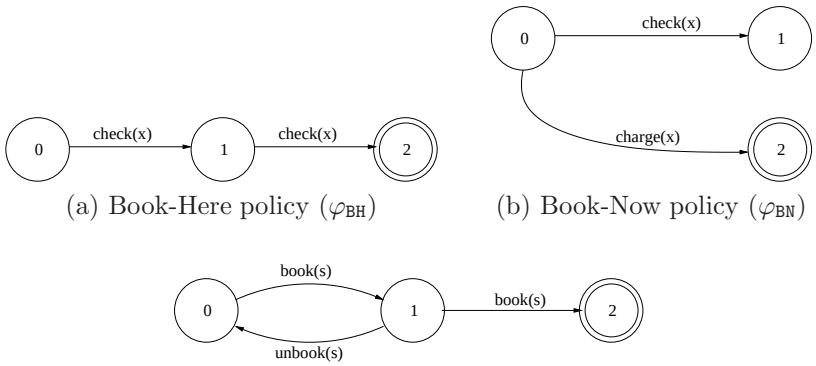


Fig. 1. A travel booking network

duces history expressions and our type and effect system. In Section 5 we provide our results on plans compositionality and propose a way to efficiently check their validity. Finally, Section 6 concludes the paper.

2 An example



(a) Book-Here policy ($\varphi_{\mathtt{BH}}$) (b) Book-Now policy ($\varphi_{\mathtt{BN}}$)

(c) Travel Agency policy ($\varphi_{\mathtt{TA}}$)

Fig. 2. Security policies as usage automata (self-loops are omitted)

In this section we introduce our working example. Figure 1 shows a simple service network for travel booking. Rounded boxes denote service locations (dashed lines contain homogeneous services, i.e. services offering similar functionalities). Clients contact the travel agency providing a credit card number for payments and receive back a receipt. Every instance of the travel agency books exactly one room and one flight. The responsibility of doing an actual reservation is delegated to booking services. Each booking service receives a card number and uses it for paying a reservation. Payment services are in charge for authorising a purchase. A payment service charges the needed amount on the credit

card (possibly after checking some credentials of the card number), and returns `TRUE`. Otherwise, it answers `FALSE` if a failure occurs.

Each service has its own security requirements expressed through a corresponding usage automaton. For example, a booking service would like to "perform at least one availability check before each payment". This is represented by Fig. 2b: every action but `check`(x) and `charge`(x) is allowed and leaves the automaton in state 0 (self-loops on actions not labelling existing arcs are omitted in Fig. 2). After a check, any action is permitted in state 1, including charging. Instead, a charge action in state 0 leads to the offending state 2. Additionally, booking service can be forbidden to check more than once ($\varphi_{\mathtt{BH}}$). Similarly, the travel agency can declare different rules focussing on different aspects, e.g. "never book two times the same service (hotel or flight)" ($\varphi_{\mathtt{TA}}$).

Clients are unspecified, so the network is open. However, clients do not affect at all the security policies introduced so far. We can therefore check secure this open network.

As a matter of fact, services only put security constraints on their own traces and on those of the services they invoke. This is because our policies are right-bounded and so their scope spans from their activation until their deactivation points. Note that in this way, the services we model cannot discriminate against clients. Everyone can invoke a service, because service behaviour is client-independent.

3 Service network

The programming model for service composition based on λ^{req} was first introduced in [4]. Its syntax resembles the classical call-by-value λ-calculus with two main differences: *security framing* and a call-by-contract *service request*.

Service networks are set of services. Each service e is hosted in a location ℓ. As expected, we assume that there exists a public *service repository* `Srv`. An element of `Srv` has the form $e_\ell : \tau \xrightarrow{H} \tau'$, where e_ℓ is the code of the service, ℓ is a unique service location (sometimes used to denote the service itself), $\tau \to \tau'$ is the type of e_ℓ, and H is the effect of e_ℓ (see Section 4.1).

3.1 Syntax

The syntax of λ^{req} is in Table 1. The expression $*$ represents a fixed, closed, event-free value. Resources, ranged over by r, r', belong to finite, statically defined classes $\mathcal{R}, \mathcal{R}'$. Access events α, β are side effects, parametrized over resources.

Function abstraction and application follow the classical syntax (we use z in $\lambda_z x.e$ as a binding to the function in e). Security framing applies the scope of a policy φ to a program e. Service request requires more attention. We state that services can not be directly accessed (for instance using a public name or address). Clients invoke services through their public interface, i.e. their type. The label ρ is a unique identifier associated with the request. A policy φ is attached to the request in a call-by-contract fashion: the invoked service must

$e, e' ::=$		
	$*$	unit
	r	resource
	x	variable
	$\alpha(e)$	access event
	$\mathtt{if}\, g\, \mathtt{then}\, e \;\; \mathtt{else} \;\; e'$	conditional
	$\lambda_z x.e$	abstraction
	$e\, e'$	application
	$\varphi[e]$	security framing
	$\mathtt{req}_\rho\, \tau \xrightarrow{\varphi} \tau'$	service request

Table 1. Syntax of λ^{req}

obey the policy φ. Since both τ and τ' can be higher-order types, we can model simple value-passing interaction, as well as mobile code scenarios.

We use v to denote values, i.e. resources, variables, abstractions and requests. Moreover, we introduce the usual abbreviations: $\lambda x.e = \lambda_z x.e$ with $z \notin fv(e)$, $\lambda.e = \lambda x.e$ with $x \notin fv(e)$ and $e; e'$ for $(\lambda.e')e$.

Here we make explicit the conditions of branching, similarly to [11]. Briefly, we check equality between resources, and we have negation and conjunction.

Definition 1. *(Syntax of guards)*

$g, g' ::= true \mid [\bar{x} = \bar{y}] \mid \neg g \mid g \wedge g'$	*($\bar{x}$,$\bar{y}$ range over variables and resources)*

We use $false$ as an abbreviation for $\neg true$, $[\bar{x} \neq \bar{y}]$ for $\neg\,[\bar{x} = \bar{y}]$ and $g \vee g'$ for $\neg(\neg g \wedge \neg g')$. We also define as expected an *evaluation function* $\mathcal{B}$ mapping guards into boolean values, namely $\{tt, ff\}$. E.g. $\mathcal{B}([\bar{x} = \bar{x}]) = tt$ and $\mathcal{B}([\bar{x} = \bar{y}]) = ff$ if $\bar{x} \neq \bar{y}$. In our model we assume resources to be uniquely identified by their (global) name, i.e. r and r' denote the same resource if and only if $r = r'$. Moreover, we use $[\bar{x} \in D]$ for $\bigvee_{d \in D} [\bar{x} = d]$.

