Planning and Verifying Service Composition

Massimo Bartoletti Pierpaolo Degano Gian Luigi Ferrari
Dipartimento di Informatica, Università di Pisa, Italy
{bartolet, degano, giangi}@di.unipi.it

Abstract

A static approach is proposed to study secure composition of services. We extend the λ-calculus with primitives for selecting and invoking services that respect given security requirements. Security-critical code is enclosed in policy framings with a possibly nested, local scope. Policy framings enforce safety and liveness properties. The actual run-time behaviour of services is over-approximated by a type and effect system. Types are standard, and effects include the actions with possible security concerns — as well as information about which services may be invoked at run-time. An approximation is model-checked to verify policy framings within their scopes. This allows for removing any run-time execution monitor, and for determining the plans driving the selection of those services that match the security requirements on demand.

1 Introduction

Service-oriented computing (SOC) is an emerging paradigm to design distributed applications [41, 40, 22]. In this paradigm, applications are built by assembling together independent computational units, called *services*. A service is a standalone component distributed over a network, and made available through standard interaction mechanisms. An important aspect is that services are *open*, in that they are built with little or no knowledge about their operating environment, their clients, and further services therein invoked. Composition of services may require peculiar mechanisms to handle complex interaction patterns (e.g. to implement transactions), while enforcing non-functional requirements on the system behaviour (e.g. security and service level agreement). Web Services [1, 45, 49] built upon XML technologies are possibly the most illustrative and well developed example of the SOC paradigm. Indeed, a variety of XML-based technologies already exists for describing, discovering and invoking web services [16, 18, 14, 50]. There are also several standards for defining and enforcing non-functional requirements of services, e.g. WS-Security [3], WS-Trust [2] and WS-Policy [48] among the others. *Orchestration* of services consists of their composition and coordination. Languages for that have been recently proposed, e.g. WS-BPEL [14, 34].

Service composition heavily depends on which information about a service is made public, on how to choose those services that match the user's requirements, and on their actual run-time behaviour. Security makes service composition even harder. Services may be offered by different providers, which only partially

trust each other. On the one hand, providers have to guarantee the delivered service to respect a given security policy, in any interaction with the operational environment, and regardless of who actually called the service. On the other hand, clients may want to protect their sensible data from the services invoked.

In this paper, we tackle the problem of modelling composition of services in the presence of security constraints. Our main result is a semantic-based method to *plan* which services an application has to choose in order to complete the original task, while guaranteeing security.

Our first technical contribution is on a foundational calculus for service orchestration, called λ^{req}. We model services as expressions of a typed extension of the λ-calculus with primitive constructs to describe and compose services, being selected to enforce the given requirements. Indeed, our selection mechanism matches (suitable abstractions of) service *behaviour*, rather than syntactic signatures, as it happens in the standard discovery mechanisms.

We are interested in enforcing *safety* and *liveness* properties over the abstract behaviour of services. This kind of properties have shown effective to reason about security. For example, history-based access control can be handled as safety properties [4, 44], while liveness properties can be exploited to formalize denial-of-service and brute-force attacks on cryptographic keys [26]. Here, we assume as given a set of primitive *access events*, that abstract from activities with possible security concerns. The security policies are *regular* properties of *execution histories* (i.e. sequences of access events) and have a possibly nested, local scope. Given an expression e, a *safety framing* $\varphi[e]$ enforces the policy φ at each step of the execution of e. A *liveness framing* $\psi\langle e\rangle$ prescribes that the evaluation of e must eventually respect the policy ψ. Our results are independent of the logic chosen for expressing regular properties of sequences, so we do not fix any logic here.

We shall exploit a static analysis technique to determine plans that drive service executions enjoying both safety and liveness properties. Note that, while safety properties can be enforced by an execution monitor, liveness properties cannot [42]. Also, liveness cannot be reduced to safety in general. Moreover, here we cannot predict any bound on the time a service needs to be completed, hence e.g. bounded liveness – a safety property – is inadequate. The considerations above further support the use of static techniques.

We introduce a type and effect system [27, 39, 46] for our calculus. Types are standard, while effects, called *history expressions*, represent all the possible behaviour of services. A service is modelled as a λ^{req} expressions with a functional type of the form $\tau_1 \xrightarrow{H} \tau_2$. Intuitively, when supplied with an argument of type τ_1, the service evaluates to a value of type τ_2, and the side effect of the invocation is an execution history belonging to the history expression H. A service request is modelled by an expression $\mathtt{req}_r\tau$, where r uniquely identifies the request, and τ is the type of the requested service, including the safety and liveness properties on demand. The safety constraint says how the caller protects itself from the service. Instead, the liveness constraint can be seen as the duties the invoked service must fulfill.

For simplicity, here we assume that services are published in a global trusted repository, i.e. a set of typed expressions $\{e_1 : \tau_1 \xrightarrow{H_1} \tau_1' \cdots e_k : \tau_k \xrightarrow{H_k} \tau_k'\}$. Types are the semantic information made visible about services. Operationally, a service request $\mathtt{req}_r\tau$ results in a sort of "call-by-contract": the repository

is searched for a service with a functional type matching the request type τ; additionally, its effect H should respect the safety and liveness constraints in τ. The effect of a service invocation $\mathtt{req}_r\tau$ has the form $\{r[\ell_1] \rhd H_1 \cdots r[\ell_k] \rhd H_k\}$, where $r[\ell_i]$ resolves the request r with the service e_i in the repository. We say that H_i is *valid* when it respects the safety and liveness constraints in τ, as well as all the security framings within H_i itself.

The effect H of a service composition e is obtained by suitably assembling the effects of the component services, and of those services they may invoke in a nested fashion. Note however that composing the effects of the selected services, yet valid, may result in a *non-valid* overall effect. This is because the effect of selecting a given service for a request is not confined to the execution of that service, but it spans over the whole execution. The validity of the effect H of e depends thus on the plan that selects a service for each request. It is convenient to lift all the service choices $r[\ell]$ to the top-level of H, collecting them in a set π, called *plan*. We define then a semantic-preserving transformation that results in effects of the form $\{\pi_1 \rhd H_1 \cdots \pi_n \rhd H_n\}$, where each H_i is free of further choices. Its intuitive meaning is that, under the plan π_i, the effect of the overall service composition e is H_i. If some H_i is valid, then the plan π_i will safely drive the execution of e, without resorting to any run-time monitor, and guaranteeing all the safety and liveness properties required.

Validity of history expressions is still to be ascertained. We do that by model checking (a suitable version of) Basic Process Algebras (BPAs) with finite state automata. A history expressions H is naturally rendered as a BPA process, while a finite state automaton models the validity of H. Because of the possible nesting of framings, validity of history expressions is a non-regular property, so standard model checking techniques cannot be directly applied. We transform then history expressions so to make model checking feasible through specially-tailored finite state automata.

All the proofs of our results are available as a Technical Report [5]. The present paper is an extended version of [6].

2 Motivating examples

To illustrate our approach, consider a simple certification service that wants to attest a contract between two external parties, while enforcing its own privacy policy. Since specifying a privacy policy can be difficult and error-prone, this task is delegated to a trusted *policy provider*.