Example 1. We specify the services Travel-Agency, Book-Here-S, Book-Now-S ($S \in \{F, H\}$), Pay-On-Line and Pay-With-Us in Fig. 1, through $e_{\mathtt{TA}}$, $e_{\mathtt{BH-S}}$, $e_{\mathtt{BN-S}}$, $e_{\mathtt{POL}}$ and $e_{\mathtt{PWU}}$ (resp.) as follows.

$$e_{\mathtt{PWU}} = \lambda x.\mathtt{if}\ [x \in \mathcal{C}]\ \mathtt{then}\ \mathtt{charge}(x); \mathtt{TRUE}\ \mathtt{else}\ \mathtt{FALSE}$$

$$e_{\mathtt{POL}} = \lambda x.\mathtt{if}\ [x \in \mathcal{C}']\ \mathtt{then}\ \mathtt{check}(x); \mathtt{charge}(x); \mathtt{check}(x); \mathtt{TRUE}\ \mathtt{else}\ \mathtt{FALSE}$$

$$e_{\mathtt{BH-S}} = \lambda x.(\lambda y.\mathtt{if}\ [y = \mathtt{TRUE}]\ \mathtt{then}\ \mathtt{book}(S)\ \mathtt{else}\ *)\ ((\mathtt{req}_\rho\, Card \xrightarrow{\varphi_{\mathtt{BH}}} Bool)x)$$

$$e_{\mathtt{BN-S}} = \lambda x.\mathtt{book}(S); (\lambda y.\mathtt{if}\ [y = \mathtt{FALSE}]\ \mathtt{then}\ \mathtt{unbook}(S)\ \mathtt{else}\ *)\ ((\mathtt{req}_{\rho'}\, Card \xrightarrow{\varphi_{\mathtt{BN}}} Bool)x)$$

$$e_{\mathtt{TA}} = \lambda x.\varphi_{\mathtt{TA}}\big[(\mathtt{req}_{\bar{\rho}}\, Card \to \mathbf{1})x; (\mathtt{req}_{\bar{\rho}'}\, Card \to \mathbf{1})x; rcpt\big]$$

Briefly, $e_{\texttt{PWU}}$ receives a card number x, verifies whether it is a registered one (i.e. $[x \in \mathcal{C}]$) and charges on x. Service $e_{\texttt{POL}}$ works similarly, but verifies money availability (event `check`) before and after the operation.

Booking services $e_{\texttt{BH}-\texttt{S}}$ require a payment and then, if it has been authorised, book the hotel (flight). On the contrary, $e_{\texttt{BN}-\texttt{S}}$ book the hotel (flight) and then cancel the reservation if the payment has been refused. Both $e_{\texttt{BH}-\texttt{S}}$ and $e_{\texttt{BN}-\texttt{S}}$ require the behaviour of the invoked service to comply with the policies $\varphi_{\texttt{BH}}$ and $\varphi_{\texttt{BN}}$.

Finally $e_{\texttt{TA}}$ simply calls two booking service instances and releases a receipt (*rcpt* of type *Rec*) to the client. Note that now the travel agency applies its policy $\varphi_{\texttt{TA}}$ to the sequential composition of service requests.

3.2 Operational semantics

Clearly, the run-time behaviour of a network of services depends on the way they interact. As we already mentioned, requests do not directly refer to a specific service that will be actually invoked during the execution, but to its behaviour, i.e. its type and effect (defined below), only. A *plan* resolves the requests by associating them with services locations. Needless to say, different plans lead to different executions. A service network can have many, one or even no valid plans. A plan is said to be *valid* if and only if the executions it drives complies with all the active security policies. More precisely, an execution trace η is valid w.r.t. a policy φ, in symbols $\eta \models \varphi$, if and only if it is not a word in the language of the security automaton of φ.

As expected, we assume that services can not be composed in a circular way. This condition amounts to say that there exists a partial order relation $\prec$ over services. We define $\texttt{Srv}\rfloor_\ell = \{e_{\ell'} : \tau \in \texttt{Srv} \mid \ell \prec \ell'\}$ as the sub-network that can be seen from a service hosted at ℓ.

A computational step of a program is a transition from a source configuration to a target one. In our model, configurations are pairs η, e where η is the execution *history*, that is the sequence of action events done so far (ε being the empty one), and e is the expression under evaluation. Actually, the syntax of histories and expressions is slightly extended with markers $[^m_\varphi$ as explained in the comment to the rule for framing. Note that the automaton for a policy φ will simply ignore these markers.

Formally, a plan is a (partial) mapping from request identifiers $(\rho, \rho', \ldots)$ to service locations $(\ell, \ell', \ldots)$ defined as

$$\pi, \pi' ::= \emptyset \mid \{\rho \mapsto \ell\} \mid \pi; \pi'$$

An empty plan $\emptyset$ is undefined for any request, while a singleton $\{\rho \mapsto \ell\}$ is only defined for ρ. Plan composition $\pi; \pi'$ combines two plans. It is defined if and only if for all ρ such that $\rho \in dom(\pi) \cap dom(\pi') \Rightarrow \pi(\rho) = \pi'(\rho)$, i.e. the same request is never resolved by different services. Two such plans are called *modular*.

Given a plan π we evaluate λ^{req} expressions, i.e. services, accordingly to the rules of the operational semantics in Table 2. Actually, a transition should

$$(\mathtt{S{-}Ev_1}) \quad \frac{\eta, e \to_\pi \eta', e'}{\eta, \alpha(e) \to_\pi \eta', \alpha(e')} \qquad (\mathtt{S{-}Ev_2}) \quad \eta, \alpha(r) \to_\pi \eta\alpha(r), *$$

$$(\mathtt{S{-}If}) \quad \eta, \mathtt{if}\ g\ \mathtt{then}\ e_{tt}\ \mathtt{else}\ e_{ff} \to_\pi \eta, e_{\mathcal{B}(g)} \qquad (\mathtt{S{-}App_1}) \quad \frac{\eta, e_1 \to_\pi \eta', e_1'}{\eta, e_1 e_2 \to_\pi \eta', e_1' e_2}$$