Policy providers return a safety policy φ in the form of a closure $\lambda x.\, \varphi[x]$ (indeed, policies are not first class objects in our model). The policy φ has to be enforced in the subsequent execution of the certification service, i.e. $(\lambda x.\, \varphi[x])e$ will evolve to $\varphi[e]$. This paradigm can be seen as a form of *dynamic sandboxing*. The published interface of policy providers guarantees that the distinguished event α_p (modelling the return of a policy) will eventually occur.

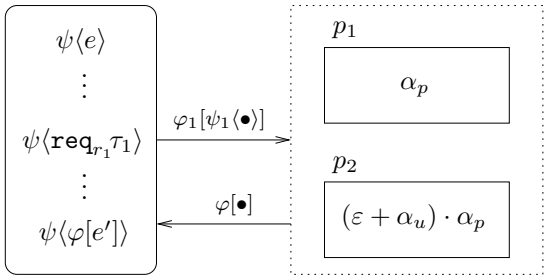
To assert its willingness to provide clients with a signed and non-repudiable copy of the contract, the certification service encloses its code into a liveness framing $\psi\langle\cdots\rangle$. The property ψ states that eventually there will be a signature (modelled by an event α_{sgn}) with no subsequent revocation (the event α_{rvk}).

The certification service requests a policy provider through the expression:

$$\mathtt{req}_{r_1}\tau_1 \qquad\qquad \text{where } \tau_1 = \tau \to (\tau \xrightarrow{\varphi_1[\psi_1\langle\bullet\rangle]} \tau)$$

(here, τ is a base type, whose specification is irrelevant). The request asks for a service that takes an argument of type τ, and returns a policy as a closure of type $\tau \to \tau$. The annotation $\varphi_1[\psi_1\langle\bullet\rangle]$ in τ_1 indicates that, when evaluated, the selected service must respect the safety policy φ_1 and the liveness policy ψ_1. The policy ψ_1 requires that eventually α_p occurs, while φ_1 says that the service cannot visit untrusted sites, modelled by the event α_u.

The leftmost part in the following diagram highlights the evolution of the certification service. The rightmost part shows the two policy providers p_1 and p_2 available in the repository. For simplicity, the boxes only represent the *effect* of the evaluation of the service; the arrows display the service invocation and the delivered safety policy.



The service p_1 exposes a history expression α_p, to mean that p_1 will generate exactly the event α_p and no other security-relevant operations. The service p_2 possibly connects to an untrusted site (thus generating the event α_u), and then provides the client with a privacy policy. This behaviour is modelled by the history expression $(\varepsilon + \alpha_u)\cdot\alpha_p$, where ε stands for the empty history, $\cdot$ for concatenation of histories, and $+$ for non-deterministic choice. The effect associated with the request r_1 is then a *planned selection* of the form:

$$\{r_1[p_1] \rhd \varphi_1[\psi_1\langle\alpha_p\rangle],\ r_1[p_2] \rhd \varphi_1[\psi_1\langle(\varepsilon + \alpha_u)\cdot\alpha_p\rangle\}$$

In the first element, the plan $r_1[p_1]$ indicates that the request r_1 is served by the provider p_1. In this case, the resulting history expression is $\varphi_1[\psi_1\langle\alpha_p\rangle]$. Similarly, when the plan is $r_1[p_2]$. Note that p_1 successfully serves the request r_1, because α_p satisfies both φ_1 and ψ_1. Instead, p_2 can generate the histories α_p and $\alpha_u\alpha_p$. Since the second history does not respect the safety constraint φ_1, then p_2 will be rejected by our static machinery.

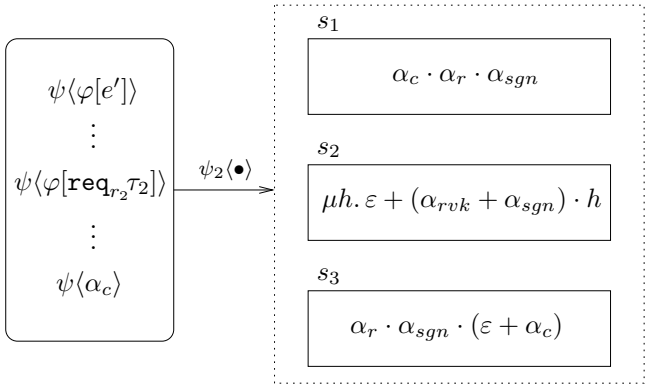
Assume now that the privacy policy φ delivered by p_1 states that connecting to the network (α_c) is prevented after a read (α_r) of local data. The diagram below depicts the invocation of the contracting parties, modelled by the request

$$\mathtt{req}_{r_2}\tau_2 \qquad\qquad \text{where } \tau_2 = \tau \xrightarrow{\psi_2\langle\bullet\rangle} \tau$$

The required guarantee is expressed by the liveness property ψ_2, saying that the certification service will eventually receive back a signed contract (note that

here a bounded liveness in place of ψ_2 would work as well, just because the code of services is visible; however, this is *not* the case in general).

There are three contracting parties. The service s_1 opens a network connection (α_c), reads the contract file (α_r) and then signs it (α_{sgn}); the service s_2 is a loop of either action of revoking or signing (recursion is written through the μ operator); the service s_3 reads the contract file (α_r), signs it (α_{sgn}), and then can possibly open a network connection (α_c).



The planned selection associated with r_2 is then:

$$\begin{aligned}&\{r_2[s_1] \rhd \psi_2\langle\alpha_c \cdot \alpha_r \cdot \alpha_{sgn}\rangle,\\ &\ r_2[s_2] \rhd \psi_2\langle\mu h.(\varepsilon + \alpha_{rvk} + \alpha_{sgn}) \cdot h\rangle,\\ &\ r_2[s_3] \rhd \psi_2\langle\alpha_r \cdot \alpha_{sgn} \cdot (\varepsilon + \alpha_c)\rangle\}\end{aligned}$$

The history expressions exposed by s_1 and s_3 clearly satisfy the requested agreement ψ_2, while that of s_2 does not, because, e.g. the history $(\alpha_{rvk})^n$ is possible, for all n. Therefore, the service request can only be served successfully by either s_1 or s_3. After completion of the service, the safety framing $\varphi[\cdots]$ is left, so the certification service can eventually connect to the network.