$$(\mathtt{S{-}App_3}) \quad \eta, (\lambda_z x.e)v \to_\pi \eta, e\{v/x, \lambda_z x.e/z\} \qquad (\mathtt{S{-}App_2}) \quad \frac{\eta, e_2 \to_\pi \eta', e_2'}{\eta, ve_2 \to_\pi \eta', ve_2'}$$

$$(\mathtt{S{-}Frm_1}) \quad \eta, \varphi[e] \to_\pi \eta[^m_\varphi, \varphi^m[e] \quad m \text{ fresh} \qquad (\mathtt{S{-}Frm_2}) \quad \eta[^m_\varphi \eta', \varphi^m[v] \to_\pi \eta\eta', v$$

$$(\mathtt{S{-}Frm_3}) \quad \frac{\eta[^m_\varphi \eta', e \to_\pi \eta[^m_\varphi \eta'', e' \quad \eta'' \models \varphi}{\eta[^m_\varphi \eta', \varphi^m[e] \to_\pi \eta[^m_\varphi \eta'', \varphi^m[e']}$$

$$(\mathtt{S{-}Req}) \quad \frac{e_{\bar{\ell}} : \tau \xrightarrow{H} \tau' \in \mathtt{Srv}\rfloor_\ell \qquad \pi(\rho) = \bar{\ell} \qquad H \models \varphi}{\eta, (\mathtt{req}_\rho\, \tau \xrightarrow{\varphi} \tau')v \to_\pi \eta, e_{\bar{\ell}} v}$$

Table 2. Operational semantics of λ^{req}

be also labelled by the location ℓ hosting the expression under evaluation. For readability, we omit this label.

Briefly, an action $\alpha(r)$ is appended to the current history (possibly after the evaluation of its parameter), a conditional branching chooses between two possible executions (depending on its guard g) and application works as usual. The rule $(\mathtt{S{-}Frm_1})$ opens an instance of framing $\varphi[e]$ and records the activation in the history with a marker $[^m_\varphi$, and in the expression with $\varphi^m[e]$ (to keep different instantiations apart we use a fresh m). The rule $(\mathtt{S{-}Frm_2})$ simply deactivates the framing and correspondingly purges the relevant marker from the history. The rule $(\mathtt{S{-}Frm_3})$ checks whether the m-th instantiation of φ is respected by the history since it has been activated (recall that the usage automaton for φ ignores the markers $[^{m'}_{\varphi'}$, where of course $m' \neq m$). This is how our right-bounded local mechanism is implemented. A service request firstly retrieves the service $e_{\bar{\ell}}$ that the current plan π associates with ρ, and that belongs to $\mathtt{Srv}\rfloor_\ell$, i.e. visible from the present evaluation location ℓ (omitted in the rule, as stipulated above). Then, the effect of the selected service is checked against the policy φ required by the client. If successful, the service is finally applied to the value provided by the client.

Example 2. Consider the service *Book-Here-F* as implemented in Example 1. If it is invoked with parameter $c \in \mathcal{C}$ starting from an empty trace, its execution under the plan $\pi = \{\rho \mapsto \mathtt{PWU}\}$ is:

$\varepsilon, (e_{\mathtt{BH-F}}\ c)$

$$\to_\pi \varepsilon, (\lambda_z y.\texttt{if}\ [y = \texttt{TRUE}]\ \texttt{then}\ \texttt{book}(F)\ \texttt{else}\ *)((\texttt{req}_\rho\ Card \xrightarrow{\varphi_{\texttt{BH}}} Bool)c)$$

$$\to_\pi \varepsilon, (\lambda_z y.\texttt{if}\ [y = \texttt{TRUE}]\ \texttt{then}\ \texttt{book}(F)\ \texttt{else}\ *)((e_{\texttt{PWU}})c)$$

$$\to_\pi \varepsilon, (\lambda_z y.\texttt{if}\ [y = \texttt{TRUE}]\ \texttt{then}\ \texttt{book}(F)\ \texttt{else}\ *)\ (\texttt{if}\ [c \in \mathcal{C}]\ \texttt{then}\ \texttt{charge}(c); \texttt{TRUE}\ \texttt{else}\ \texttt{FALSE})$$

$$\to_\pi \varepsilon, (\lambda_z y.\texttt{if}\ [y = \texttt{TRUE}]\ \texttt{then}\ \texttt{book}(F)\ \texttt{else}\ *)(\texttt{charge}(c); \texttt{TRUE})$$

$$\to_\pi \texttt{charge}(c), (\lambda_z y.\texttt{if}\ [y = \texttt{TRUE}]\ \texttt{then}\ \texttt{book}(F)\ \texttt{else}\ *)(\texttt{TRUE})$$

$$\to_\pi \texttt{charge}(c), \texttt{if}\ [\texttt{TRUE} = \texttt{TRUE}]\ \texttt{then}\ \texttt{book}(F)\ \texttt{else}\ *$$

$$\to_\pi \texttt{charge}(c), \texttt{book}(F)$$

$$\to_\pi \texttt{charge}(c)\texttt{book}(F), *$$

Example 3. Let $z = \lambda_z x.\varphi[\alpha; z(x)]$ and let φ be the policy saying that "a single α is allowed". Then, we have the following computation (under the empty plan).

$$\varepsilon, (z\ *) \to_\emptyset \varepsilon, \varphi[\alpha; z(*)] \to_\emptyset [^1_\varphi, \varphi^1[\alpha; z(*)] \to_\emptyset [^1_\varphi \alpha, \varphi^1[z(*)]$$

Note that the last transition is possible because $\alpha \models \varphi$. Then the computation proceeds as follows.

$$\to_\emptyset [^1_\varphi \alpha, \varphi^1[\varphi[\alpha; z(*)]] \to_\emptyset [^1_\varphi \alpha [^2_\varphi, \varphi^1\left[\varphi^2[\alpha; z(*)]\right]$$

This configuration is stuck. Indeed, a further step would lead to the configuration $[^1_\varphi \alpha [^2_\varphi \alpha, \varphi^1\left[\varphi^2[z(*)]\right]$. In spite of the fact that the second instance of φ is satisfied by the suffix α of the history after the marker $[^2_\varphi$, the first instance is not: $\alpha[^2_\varphi \alpha \not\models \varphi$ (recall that the marker $[^2_\varphi$ is invisible to φ).

4 Type and effect system

We now introduce the type and effect system for λ^{req}. Our system builds upon [4] introducing two new rules: guarded effects and a new typing rule, namely *strengthening*.