Considering only the services matching the requests (i.e. p_1 for r_1 and s_1, s_3 for r_2) we associate the following overall history expression H with the whole certification service:

$$\begin{aligned}H \ = \ &\psi\langle\{r_1[p_1] \rhd \varphi_1[\psi_1\langle\alpha_p\rangle]\} \cdot\\ &\varphi[\{r_2[s_1] \rhd \psi_2\langle\alpha_c \cdot \alpha_r \cdot \alpha_{sgn}\rangle,\ r_2[s_3] \rhd \psi_2\langle\alpha_r \cdot \alpha_{sgn} \cdot (\varepsilon + \alpha_c)\rangle\}] \cdot \alpha_c\rangle\end{aligned}$$

To determine the correct service compositions, we first "linearize" H, by moving all the associations of requests with services at the top-level. Here we obtain:

$$\begin{aligned}\{r_1[p_1] \mid r_2[s_1] \ &\rhd\ \psi\langle\varphi_1[\psi_1\langle\alpha_p\rangle]\ \cdot \varphi[\psi_2\langle\alpha_c \cdot \alpha_r \cdot \alpha_{sgn}\rangle] \cdot \alpha_c\rangle,\\ r_1[p_1] \mid r_2[s_3] \ &\rhd\ \psi\langle\varphi_1[\psi_1\langle\alpha_p\rangle]\ \cdot \varphi[\psi_2\langle\alpha_r \cdot \alpha_{sgn} \cdot (\varepsilon + \alpha_c)\rangle] \cdot \alpha_c\rangle\}\end{aligned}$$

where $\mid$ composes independent plans. The first element is valid, because choosing p_1 for r_1 and s_1 for r_2 drives a successful computation. Indeed, the liveness framing ψ is satisfied by the signature α_{sgn} which is never revoked afterwards. Also, the safety framing φ is obeyed, since within its scope there is no connection

α_c after a read α_r. Instead, the history expression associated with the plan $r_1[p_1] \mid r_2[s_3]$ is *not* valid, because α_c may occur after an α_r within the scope of φ. This verification phase is mechanizable and determines all and only the valid elements.

In the next sections we shall describe a way of extracting the relevant information from services (through a type and effect system) and for verifying when their composition is valid (by model checking), so providing us at static time with a winning planning strategy – in our example the plan $r_1[p_1] \mid r_2[s_1]$. The paramount point is that we can execute an expression and resolve its requests according to a winning plan, with no run-time monitoring and with the guarantee that the overall behaviour will always comply with the safety and liveness policies on demand.

3 Programming model

To study secure service composition in a pure framework, we consider λ^{req}, a call-by-value λ-calculus enriched with local security policies and service requests. An *access event* $\alpha \in \mathsf{Ev}$ abstracts from a security critical operation (e.g. writing a file, opening a socket connection). A *history* η is a sequence of access events. A *security policy* $\varphi \in \mathsf{Pol}$ is a regular property of histories. A *safety framing* $\varphi[e]$ enforces the policy φ at each step of the evaluation of e. A *liveness framing* $\psi\langle e\rangle$ requires that the policy ψ will eventually be satisfied while evaluating e. Services $e : \tau$ are typed λ^{req} expressions, collected in a trusted, finite, and global repository Srv, abstracting from the distributed service directories (e.g. UDDI [50]). The types τ are annotated with *history expressions* that over-approximate the possible run-time histories. E.g., when a function with type $\tau \xrightarrow{H} \tau'$ is applied to a value, it will generate one of the histories denoted by H. The repository Srv guarantees that H represents all the possible histories of e.

A *service request* has the form $\mathtt{req}_r\tau$. The label r uniquely identifies the request in an expression, and the request type τ is defined as:

$$\tau \quad ::= \quad \mathit{unit} \mid \tau \xrightarrow{(\varphi,\psi)} \tau$$

where *unit* is the standard singleton type. The annotations (φ, ψ) on the arrow are the safety/liveness constraints imposed on the service. Operationally, this drives a search in the repository Srv for a service with a functional type τ' "compatible" with τ (defined later on in Section 5) and such that τ' respects the constraints imposed by τ. For clarity, we omit the null constraint (tt, tt), we write $\varphi[\bullet]$ for (φ, tt), $\psi\langle\bullet\rangle$ for (tt, ψ) and $\varphi[\psi\langle\bullet\rangle]$ in the general case. Intuitively, a constraint can be seen as a context that wraps the behaviour H of a service, obtaining $\varphi[\psi\langle H\rangle]$, and meaning that the histories denoted by H must satisfy "always φ" and "eventually ψ".

We put some restrictions on the types a programmer can use in a request. First, only functional types are allowed: this models services being considered as remote procedures. Higher-order return values are allowed: if the type of a returned value is functional, then the request can be seen as a code download, e.g. an applet. Second, no constraints should be imposed over the type τ_0 of a request type $\tau_0 \xrightarrow{(\varphi,\psi)} \tau_1$, i.e. in τ_0 there are no annotations. This is because the constraints on the selected service should not affect its argument.

3.1 Syntax

The syntax of our calculus follows. We assume as given the languages for (regular) policies φ, ψ and for guards b. We omit their definition here, as they are not relevant for the subsequent technical development. To enhance readability, our calculus comprises conditional expressions and named abstractions (the variable z in $e' = \lambda_z x.\, e$ stands for e' itself within e).

Expressions

e, e'	$::=$	
	$*$	unit
	x	variable
	α	access event
	$\texttt{if}\, b\, \texttt{then}\, e\, \texttt{else}\, e$	conditional
	$\lambda_z x.\, e$	abstraction
	$e\, e'$	application
	$\varphi[e]$	safety framing
	$\psi\langle e \rangle$	liveness framing
	$\texttt{req}_r \tau$	service request

The values v of our calculus are the variables, the abstractions, the requests, and the distinguished element $*$. The following abbreviation is standard: $e; e' = (\lambda.\, e')\, e$. Without loss of generality, we assume that each framing has an opening event, e.g. for all $\varphi[e]$, the expression e has the form $\alpha; e'$, for some α and e'. The opening event can be dummy, with no influence on security.

3.2 Operational semantics

The evaluation of a λ^{req} expression requires to check all the policies within their framings, and to serve requests. In this section, we do not have yet an operational mechanism to resolve requests, and so we resort to an *oracle*. We assume that, upon a request $\texttt{req}_r \tau$, the oracle selects from Srv a service (if any) that respects the type and the constraints expressed by τ. The oracle guarantees that the execution of the selected service satisfies the safety/liveness policies in τ (although, e.g. it might violate a policy that was already active before the request). In the following sections, we will develop a static machinery that will enable us to efficiently implement the oracle, guaranteeing that an expression will never go wrong. Consequently, we will safely discard all the security framings, and avoid to check them dynamically.

Here, we are not interested in characterizing what the oracle guarantees, and we simply model it as a set of service choices, called *plan*. A plan formalises how a request is resolved into an actual service, and takes the form of a *finite* injective mapping from request labels to services. Plans have the following syntax:

Plans

π, π'	$::=$	
	0	empty
	$r[\ell]$	service choice
	$\pi \mid \pi'$	composition

The empty plan 0 has no choices; the plan $r[\ell]$ associates the service $e_\ell : \tau_\ell$ with the request labelled r. The composition operator $\mid$ on plans is associative, commutative and idempotent, and its identity is the empty plan 0.

Now we define the behaviour of expressions through the following small-step operational semantics. The configurations are pairs η, e and a transition $\eta, e \rightarrow_\pi \eta', e'$ means that, starting from a history η, the plan π allows the expression e to evolve to e' and to extend η to η'. An expression is initially evaluated starting from the empty history ε. We write $\eta \models \varphi$ when the history η obeys the (safety/liveness) policy φ. We assume as given a total function $\mathcal{B}$ that evaluates the guards in conditionals.

Operational semantics of λ^{req}

$$\frac{\eta, e_1 \rightarrow_\pi \eta', e_1'}{\eta, e_1 e_2 \rightarrow_\pi \eta', e_1' e_2} \quad \text{(E-App1)} \qquad \frac{\eta, e_2 \rightarrow_\pi \eta', e_2'}{\eta, v e_2 \rightarrow_\pi \eta', v e_2'} \quad \text{(E-App2)}$$

$$\eta, (\lambda_z x.\, e) v \rightarrow_\pi \eta, e\{v/x, \lambda_z x.\, e/z\} \quad \text{(E-AbsApp)}$$

$$\eta, \alpha \rightarrow_\pi \eta\alpha, * \quad \text{(E-Ev)} \qquad \eta, \mathtt{if}\ b\ \mathtt{then}\ e_{tt}\ \mathtt{else}\ e_{ff} \rightarrow_\pi \eta, e_{\mathcal{B}(b)} \quad \text{(E-If)}$$

$$\frac{\eta, e \rightarrow_\pi \eta', e' \quad \eta' \models \varphi}{\eta, \varphi[e] \rightarrow_\pi \eta', \varphi[e']} \quad \text{(E-SF1)} \qquad \frac{\eta \models \varphi}{\eta, \varphi[v] \rightarrow_\pi \eta, v} \quad \text{(E-SF2)}$$

$$\frac{\eta, e \rightarrow_\pi \eta', e' \quad \eta \not\models \psi}{\eta, \psi\langle e \rangle \rightarrow_\pi \eta', \psi\langle e' \rangle} \quad \text{(E-LF1)} \qquad \frac{\eta \models \psi}{\eta, \psi\langle e \rangle \rightarrow_\pi \eta, e} \quad \text{(E-LF2)}$$

$$\frac{e_\ell : \tau_\ell \in \mathsf{Srv} \quad \pi = r[\ell] \mid \pi'}{\eta, (\mathtt{req}_r \tau) v \rightarrow_\pi \eta, e_\ell\, v} \quad \text{(E-Req)}$$

The first two rules implement call-by-value evaluation; as usual, functions are not reduced within their bodies. The third rule implements β-reduction. Notice that the whole function body $\lambda_z x.\, e$ replaces the self variable z after the substitution, so giving an explicit copy-rule semantics to recursive functions. The evaluation of an event α consists in appending α to the current history, and producing the no-operation value $*$. A conditional $\mathtt{if}\ b\ \mathtt{then}\ e_{tt}\ \mathtt{else}\ e_{ff}$ evaluates to e_{tt} (resp. e_{ff}) if b evaluates to true (resp. false).

To evaluate a safety framing $\varphi[e]$, we must consider two cases. If, starting from the current history η, e may evolve to e' and extend the history to η', then the whole framing $\varphi[e]$ may evolve to $\varphi[e']$, provided that η' satisfies φ. Otherwise, if e is a value and the current history satisfies φ, then the scope of the framing is left. In both cases, as soon as a history is found not to respect φ, the evaluation gets stuck, to model a security exception. For simplicity, we do not model here exceptions and exception handling, but extending our language in this direction is straightforward.

Within a liveness framing $\psi\langle e \rangle$, the expression e evolves as long as the property ψ is not satisfied. As soon as the current history obeys ψ, the framing is discarded. Note that we cannot operationally guarantee that ψ will eventually

hold: indeed, this is a liveness property, so it cannot be enforced by execution monitoring alone [42]. It is then particularly relevant that our static analysis is able to discover whether a liveness framing will be eventually discarded or not.

The rule for service invocation enquires the oracle to select from Srv a service that respects the types and the required constraints. If no such service exists, the execution gets stuck.

4 History expressions

To statically predict the histories generated by programs at run-time, as well as the scopes of policies, we introduce *history expressions* with the following abstract syntax. History expressions are a sort of context-free grammars, and include the empty history ε, access events α, sequencing $H \cdot H'$, non-deterministic choice $H + H'$, safety and liveness framings $\varphi[H]$ and $\psi\langle H\rangle$, recursion $\mu h.H$ (μ binds the occurrences of the variable h in H), and planned selection.

History Expressions

H, H'	$::=$	
	ε	empty
	h	variable
	α	access event
	$H \cdot H'$	sequence
	$H + H'$	choice
	$\varphi[H]$	safety framing
	$\psi\langle H\rangle$	liveness framing
	$\mu h.H$	recursion
	$\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$	planned selection

Safety and liveness framings are the abstract counterparts of the analogous constructs in λ^{req}. Given a plan π, a planned selection $\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$ chooses those H_i such that π includes π_i. Intuitively, the history expression $H = \{r[\ell_1] \triangleright H_1, r[\ell_2] \triangleright H_2\}$ is associated with a request r that can be resolved into either e_{ℓ_1} or e_{ℓ_2}. The histories denoted by H depend on the given plan π: if π chooses ℓ_1 (resp. ℓ_2) for r, then H denotes one of the histories represented by H_1 (resp. H_2); otherwise, H denotes no histories. Due to our assumption on the form of plans, we always discard the components $\pi_i \triangleright H_i$ from a planned selection, if π_i turns out to be non-injective.

We assume that the operator $\cdot$ has precedence over $+$, that in turn has precedence over μ. We say that a history expression H is *closed* when it has no free variables, i.e. $fv(H) = \emptyset$, where free variables are defined as expected, e.g.:

$$fv(h) = \{h\} \qquad fv(\mu h.\, H) = fv(H) \setminus \{h\}$$

$$fv(\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}) = \textstyle\bigcup_{i \in 1..k} fv(H_i)$$

To define the semantics of history expressions, we enrich histories with a set Frm of special *framing events*, parametrized by policies in Pol. The events $[_\varphi$ and $]_\varphi$ denote the opening and closing of a safety framing $\varphi[\cdots]$, while $\langle_\psi$ and $\rangle_\psi$ play

the same role for liveness framings. Formally, a *history* η is a sequence $\beta_1 \cdots \beta_k$ where $\beta_i \in \mathsf{Ev} \cup \mathsf{Frm}$, $\mathsf{Frm} = \{\, [_\varphi,]_\varphi, \langle_\varphi, \rangle_\varphi \mid \varphi \in \mathsf{Pol} \,\}$, and $\mathsf{Ev} \cap \mathsf{Frm} = \emptyset$.

For example, a history $\alpha[_\varphi \alpha']_\varphi$ represents a computation that (i) generates an event α, (ii) enters the scope of the safety framing $\varphi[\cdots]$, (iii) generates α' within the scope of φ, and (iv) leaves the scope of φ. Note that histories with events in Ev only were enough to give the operational semantics of our calculus, because the role of framing events is played by framed expressions.