4.1 History expressions

History expressions are used to denote sets of histories. They are defined through the abstract syntax of Table 3, which extends the syntax introduced in [4].

A history expression can be empty (ε), a single access event to some resource ($\alpha(r)$) or it can be obtained either through sequential composition ($H \cdot H'$) or non-deterministic choice ($H + H'$). Moreover, we use safety framing $\varphi[H]$ for specifying that all the execution histories represented by H are under the scope of the policy φ. Additionally, $\mu h.H$ (where μ binds the free occurrences of h in H) represents recursive history expressions. Finally, we introduce guarded histories gH (where g respects the syntax of guards given in Def. 1).

The *denotational semantics* of history expressions (Table 4) maps a history expression H to a set of histories $\mathcal{H}$. The domain $\mathcal{H}$ is the lifted complete partial

$H, H' ::=$	
ε	empty
h	variable
$\alpha(r)$	access event
$H \cdot H'$	sequence
$H + H'$	choice
$\varphi[H]$	security framing
$\mu h.H$	recursion
gH	guard

Table 3. Syntax of history expressions

$$[\![\varepsilon]\!]^\sigma_\delta = \emptyset \qquad [\![h]\!]^\sigma_\delta = \delta(h)$$

$$[\![\alpha(r)]\!]^\sigma_\delta = \{\alpha(r)\} \qquad [\![H \cdot H']\!]^\sigma_\delta = [\![H]\!]^\sigma_\delta [\![H']\!]^\sigma_\delta$$

$$[\![H + H']\!]^\sigma_\delta = [\![H]\!]^\sigma_\delta \cup [\![H']\!]^\sigma_\delta \qquad [\![gH]\!]^\sigma_\delta = \begin{cases} [\![H]\!]^\sigma_\delta & \text{if } \sigma \models g \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\varphi[H]]\!]^\sigma_\delta = \varphi[[\![H]\!]^\sigma_\delta] \qquad [\![\mu h.H]\!]^\sigma_\delta = \bigcup_{n>0} f^n(!)$$

$$\text{where } f(X) = [\![H]\!]^\sigma_{\delta\{X/h\}}$$

Table 4. Semantics of history expressions

order of sets of histories [21]. Sets are ordered by (lifted) inclusion $\subseteq_\perp$ (where $\forall \mathcal{H}.\perp \subseteq_\perp \mathcal{H}$ and $\mathcal{H} \subseteq_\perp \mathcal{H}'$ whenever $\eta \in \mathcal{H} \Rightarrow \eta \in \mathcal{H}'$).

We first need a couple of standard, auxiliary notions. An environment δ binds variables $(h, h', \ldots)$ to history expressions $(H, H', \ldots)$ and a substitution σ maps variable names $(x, y, \ldots)$ to variable names or resources $(x, y, \ldots$ or $r, r', \ldots)$. Given δ and σ, the semantic function maps a history expression to a corresponding set. As expected, ε denotes the empty set under any configuration and $\alpha(r)$ denotes the singleton containing only $\alpha(r)$. The semantics of a sequential composition $H \cdot H'$, of non-deterministic choice $H + H'$ and of $\varphi[H]$ is obvious. The semantics for gH is a little more tricky. We assert gH to define two distinct sets: $[\![H]\!]^\sigma_\delta$, if σ *satisfies* g (we write $\sigma \models g$ if and only if $\mathcal{B}(g\sigma) = tt$), or $\emptyset$ otherwise. The semantics of $\mu h.H$ is the least fixed point of the function f (see [3] for further details).

$$\tau, \tau' ::= \mathbf{1} \mid \mathcal{R} \mid \tau \xrightarrow{H} \tau' \qquad\qquad \Gamma, \Gamma' ::= \emptyset \mid \Gamma; x : \tau$$

Table 5. Types and type environments

$$(\texttt{T}-\texttt{Unit})\ \Gamma, \varepsilon \vdash_g * : \mathbf{1} \qquad (\texttt{T}-\texttt{Res})\ \Gamma, \varepsilon \vdash_g r : \mathcal{R} \qquad (\texttt{T}-\texttt{Var})\ \Gamma, \varepsilon \vdash_g x : \Gamma(x)$$

$$(\texttt{T}-\texttt{Abs})\ \frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash_g e : \tau'}{\Gamma, \varepsilon \vdash_g \lambda_z x.e : \tau \xrightarrow{H} \tau'} \qquad (\texttt{T}-\texttt{Ev})\ \frac{\Gamma, H \vdash_g e : \mathcal{R}}{\Gamma, H \cdot \sum_{r \in \mathcal{R}} \alpha(r) \vdash_g \alpha(e) : \mathbf{1}}$$

$$(\texttt{T}-\texttt{App})\ \frac{\Gamma, H \vdash_g e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash_g e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash_g e\, e' : \tau'} \qquad (\texttt{T}-\texttt{Frm})\ \frac{\Gamma, H \vdash_g e : \tau}{\Gamma, \varphi[H] \vdash_g \varphi[e] : \tau}$$

$$(\texttt{T}-\texttt{If})\ \frac{\Gamma, H \vdash_{g \wedge g'} e : \tau \quad \Gamma, H \vdash_{g \wedge \neg g'} e' : \tau}{\Gamma, H \vdash_g \texttt{if}\ g'\ \texttt{then}\ e\ \texttt{else}\ e' : \tau} \qquad (\texttt{T}-\texttt{Wkn})\ \frac{\Gamma, H \vdash_g e : \tau \quad H \sqsubseteq H'}{\Gamma, H' \vdash_g e : \tau}$$

$$(\texttt{T}-\texttt{Req})\ \frac{I = \{H \mid e_{\ell'} : \tau \xrightarrow{H} \tau' \in \texttt{Srv}\rfloor_\ell \wedge H \models \varphi\}}{\Gamma, \varepsilon \vdash_g \texttt{req}_\rho\, \tau \xrightarrow{\varphi} \tau' : \tau \xrightarrow{\sum_{i \in I} H_i} \tau'} \qquad (\texttt{T}-\texttt{Str})\ \frac{\Gamma, H \vdash_g e : \tau \quad g \Rightarrow g'}{\Gamma, g'H \vdash_g e : \tau}$$

Table 6. Typing relation

4.2 Typing relation

Before introducing a type and effect system for our calculus, we define the relation $\sqsubseteq_\sigma$ as the partial order between history expressions

$$H \sqsubseteq_\sigma H' \text{ if and only if } [\![H]\!]^\sigma_\emptyset \subseteq_\perp [\![H']\!]^\sigma_\emptyset$$

We write $H \sqsubseteq H'$ when $\forall \sigma. H \sqsubseteq_\sigma H'$.

Types and type environments are introduced in Table 5. Type environments are defined in a standard way as mappings from variables to types. Types can be either base types, i.e. unit or resources, or higher-order types $\tau \xrightarrow{H} \tau'$ annotated with the history expression H.

A typing judgement (Table 6) has the form $\Gamma, H \vdash_g e : \tau$ and means that the expression e is associated with the type τ and the history expression H. The proposition g records information about the branching path collected during the typing process.

Rules ($\texttt{T}-\texttt{Unit}$, $\texttt{T}-\texttt{Res}$, $\texttt{T}-\texttt{Var}$) for $*$, resources and variables are straightforward. An event has type $\mathbf{1}$ and produces a history that is the one obtained from the evaluation of its parameter increased with the event itself ($\texttt{T}-\texttt{Ev}$). Note that, since the class of resources is finite, an event only has finitely many instantiations. An abstraction has an empty effect and a functional type carrying a

latent effect, i.e. the effect that will be produced when the function is actually applied (T−Abs). The application moves the latent effect to the actual history expression and concatenates it with the actual effects according the call-by-value semantics (T−App). Security framing extends the scope of the property φ to the effect of its target (T−Frm). The rule for conditional branching says that if we can type e and e' to the same τ generating the same effect H, then we can extend τ and H to be the type and effect of the whole expression (T−If). Moreover, in typing the sub-expressions we take into account g' and its negation, respectively. Similarly to abstractions, service requests have an empty effect (T−Req). However, the type of a request is obtained as the composition of all the types of the possible servers. In particular, the resulting latent effect is the (unguarded) non-deterministic choice among them. Observe that we only accept exact matching for input/output types (see [4] for a different composition of types). The last two rules are for weakening and strengthening. The first (T−Wkn) states that is always possible to make a generalisation of the effect inferred from an expression e. Finally, (T−Str) applies a guard g' to an effect H provided that $g \Rightarrow g'$ (if and only if $\forall\, \sigma . \sigma \models g \Rightarrow \sigma \models g'$). This rule says that we can use the information stored in g for wrapping an effect in a guarded context.

Referring to the type system introduced above, we assert the following statement to be always respected by services in Srv

$$\boxed{e_\ell : \tau \xrightarrow{H} \tau' \in \mathtt{Srv} \Longrightarrow \emptyset, \varepsilon \vdash_{tt} e_\ell : \tau \xrightarrow{H} \tau'}$$

This assumption guarantees that Srv records are always consistent with our type system.

Example 4. Let $e_{\mathtt{PWU}}$ be the service introduced in Example 1, then the following derivation is possible (dots stands for trivial or symmetrical derivations)

$$\dfrac{\dfrac{\dfrac{\vdots}{x : Card, [x \in \mathcal{C}]\, \mathtt{charge}(\mathcal{C}) \vdash_{[x \in \mathcal{C}]} \mathtt{charge}(x); \mathtt{TRUE} : Bool} \quad \vdots}{x : Card, [x \in \mathcal{C}]\, \mathtt{charge}(\mathcal{C}) \vdash_{true} \mathtt{IF} : Bool}}{\emptyset, \varepsilon \vdash_{true} e_{\mathtt{PWU}} : Card \xrightarrow{[x \in \mathcal{C}]\mathtt{charge}(\mathcal{C})} Bool}$$

where $\mathtt{charge}(\mathcal{C}) = \sum_{c \in \mathcal{C}} \mathtt{charge}(c)$ and
$\mathtt{IF} = \mathtt{if}\ [x \in \mathcal{C}]\ \mathtt{then}\, \mathtt{charge}(x); \mathtt{TRUE}\, \mathtt{else}\, \mathtt{FALSE}$.

Similarly, we observe that

$$\emptyset, \varepsilon \vdash_{true} e_{\mathtt{POL}} : Card \xrightarrow{\left[x \in \mathcal{C}'\right]\mathtt{check}(\mathcal{C}')\mathtt{charge}(\mathcal{C}')\mathtt{check}(\mathcal{C}')} Bool$$

Example 5. Consider now $e_{\mathtt{BH-S}}$ from Example 1. For the rightmost expression we can derive

$$\dfrac{x : Card, \varepsilon \vdash_{true} \mathtt{req}_\rho\, Card \xrightarrow{\varphi_{\mathtt{BH}}} Bool : \tau \qquad x : Card, \varepsilon \vdash_{true} x : Card}{x : Card, H \vdash_{true} (\mathtt{req}_\rho\, Card \xrightarrow{\varphi_{\mathtt{BH}}} Bool) x : Bool}$$

where

$$\tau = Card \xrightarrow{[x \in \mathcal{C}]\texttt{charge}(\mathcal{C})} Bool$$

Moreover we can derive

$$\dfrac{\dfrac{\Gamma, \texttt{book}(S) \vdash_{[y=\texttt{TRUE}]} \texttt{book}(S) : \mathbf{1}}{\Gamma, [y = \texttt{TRUE}]\texttt{book}(S) \vdash_{[y=\texttt{TRUE}]} \texttt{book}(S) : \mathbf{1}} \qquad \dfrac{\Gamma, \varepsilon \vdash_{[y \neq \texttt{TRUE}]} * : \mathbf{1}}{\Gamma, [y = \texttt{TRUE}]\texttt{book}(S) \vdash_{[y \neq \texttt{TRUE}]} * : \mathbf{1}}}{\Gamma, [y = \texttt{TRUE}]\texttt{book}(S) \vdash_{true} \texttt{if}\ [y = \texttt{TRUE}]\ \texttt{then}\ \texttt{book}(S)\ \texttt{else}\ * : \mathbf{1}}$$

where $\Gamma = x : Card; y : Bool$. Combining the two we obtain

$$\emptyset, \varepsilon \vdash_{true} e_{\texttt{BH}-\texttt{S}} : Card \xrightarrow{[x \in \mathcal{C}]\texttt{charge}(\mathcal{C}) \cdot [y=\texttt{TRUE}]\texttt{book}(S)} \mathbf{1}$$