Hereafter, a history may end with the truncation marker !. The history $\eta!$ represents a prefix of a possibly non-terminating computation that generates the sequence of events η. We assume that histories are undistinguishable after truncation, i.e. $\eta!$ followed by η' equals to $\eta!$. A history η is *balanced* when either η is empty, or η is an access event, or $\eta =!$, or $\eta = [_\varphi \eta']_\varphi$ with η' balanced, or $\langle_\psi \eta' \rangle_\psi$ with η' balanced, or $\eta = \eta'\eta''$ with both η' and η'' balanced. A history is *well-formed* when it is the prefix of a balanced history. For example, $\alpha[_\varphi \alpha'[_{\varphi'} \alpha'']_{\varphi'}]_\varphi$ is balanced, $\alpha[_\varphi \alpha'$ is well-formed but not balanced, and $\alpha[_\varphi \alpha'[_{\varphi'} \alpha'']_\varphi$ is not well-formed. Hereafter, we will only deal with well-formed histories, because they model exactly the histories that will show up when executing λ^{req} expressions. Let $\mathcal{H}$ range over sets of balanced histories. We define $\mathcal{H}\mathcal{H}'$ as the set of histories $\{\, \eta\eta' \mid \eta \in \mathcal{H}, \eta' \in \mathcal{H}' \,\}$, $\varphi[\mathcal{H}]$ as $\{\, [_\varphi \eta]_\varphi \mid \eta \in \mathcal{H} \,\}$, and $\psi\langle\mathcal{H}\rangle$ as the set $\{\, \langle_\psi \eta \rangle_\psi \mid \eta \in \mathcal{H} \,\}$.

The *denotational semantics* of history expressions is defined over the lifted cpo of sets of balanced histories [51], ordered by (lifted) set inclusion $\subseteq_\perp$, where, for all balanced histories $\mathcal{H}, \mathcal{H}'$, $\perp \subseteq_\perp \mathcal{H}$, and $\mathcal{H} \subseteq_\perp \mathcal{H}'$ if $\mathcal{H} \subseteq \mathcal{H}'$. The least upper bound between two elements of the cpo is standard set union $\cup$, assuming that $\perp \cup \mathcal{H} = \mathcal{H}$. The *strict* least upper bound is denoted by $\cup_\perp$, and it is such that $\perp \cup_\perp \mathcal{H} = \perp$. We stipulate that concatenation of sets of histories is strict, i.e. it returns $\perp$ whenever one of its arguments is such.

The semantics $[\![H]\!]^\pi_\rho$ of a history expression H (in an environment ρ and under a plan π) is defined by the following rules. The environment ρ maps variables to sets of balanced histories. Hereafter, we feel free to omit curly braces when writing singleton sets, and we omit the empty environment for closed expressions.

Semantics of history expressions

$$[\![\varepsilon]\!]^\pi_\rho = \varepsilon \qquad [\![H \cdot H']\!]^\pi_\rho = [\![H]\!]^\pi_\rho \, [\![H']\!]^\pi_\rho \qquad [\![H + H']\!]^\pi_\rho = [\![H]\!]^\pi_\rho \cup_\perp [\![H']\!]^\pi_\rho$$

$$[\![\alpha]\!]^\pi_\rho = \alpha \qquad [\![\varphi[H]]\!]^\pi_\rho = \varphi[[\![H]\!]^\pi_\rho] \qquad [\![\psi\langle H\rangle]\!]^\pi_\rho = \psi\langle[\![H]\!]^\pi_\rho\rangle$$

$$[\![h]\!]^\pi_\rho = \rho(h) \qquad [\![\mu h.H]\!]^\pi_\theta = \bigcup_{n>0} f^n(!) \quad \text{where } f(X) = [\![H]\!]^\pi_{\theta\{X/h\}}$$

$$[\![\{\}]\!]^\pi_\rho = \perp \qquad [\![\{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}]\!]^\pi_\rho = \textstyle\bigcup_{i\in 1..k} [\![\{\pi_i \rhd H_i\}]\!]^\pi_\rho$$

$$[\![\{0 \rhd H\}]\!]^\pi_\rho = [\![H]\!]^\pi_\rho \qquad [\![\{\pi_0 \mid \pi_1 \rhd H\}]\!]^\pi_\rho = [\![\{\pi_0 \rhd H\}]\!]^\pi_\rho \cup_\perp [\![\{\pi_1 \rhd H\}]\!]^\pi_\rho$$

$$[\![\{r[\ell] \rhd H\}]\!]^\pi_\rho = \begin{cases} [\![H]\!]^\pi_\rho & \text{if } \pi = r[\ell] \mid \pi' \\ \perp & \text{otherwise} \end{cases}$$

Example 1. Consider $H = \mu h.\, \alpha + h \cdot h + \varphi[h]$. For all plans π, the semantics of H consists of all the (possibly truncated) histories having an arbitrary number of occurrences of α, and arbitrarily nested, balanced safety framings of φ. For instance, $\varphi[\alpha]\varphi[\alpha\varphi[\alpha]] \in [\![H]\!]^\pi$, for all plans π. □

Example 2. Let $H = \{r[\ell_1] \rhd \alpha_1 \cdot \{r'[\ell'_1] \rhd \beta_1, r'[\ell'_2] \rhd \beta_2\},\ r[\ell_2] \rhd \alpha_2\}$, and let $\pi = r[\ell_1] \mid r'[\ell'_2]$. Then:

$$\begin{aligned}
[\![H]\!]^\pi &= [\![\{r[\ell_1] \rhd \alpha_1 \cdot \{r'[\ell'_1] \rhd \beta_1, r'[\ell'_2] \rhd \beta_2\}\}]\!]^\pi \cup [\![\{r[\ell_2] \rhd \alpha_2\}]\!]^\pi \\
&= [\![\alpha_1 \cdot \{r'[\ell'_1] \rhd \beta_1, r'[\ell'_2] \rhd \beta_2\}]\!]^\pi \cup \bot \\
&= [\![\alpha_1]\!]^\pi [\![\{r'[\ell'_1] \rhd \beta_1, r'[\ell'_2] \rhd \beta_2\}]\!]^\pi \\
&= \{\alpha_1\}([\![\{r'[\ell'_1] \rhd \beta_1\}]\!]^\pi \cup [\![\{r'[\ell'_2] \rhd \beta_2\}]\!]^\pi) \\
&= \{\alpha_1\}(\bot \cup [\![\beta_2]\!]^\pi) \;=\; \{\alpha_1\}\{\beta_2\} \;=\; \{\alpha_1\beta_2\}
\end{aligned}$$ □

Example 3. Let $H = \alpha + \{r[\ell] \rhd \beta\}$, and let $\pi = 0$. Then:

$$[\![H]\!]^\pi \;=\; [\![\alpha]\!]^\pi \cup_\bot [\![\{r[\ell] \rhd \beta\}]\!]^\pi \;=\; \{\alpha\} \cup_\bot \bot \;=\; \bot$$

This example models a situation where, e.g. H has been extracted from an expression $\texttt{if}\, b\, \texttt{then}\, \alpha\, \texttt{else}\, (\texttt{req}_r \tau)*$, and e_ℓ is the only service whose type is compatible with τ. The semantics of H is undefined (i.e. $\bot$) because, in case b is false, the plan $\pi = 0$ is not able to choose any of the proposed services. □

4.1 Validity

We now define when histories and history expressions are valid. Intuitively, valid histories represent viable computations. Instead, invalid ones happen to violate some security constraints, so they are going to be identified and rejected by our static analysis. For example, consider the history $\eta_0 = \alpha_c\alpha_r\varphi[\alpha_c]$, where φ requires that no α_c occurs after α_r (see Section 2). Then, η_0 is *not* valid according to our intended meaning, because the rightmost α_c occurs within a safety framing enforcing φ, and $\alpha_c\alpha_r\alpha_c$ does not obey φ. Consider now the history $\eta_1 = \alpha\psi\langle\alpha\rangle\alpha_{sgn}$, where ψ requires that eventually α_{sgn}. Then, η_1 is *not* valid, because the event α_{sgn} occurs after the liveness framing has been closed.

Note that our notion of validity ensures that, at each step of execution, the policies enforced by safety and liveness framings can always inspect the *whole* history generated so far. This is motivated by our basic assumption that no events can be hidden. For example, the history $\alpha_0\alpha_1\varphi[\alpha_2]\alpha_3$ is valid when $\alpha_0\alpha_1 \models \varphi$ (to make sure φ is satisfied while entering the framing) and $\alpha_0\alpha_1\alpha_2 \models \varphi$ (even if α_0 and α_1 are outside of the safety framing), while $\alpha_0\alpha_1\alpha_2\alpha_3$ is not required to satisfy φ any longer. However, this is not a limitation, as shown in the discussion preceding Lemma 8 below.

To give a formal definition of validity, we introduce the notion of safe and live sets (S- and L-sets for short), which are sets of histories. For example, the history η_0 above has one S-set $\varphi[\{\alpha_c\alpha_r, \alpha_c\alpha_r\alpha_c\}]$. Intuitively, this means that the scope of the framing $\varphi[\cdots]$ spans over the histories $\alpha_c\alpha_r$ and $\alpha_c\alpha_r\alpha_c$. For each S-set of the form $\varphi[\mathcal{H}]$, validity requires that *all* the histories in $\mathcal{H}$ obey φ. Similarly, η_1 has one L-set, $\psi\langle\{\alpha, \alpha\alpha\}\rangle$. For each L-set $\psi\langle\mathcal{H}\rangle$, validity requires that *at least one* of the histories in $\mathcal{H}$ satisfies ψ.

Some notations are now needed. Let $\eta^\flat$ be the history obtained from η by erasing all the framing events, and let η^∂ be the set of all the prefixes of η, including the empty history ε. For example, if $\eta_0 = \alpha_c\alpha_r\varphi[\alpha_c]$, then $(\eta_0^\flat)^\partial = ((\alpha_c\alpha_r[_\varphi\alpha_c]_\varphi)^\flat)^\partial = (\alpha_c\alpha_r\alpha_c)^\partial = \{\varepsilon, \alpha_c, \alpha_c\alpha_r, \alpha_c\alpha_r\alpha_c\}$.

Let η be a (well-formed) history. To have a short definition of S-set and L-set, it is convenient to balance all the safety framings of η, e.g. $[_\varphi\alpha$ becomes $[_\varphi\alpha]_\varphi = \varphi[\alpha]$. Then, the S-set $S(\eta)$, the L-set $L(\eta)$, and validity of histories and history expressions are defined as follows:

Safe/Live sets and validity

$$
\begin{array}{ll}
S(\varepsilon) = \emptyset & L(\varepsilon) = \emptyset \\
S(\eta\,\alpha) = S(\eta\,\langle_\psi) = S(\eta\,\rangle_\psi) = S(\eta) & L(\eta\,\alpha) = L(\eta\,[_\varphi) = L(\eta\,]_\varphi) = L(\eta) \\
S(\eta_0\,\varphi[\eta_1]) = S(\eta_0\,\eta_1) \cup \varphi[\eta_0^\flat\,(\eta_1^\flat)^\partial] & L(\eta_0\,\psi\langle\eta_1\rangle) = L(\eta_0\,\eta_1) \cup \psi\langle\eta_0^\flat\,(\eta_1^\flat)^\partial\rangle
\end{array}
$$

A history η is *valid* ($\models \eta$ in symbols) when:

$$
\begin{array}{rcl}
\varphi[\mathcal{H}] \in S(\eta) & \Longrightarrow & \forall \eta' \in \mathcal{H}.\, \eta' \models \varphi \\
\psi\langle\mathcal{H}\rangle \in L(\eta) & \Longrightarrow & \exists \eta' \in \mathcal{H}.\, \eta' \models \psi
\end{array}
$$

A history expression H is *π-valid* when $[\![H]\!]^\pi \neq \bot$ and $\eta \in [\![H]\!]^\pi \Longrightarrow \models \eta$

Example 4. Consider again the history $\eta_1 = \alpha\psi\langle\alpha\rangle\alpha_{sgn}$. We have that:

$$
\begin{array}{rl}
L(\eta_1) & = L(\alpha\psi\langle\alpha\rangle\alpha_{sgn}) \\
& = L(\alpha\psi\langle\alpha\rangle) = L(\alpha\alpha) \cup \psi\langle\alpha^\flat(\alpha^\flat)^\partial\rangle \\
& = \emptyset \cup \psi\langle\alpha\{\varepsilon, \alpha\}\rangle = \psi\langle\{\alpha, \alpha\alpha\}\rangle
\end{array}
$$

Since neither $\alpha \models \psi$ nor $\alpha\alpha \models \psi$, then η_1 is not valid. Consider now the history:

$$\eta = \langle_\psi[_\varphi\alpha_1]_\varphi\langle_\psi\alpha_2\rangle_\psi\alpha_3[_{\varphi'}\alpha_4$$

Then, after rewriting η as $\eta]_{\varphi'}$ to balance the safety framings, we have:

$$
\begin{array}{l}
S(\eta) = \{\ \varphi[\{\varepsilon, \alpha_1\}], \varphi'[\{\alpha_1\alpha_2\alpha_3, \alpha_1\alpha_2\alpha_3\alpha_4\}]\ \} \\
L(\eta) = \{\ \psi\langle\{\alpha_1, \alpha_1\alpha_2\}\rangle\ \}
\end{array}
$$

□

5 Type and effect system

We now introduce a type and effect system for our calculus, building upon [4, 44]. Types and type environments, ranged over by τ and Γ, are mostly standard and are defined in the following table. The history expression H in the functional type $\tau \xrightarrow{H} \tau'$ describes the latent effect associated with an abstraction, i.e. one of the histories represented by H is generated when a value is applied to an abstraction with that type. Note that we overload the symbol τ to range over both expression types and request types $\tau \xrightarrow{\varphi[\psi\langle\bullet\rangle]} \tau'$.