Similarly, the result of typing $e_{\texttt{BN}-\texttt{S}}$ is

$$\emptyset, \varepsilon \vdash_{true} e_{\texttt{BN}-\texttt{S}} : Card \xrightarrow{\texttt{book}(S) \cdot \left[x \in \mathcal{C}'\right]\texttt{check}(\mathcal{C}')\texttt{charge}(\mathcal{C}')\texttt{check}(\mathcal{C}')[y=\texttt{FALSE}]\texttt{unbook}(S)} \mathbf{1}$$

Example 6. Consider now the term $e_{\texttt{TA}}$. The following derivation is possible

$$\dfrac{\dfrac{\vdots}{x : Card, \varepsilon \vdash_{true} \texttt{req}_{\bar{\rho}}\, Card \to \mathbf{1} : Card \xrightarrow{H+H'} \mathbf{1}} \qquad \dfrac{}{x : Card, \varepsilon \vdash_{true} x : Card}}{x : Card, H + H' \vdash_{true} (\texttt{req}_{\bar{\rho}}\, Card \to \mathbf{1})x : \mathbf{1}}$$

where

$$H = [x \in \mathcal{C}]\texttt{charge}(\mathcal{C}) \cdot [y = \texttt{TRUE}]\texttt{book}(S)$$

$$H' = \texttt{book}(S) \cdot [x \in \mathcal{C}']\,\texttt{check}(\mathcal{C}')\texttt{charge}(\mathcal{C}')\texttt{check}(\mathcal{C}') \cdot [y = \texttt{FALSE}]\texttt{unbook}(S)$$

Then, we can type the whole service to

$$\emptyset, \varepsilon \vdash_{true} e_{\texttt{TA}} : Card \xrightarrow{\varphi_{\texttt{TA}}\left[(H+H') \cdot (H+H')\right]} Rec$$

Our type and effect system produces history expressions that approximate the run-time behaviour of programs. The soundness of our approach relies on producing *safe* history expressions, i.e. any trace produced by the execution of an expression e (under any valid plan) is denoted by the history expression obtained typing e. To prove type and effect safety we need the following lemmata. Lemma 1 states that type and effect inference is closed under logical implication for guards, and Lemma 2 says that history expressions are preserved by programs execution.

Lemma 1. *If $\Gamma, H \vdash_g e : \tau$ and $g' \Rightarrow g$ then $\Gamma, H \vdash_{g'} e : \tau$*

Lemma 2. ***(Subject reduction)*** *Let $\Gamma, H \vdash_g e : \tau$ and $\eta, e \to_\pi^* \eta', e'$. For each g' such that $g' \Rightarrow g$, if $\Gamma, H' \vdash_{g'} e' : \tau$ then $\forall \sigma . \sigma \models g' \Longrightarrow \eta' [\![H']\!]_\emptyset^\sigma \subseteq \eta [\![H]\!]_\emptyset^\sigma$*

Then, the soundness of our approach is established through the following theorem, where we use η for both a history generated by the operational semantics and a history belonging to the denotational semantics of a history expression H. To compare histories of the two kinds, given $\eta \in [\![H]\!]^{\sigma}_{\emptyset}$ we write $\eta^{-\Phi}$ for η where all the security framings $\varphi[]$ have been removed, as defined below:

$$\varepsilon^{-\Phi} = \varepsilon \qquad (\alpha\eta)^{-\Phi} = \alpha(\eta^{-\Phi}) \qquad \varphi[\eta]^{-\Phi} = \eta^{-\Phi}$$

We can now state our type safety theorem.

Theorem 1. ***(Type safety)*** *If $\Gamma, H \vdash_{true} e : \tau$ and $\varepsilon, e \rightarrow^{*}_{\pi} \eta', v$, then $\forall\, \sigma.\ \exists\, \eta \in [\![H]\!]^{\sigma}_{\emptyset}$ such that $\eta' = \eta^{-\Phi}$.*

The type and effect system of [3] has no rule for strengthening like our rule (`T−Str`). The presence of this rule in our system makes it possible to discard some of the denoted traces. These traces correspond to executions that, due to the actual instantiation of formal parameters, can not take place. Consequently, our type and effect system produces more compact history expressions than those of [3]. This has several advantages, and we mention here a couple of them. First, having small history expression reduces the search space contributing to speed up verification algorithms. Secondly, it removes possible false negatives raising from traces that violate the active policy but will never be produced at run-time.

History expressions validity guarantees that no security violations can happen at run-time. We define it as follows.

Definition 2. *(Validity of History Expressions)*

$H \models \varphi$ if and only if $\forall\, \sigma$ and $\forall\, \varphi$ occurring in H, $\forall\, \eta \in [\![H]\!]^{\sigma}_{\emptyset} : \eta \models \varphi$.

We say that a history expression H is *valid* if and only if $\forall \eta \in [\![H]\!]$ and $\forall \varphi$ occurring in H then $\eta \models \varphi$. Note in passing that validity of histories is not compositional being our framework a history-dependent one. E.g. if φ says "never α" from the validity of α and of $\varphi[\beta]$ it does not follow $\alpha\beta \models \varphi$. As a matter of fact, the validity of H is established through model-checking in polynomial time [8].

5 Modular plans

In this section we introduce the main contribution of this work, i.e. how plans can be composed.

5.1 Properties of Plans

In Section 3.2 we saw how plans drive the execution of services turning service requests into actual service invocations. We can also interpret plans as a substitution strategy of request expressions. We can therefore inline e_ℓ for $\mathtt{req}_\rho\,\tau$ whenever $\pi(\rho) = \ell$ (note that recursion is not a problem here because the binding request-service defined by π is static). We call this semantics preserving process *flattening*, and *flat* a service without requests.

Definition 3. *(Flattening)*

$$* \mid_\pi = * \qquad r \mid_\pi = r \qquad x \mid_\pi = x$$

$$\alpha(e) \mid_\pi = \alpha(e \mid_\pi) \qquad (\lambda_z x.e) \mid_\pi = \lambda_z x.e \mid_\pi \qquad (e\,e') \mid_\pi = e \mid_\pi \; e' \mid_\pi$$

$$\varphi[e] \mid_\pi = \varphi[e \mid_\pi] \qquad (\mathtt{if}\; g\; \mathtt{then}\; e\; \mathtt{else}\; e') \mid_\pi = \mathtt{if}\; g\; \mathtt{then}\; e \mid_\pi \; \mathtt{else}\; e' \mid_\pi$$

$$(\mathtt{req}_\rho\,\tau \to \tau') \mid_\pi = \begin{cases} \mathtt{req}_\rho\,\tau \to \tau' & \text{if } \pi(\rho) = \bot \\ e_\ell \mid_\pi & \text{if } \pi(\rho) = \ell \end{cases}$$

Flattening an expression e with larger plans makes smaller the set of histories associated with e. This property is stated by the following theorem.