Types and Type Environments

$$\tau, \tau' ::= \mathit{unit} \mid \tau \xrightarrow{H} \tau' \qquad \Gamma ::= \emptyset \mid \Gamma; x : \tau \quad (x \notin \mathit{dom}(\Gamma))$$

A typing judgment $\Gamma, H \vdash_{\mathsf{Srv}} e : \tau$ means that, given a service repository Srv (omitted in the index, when immaterial), the expression e evaluates to a value of type τ, and produces a history belonging to the effect H. The relation $\Gamma, H \vdash_{\mathsf{Srv}} e : \tau$ is defined as the least relation closed under the rules below. Typing judgments are similar to those of the simply-typed λ-calculus. The effects in the rule for application are concatenated according to the evaluation order of the call-by-value semantics (function, argument, latent effect). The actual effect of an abstraction is the empty history expression, while the latent effect is equal to the actual effect of the function body. The rule for abstraction constraints the premise to equate the actual and latent effects, up to associativity, commutativity, idempotency and zero of $+$, associativity and zero of $\cdot$, α-conversion, unfolding of recursion, and elimination of vacuous μ-binders. The last rule allows for *weakening* of effects. A service invocation $\mathtt{req}_r\tau$ has an empty actual effect, and a functional type τ', whose latent effect is a planned selection that picks from Srv those services matching the constraints on the request type τ. A more detailed explanation will follow after Example 5.

Typing relation

$$\Gamma, \varepsilon \vdash * : \mathit{unit} \quad \text{(T-Unit)} \qquad \Gamma, \alpha \vdash \alpha : \mathit{unit} \quad \text{(T-Ev)}$$

$$\Gamma, \varepsilon \vdash x : \Gamma(x) \quad \text{(T-Var)} \qquad \frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash e : \tau'}{\Gamma, \varepsilon \vdash \lambda_z x.\, e : \tau \xrightarrow{H} \tau'} \quad \text{(T-Abs)}$$

$$\frac{\Gamma, H \vdash e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash e\, e' : \tau'} \quad \text{(T-App)}$$

$$\frac{\Gamma, H \vdash e : \tau}{\Gamma, \varphi[H] \vdash \varphi[e] : \tau} \quad \text{(T-SF)} \qquad \frac{\Gamma, H \vdash e : \tau}{\Gamma, \psi\langle H\rangle \vdash \psi\langle e\rangle : \tau} \quad \text{(T-LF)}$$

$$\frac{\tau' = \biguplus\{\, \tau \oplus_{r[\ell]} \tau_\ell \mid e_\ell : \tau_\ell \in \mathsf{Srv} \wedge \tau_\ell \approx \tau \,\}}{\Gamma, \varepsilon \vdash_{\mathsf{Srv}} \mathtt{req}_r\tau : \tau'} \quad \text{(T-Req)}$$

$$\frac{\Gamma, H \vdash e : \tau \quad \Gamma, H \vdash e' : \tau}{\Gamma, H \vdash \mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' : \tau} \quad \text{(T-If)} \qquad \frac{\Gamma, H \vdash e : \tau}{\Gamma, H + H' \vdash e : \tau} \quad \text{(T-Wkn)}$$

Example 5. Consider the following expression:

$$e \;=\; \mathtt{if}\ b\ \mathtt{then}\ \lambda_z x.\, \alpha\ \mathtt{else}\ \lambda_z x.\, \alpha'$$

Let $\tau = \mathit{unit}$, and $\Gamma = \{z : \tau \xrightarrow{\alpha+\alpha'} \tau; x : \tau\}$. Then, the following typing

derivation is possible:

$$\dfrac{\dfrac{\dfrac{\Gamma, \alpha \vdash \alpha : \tau}{\Gamma, \alpha + \alpha' \vdash \alpha : \tau}}{\emptyset, \varepsilon \vdash \lambda_z x.\, \alpha : \tau \xrightarrow{\alpha+\alpha'} \tau} \qquad \dfrac{\dfrac{\Gamma, \alpha' \vdash \alpha' : \tau}{\Gamma, \alpha' + \alpha \vdash \alpha' : \tau}}{\emptyset, \varepsilon \vdash \lambda_z x.\, \alpha' : \tau \xrightarrow{\alpha'+\alpha} \tau}}{\emptyset, \varepsilon \vdash \mathtt{if}\ b\ \mathtt{then}\ \lambda_z x.\, \alpha\ \mathtt{else}\ \lambda_z x.\, \alpha' : \tau \xrightarrow{\alpha+\alpha'} \tau}$$

Note that we can equate the history expressions $\alpha + \alpha'$ and $\alpha' + \alpha$, because $+$ is commutative. The typing derivation above shows the use of the weakening rule to unify the latent effects on arrow types. Let now:

$$e' \;=\; \lambda_w x.\, \mathtt{if}\ b'\ \mathtt{then}\ *\ \mathtt{else}\ w(e\,x)$$

Let $\Gamma = \{w : \tau \xrightarrow{H} \tau, x : \tau\}$, where H is left undefined. Then, recalling that $\varepsilon \cdot H' = H' = H' \cdot \varepsilon$ for any history expression H', we have:

$$\dfrac{\Gamma, \varepsilon \vdash * : \tau \qquad \dfrac{\dfrac{\Gamma, \varepsilon \vdash w : \tau \xrightarrow{H} \tau \quad \dfrac{\Gamma, \varepsilon \vdash e : \tau \xrightarrow{\alpha+\alpha'} \tau \quad \Gamma, \varepsilon \vdash x : \tau}{\Gamma, \alpha + \alpha' \vdash e\,x : \tau}}{\Gamma, (\alpha + \alpha') \cdot H \vdash w(e\,x) : \tau}}{\Gamma, \varphi[(\alpha + \alpha') \cdot H] \vdash \varphi[w(e\,x)] : \tau}}{\Gamma, \varepsilon + \varphi[(\alpha + \alpha') \cdot H] \vdash \mathtt{if}\ b'\ \mathtt{then}\ *\ \mathtt{else}\ \varphi[w(e\,x)] : \tau}$$

To apply the typing rule for abstractions, the constraint $H = \varepsilon + \varphi[(\alpha + \alpha') \cdot H]$ must be solved. Let $H = \mu h.\, \varepsilon + \varphi[(\alpha + \alpha') \cdot h]$. It is easy to prove that:

$$[\![H]\!] \;=\; [\![\varepsilon + \varphi[(\alpha + \alpha') \cdot h]]\!]_{\{[\![H]\!]/h\}} \;=\; \{\varepsilon\} \cup \varphi[(\alpha + \alpha') \cdot [\![H]\!]]$$

We have then found a solution to the constraint above, so we can conclude that:

$$\emptyset, \varepsilon \vdash e' : \tau \xrightarrow{\mu h.\, \varepsilon + \varphi[(\alpha+\alpha') \cdot h]} \tau$$

Note in passing that a simple extension of the type inference algorithm of [44] suffices for solving constraints as the one above. The main difference concerns planned selections. To deal with that, our algorithm uses a straightforward implementation of the rule (T-Req) □

To give a type to requests, we need to define the auxiliary operators $\approx$, $\oplus$ and $\Cup$. We introduce them below, with the help of a running example.