Theorem 2. *Given two plans π and π', if $\Gamma, H \vdash_g e \mid_\pi : \tau$ and $\Gamma, H' \vdash_g e \mid_{\pi;\pi'} : \tau$ then $H' \sqsubseteq H$*

To securely compose plans we must guarantee that the resulting plan preserves validity. Hence, we introduce the key concept of plan *completeness*. A plan π is complete w.r.t. a service e if it is valid and makes e flat.

Definition 4.

Given a closed term e, a plan π is said to be complete *for e if and only if*

1. *$e \mid_\pi$ is flat and*
2. *if $\emptyset, H \vdash_{true} e \mid_\pi : \tau$ then H is valid.*

Complete, modular plans for services can be composed preserving completeness, as stated below.

Lemma 3. *Given two terms e,e' and two modular plans π,π' complete for e and e', respectively, then $\pi;\pi'$ is complete for both e and e'.*

We present now a way of incrementally building services in a secure manner. We first state a theorem characterising our notion of composition. As expected, we cannot aim at a full compositionality because history validity itself is not compositional. The lemma above helps in finding conditions that permit to obtain secure services by putting together components already proved secure. We

shall then outline a way of efficiently reuse the proofs of validity already known for components, so to incrementally prove the validity of a plan for the composed service. (Recall that services are closed expressions, typable in an empty environment.)

Theorem 3. *Let π_i be modular plans complete for services e_i, and let $\emptyset, H_i \vdash_{g_i} e_i : \tau_i$, $i = 0, 1$. Then,*

- *$\pi_0; \pi_1$ is complete for both $\alpha(e_0)$ and $\mathtt{if}\ g\ \mathtt{then}\ e_0\ \mathtt{else}\ e_1$*
- *$\forall \varphi$, if $H_0 \models \varphi$ then*
  - *$\pi_0; \pi_1$ is complete for $\varphi[e_0]$*
  - *$\{\rho \mapsto \ell_0\}; \pi_0$ is complete for $\mathtt{req}_\rho\, \tau \xrightarrow{\varphi} \tau'$, where ℓ_0 hosts e_0*
- *if $\tau_0 = \tau_1 \xrightarrow{H'_0} \tau'$ and $H_0 \cdot H_1 \cdot H'_0$ is valid, then $\pi_0; \pi_1$ is complete for $e_0 e_1$*

We now sketch a procedure for checking the validity of the composition of histories that occurs when applying a service to another one (third item above). First, notice that $H_0 \cdot H_1 \cdot H'_0$ is valid if and only if (i) H_0 is valid, (ii) $\forall \varphi$ in H_1, $H_0 \cdot H_1 \models \varphi$ and (iii) $\forall \varphi'$ in H'_0, $H_0 \cdot H_1 \cdot H'_0 \models \varphi'$. Item (i) holds by hypothesis (as well as $H_1 \models \varphi$ and $H'_0 \models \varphi'$). So, it suffices to define a method for checking validity of the sequential composition of two histories $H \cdot H'$ w.r.t. a policy φ.

To check validity of $H \cdot H'$ w.r.t. φ, we look for a usage policy ψ that rejects any trace η that, once extended with any η' of H', leads to a violation of φ, i.e. $\eta\eta' \not\models \varphi$. Clearly for $H \cdot H' \models \varphi$, η should not be trace of H if $\eta \not\models \psi$.

We proceed as follows, recalling that φ is a regular language, the strings of which are rejected by a finite-state automaton A. We first build A^R, the automaton that rejects the reverse strings η^R, for η not in the language of A. Note that $\eta \not\models \varphi$ if and only if η^R belongs to the language of A^R. Now, mark all the states q of A^R reachable with η^R for all η denoted by H'. For all such q, build a new automaton A^R_q equal to A^R, except that its initial state is q. Let η^R_q belong to the language of A^R_q, for some q. This implies that there exists η denoted by H' such that $\eta^R \cdot \eta^R_q$ belongs to the language of A^R. Consequently, $\eta_q \cdot \eta \not\models \varphi$. The required policy ψ is given by the reverse of the union of the languages accepted by the automata A^R_q specified above, for all relevant q.

For example, let $H' = \mathtt{charge}(x) + \mathtt{check}(x)$ and $\varphi = \varphi_{\mathtt{BH}}$ (see Fig. 2a). Applying the procedure introduced above, we find $\psi =$ "never check(x)".

Example 7. Consider again the service network of Section 2 typed in Section 4. Let $H_{\mathtt{PWU}}$, $H_{\mathtt{POL}}$, $H_{\mathtt{BH}}$, $H_{\mathtt{BN}}$ and $H_{\mathtt{TA}}$ their latent effects.

Consider now the three service policies defined in Figure 2. It is immediate to verify that the plans $\pi_1 = \{\rho \mapsto \mathtt{PWU}\}$ and $\pi_4 = \{\rho' \mapsto \mathtt{POL}\}$ are modular and complete for Book-Here-F and Book-Now-H, respectively.

We try to obtain the service Travel Agency by sequentially composing the above two components (recall that sequentialization is a simple form of application). To guarantee security we have to prove the validity of $H_{\mathtt{BH-F}} \cdot H_{\mathtt{BN-H}}$ (or vice versa). However, state 2 of $\varphi_{\mathtt{BN}}$ is reachable through a prefix of a history in the semantics of $H_{\mathtt{BH-F}}$, e.g. $\mathtt{charge}(x)\mathtt{book}(F)$ originated when $x \in \mathcal{C}$. Similarly, for $H_{\mathtt{BN-F}} \cdot H_{\mathtt{BH-H}}$.

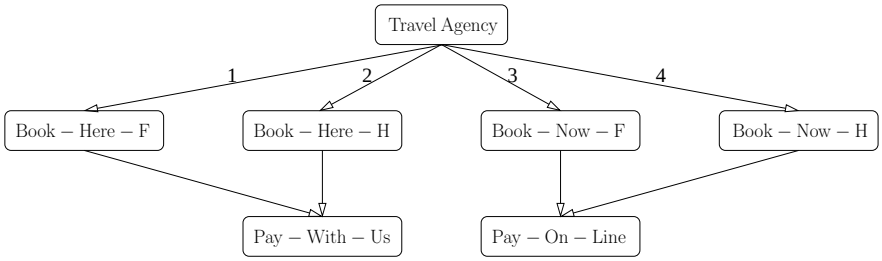


Fig. 3. A travel booking network

Instead, if we consider $H_{\texttt{BH}-\texttt{F}} \cdot H_{\texttt{BH}-\texttt{H}}$ no violations can occur (neither service performs a check). Now, the composed service is secure because it is compliant with the policy $\varphi_{\texttt{TA}}$.

Similarly for the three cases $H_{\texttt{BH}-\texttt{H}} \cdot H_{\texttt{BH}-\texttt{F}}$, $H_{\texttt{BN}-\texttt{F}} \cdot H_{\texttt{BN}-\texttt{H}}$ and $H_{\texttt{BN}-\texttt{H}} \cdot H_{\texttt{BN}-\texttt{F}}$.

Finally, although $H_{\texttt{BH}-\texttt{F}} \cdot H_{\texttt{BH}-\texttt{F}}$ (and the analogous three compositions) can be proved secure, it does not obey $\varphi_{\texttt{TA}}$.

6 Conclusion and related work

In this paper we presented a framework for secure service composition. Services are implemented in λ^{req} [3], a service-oriented version of the call-by-value λ-calculus. Security properties are defined through usage automata and locally applied to pieces of code using a proper operator, namely security framing. A type and effect system extracts a history expression from a service. Since it considers service assertions on resources, our type and effect system can also deal with open service networks, i.e. networks where one or more services are unspecified. History expressions are safe approximation of a program, i.e. they denote every possible sequence of security relevant actions fired at run-time. Moreover, history expressions can be model-checked against local policies. The result of this process is a policy compliance proof, if it exists, guaranteeing that no security violation will happen at run-time. We used this technique for generating viable composition plans. Plans produced in this way drive the service execution respecting the security constraints. Additionally, we showed a composition strategy for plans. If two or more plans satisfy the necessary conditions they can be composed in a wider plan. Then, we defined a simple procedure for verifying the validity of composed plans.

This work is a first step toward a bottom-up composition strategy for service network. Many further directions can be investigated. An interesting result could be obtained by exploiting the correspondence between history expressions and basic process algebras (BPAs [9]). Using an approach similar to [11], we could define a symbolic transition system (STS). STSs offer many interesting properties and allow for a fine-grained analysis of security properties.

Another possible improvement consists in extending out programming model introducing a concurrent composition of services. This scenario poses several issues increasing the complexity of our framework. Mainly, the well known state explosion problem could make the model checking step practically infeasible.

To the best of our knowledge, there are no proposals for the secure composition of services within a linguistic framework. Below we briefly survey on a few related work.

In [12] the authors present a framework for contract-based creation of choreographies. Roughly, they use a contract system for finding a match between contracts and choreographies. In this way they verify whether a given contract, declared by a service joining the network, is consistent with the current choreography. However, this framework exploits a global knowledge about the network structure while our model also deals with open networks.

Castagna et al. [14] use a variant of CCS with internal and external choice operators [18] for defining service contracts. A subcontract relation guarantees that the choreography always respects the contracts of both client and service. Nevertheless, this approach assumes that clients always know the request contract and it is focused on finding a satisfactory composition with some available service. Since the composition depends on the possible instantiations of a service, such a contract could be unavailable during the analysis phase. In this sense, our technique overcomes this limitation by producing valid plans independently from the actual instantiation of resources.

Busi et al. [13] propose an analysis of service orchestration and choreography. In their work, the validity of the service orchestration is a consequence of its conformance with respect to the intended choreography. The main difference with our work consists in the approach to choreography. Indeed, they start from a pre-defined choreography and verify the validity of an orchestration. Instead, our approach aims at checking partial service composition without relying on any global orchestrator.

In [16] a framework for the synthesis of orchestrators is presented. This technique consists in automatically producing an orchestrator guaranteeing that the service composition respects the desired security policy. This approach defines a composition that complies with the policy of a client. Since our method produces partial compositions that respect all the involved policies (of both clients and servers), it seems to be more general. However, as [16] generates dynamic orchestrators, the two systems work under completely different assumptions.

References

1. Martín Abadi and Cédric Fournet. Access control based on execution history. In *NDSS*, 2003.
2. Massimo Bartoletti, Gabriele Costa, Pierpaolo Degano, Fabio Martinelli, and Roberto Zunino. Securing Java with local policies. *Journal of Object Technology (JOT)*, 2008.
3. Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. History-based access control with local policies. In *FoSSaCS*, pages 316–332, 2005.

4. Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Planning and verifying service composition. *Journal of Computer Security (JCS)*, 17(5):799–837, 2009. (Abridged version In *Proc. 18th Computer Security Foundations Workshop (CSFW)*, 2005).
5. Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Secure service orchestration. In *FOSAD*, pages 24–74, 2007.
6. Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Types and effects for resource usage analysis. In *FoSSaCS*, pages 32–47, 2007.
7. Massimo Bartoletti, Pierpaolo Degano, Gian-Luigi Ferrari, and Roberto Zunino. Local policies for resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 31(6):1–43, 2009.
8. Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Model checking usage policies. *Trustworthy Global Computing: 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, pages 19–35, 2009.
9. Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.
10. Frédéric Besson, Thomas P. Jensen, and Daniel Le Métayer. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217–250, 2001.
11. Michele Boreale and Rocco De Nicola. A symbolic semantics for the pi-calculus. *Inf. Comput.*, 126(1):34–52, 1996.
12. Mario Bravetti, Ivan Lanese, and Gianluigi Zavattaro. Contract-driven implementation of choreographies. In *TGC*, pages 1–18, 2008.
13. Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration: A synergic approach for system design. In *ICSOC*, pages 228–240, 2005.
14. Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5), 2009.
15. Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.
16. Fabio Martinelli and Ilaria Matteucci. Synthesis of web services orchestrators in a timed setting. In *WS-FM*, pages 124–138, 2007.
17. Fabio Martinelli and Ilaria Matteucci. Synthesis of local controller programs for enforcing global security properties. In *ARES*, pages 1120–1127, 2008.
18. Rocco De Nicola and Matthew Hennessy. Ccs without tau's. In *TAPSOFT, Vol.1*, pages 138–152, 1987.
19. Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
20. Christian Skalka and Scott F. Smith. History effects and verification. In *APLAS*, pages 107–128, 2004.
21. Glynn Winskel. *The formal semantics of programming languages*. MIT Press, 1993.