We write $\tau \approx \tau'$, and say τ, τ' *compatible*, whenever, omitting the annotations on the arrows, τ and τ' are equal. Formally:

$$\mathit{unit} \approx \mathit{unit} \qquad (\tau_0 \xrightarrow{X} \tau_1) \approx (\tau_0' \xrightarrow{Y} \tau_1') \quad \text{iff } \tau_0 \approx \tau_0' \text{ and } \tau_1 \approx \tau_1'$$

Example 6. Let $\tau = \mathit{unit}$, let $e = \mathtt{req}_r \tau_r$, where $\tau_r = (\tau \to \tau) \xrightarrow{\varphi[\bullet]} (\tau \xrightarrow{\psi\langle\bullet\rangle} \tau)$, and let $\mathsf{Srv} = \{e_{\ell_1} : \tau_{\ell_1}, e_{\ell_2} : \tau_{\ell_2}\}$, where $\tau_{\ell_i} = (\tau \xrightarrow{h_i} \tau) \xrightarrow{\alpha_i \cdot h_i} (\tau \xrightarrow{\beta_i} \tau)$ for $i \in 1..2$. We have that $\tau_r \approx \tau_{\ell_1} \approx \tau_{\ell_2}$, i.e. both the services in Srv are compatible with the request in e. □

The operator $\oplus_{r[\ell]}$ combines a request type τ and a service type τ', when they are compatible. Given a request type $\hat{\tau} = \tau_0 \xrightarrow{\varphi[\psi\langle\bullet\rangle]} \tau_1$ and a service type $\hat{\tau}' = \tau'_0 \xrightarrow{H} \tau'_1$, the result of $\hat{\tau} \oplus_{r[\ell]} \hat{\tau}'$ is $\tau'_0 \xrightarrow{\{r[\ell]\rhd\varphi[\psi\langle H\rangle]\}} (\tau_1 \oplus'_{r[\ell]} \tau'_1)$, where:

$$\begin{array}{l} \mathit{unit} \oplus'_{r[\ell]} \mathit{unit} = \mathit{unit} \\ (\tau_0 \xrightarrow{\varphi[\psi\langle\bullet\rangle]} \tau_1) \oplus'_{r[\ell]} (\tau'_0 \xrightarrow{H} \tau'_1) = (\tau_0 \oplus'_{r[\ell]} \tau'_0) \xrightarrow{\{r[\ell]\rhd\varphi[\psi\langle H\rangle]\}} (\tau_1 \oplus'_{r[\ell]} \tau'_1) \end{array}$$

Note that combining functional types does not affect the type of the argument. This reflects the intuition that the type of the argument to be passed to the selected service cannot be constrained by the request.

Example 6 (cont.)**.** The request type τ_r is composed with the service types in Srv as follows:

$$\begin{array}{l} \tau_r \oplus_{r[\ell_1]} \tau_{\ell_1} = (\tau \xrightarrow{h_1} \tau) \xrightarrow{\{r[\ell_1]\rhd\varphi[\alpha_1\cdot h_1]\}} (\tau \xrightarrow{\{r[\ell_1]\rhd\psi\langle\beta_1\rangle\}} \tau) \\ \tau_r \oplus_{r[\ell_2]} \tau_{\ell_2} = (\tau \xrightarrow{h_2} \tau) \xrightarrow{\{r[\ell_2]\rhd\varphi[\alpha_2\cdot h_2]\}} (\tau \xrightarrow{\{r[\ell_2]\rhd\psi\langle\beta_2\rangle\}} \tau) \end{array} \qquad \square$$

Eventually, the operator $\Cup$ combines the types obtained by combining the request type with the service types. Given two compatible types $\hat{\tau} = \tau_0 \xrightarrow{H} \tau_1$ and $\hat{\tau}' = \tau'_0 \xrightarrow{H'} \tau'_1$, the result of $\hat{\tau} \Cup \hat{\tau}'$ is $\tau''_0 \xrightarrow{H\cup H'} (\tau_1\sigma \Cup' \tau'_1\sigma)$, where σ unifies τ_0 and τ'_0 (i.e. $\tau_0\sigma = \tau'_0\sigma = \tau''_0$), and:

$$\begin{array}{l} \mathit{unit} \Cup' \mathit{unit} = \mathit{unit} \\ (\tau_0 \xrightarrow{H} \tau_1) \Cup' (\tau'_0 \xrightarrow{H'} \tau'_1) = (\tau_0 \Cup' \tau'_0) \xrightarrow{H\cup H'} (\tau_1 \Cup' \tau'_1) \end{array}$$

Example 6 (cont.)**.** Now, we can unify the combination of the request type τ_r with the service types, obtaining:

$$\tau' \;=\; (\tau \xrightarrow{h} \tau) \xrightarrow{\{r[\ell_1]\rhd\varphi[\alpha_1\cdot h],\, r[\ell_2]\rhd\varphi[\alpha_2\cdot h]\}} (\tau \xrightarrow{\{r[\ell_1]\rhd\psi\langle\beta_1\rangle,\, r[\ell_2]\rhd\psi\langle\beta_2\rangle\}} \tau)$$

where $\sigma = \{h_1/h, h_2/h\}$ is the selected unifier between $\tau \xrightarrow{h_1} \tau$ and $\tau \xrightarrow{h_2} \tau$. $\square$

The following example further illustrates how requests and services are typed.

Example 7. Let e and Srv as in Example 6, and consider the expression $(e(\lambda.\gamma))*$. Note that applying (any service resulting from) the request to the function $\lambda.\gamma$ yields a new function, which we eventually apply to the value $*$. We have the following typing derivation, where we omit the component $\Gamma = \emptyset$:

$$\dfrac{\dfrac{\varepsilon \vdash e : \tau' \qquad \varepsilon \vdash (\lambda.\gamma) : \tau \xrightarrow{\gamma} \tau}{\{r[\ell_1] \rhd \varphi[\alpha_1\cdot\gamma],\ r[\ell_2] \rhd \varphi[\alpha_2\cdot\gamma]\} \vdash e(\lambda.\gamma) : \tau \xrightarrow{\{r[\ell_1]\rhd\psi\langle\beta_1\rangle,\, r[\ell_2]\rhd\psi\langle\beta_2\rangle\}} \tau}}{\{r[\ell_1] \rhd \varphi[\alpha_1\cdot\gamma],\ r[\ell_2] \rhd \varphi[\alpha_2\cdot\gamma]\} \cdot \{r[\ell_1] \rhd \psi\langle\beta_1\rangle,\ r[\ell_2] \rhd \psi\langle\beta_2\rangle\} \vdash (e(\lambda.\gamma))* : \tau}$$

Evaluating the resulting history expression H with the plan $\pi = r[\ell_1]$ yields:

$$\begin{array}{l} [\![H]\!]^\pi = [\![\{r[\ell_1] \rhd \varphi[\alpha_1\cdot\gamma],\ r[\ell_2] \rhd \varphi[\alpha_2\cdot\gamma]\}]\!]^\pi \cdot [\![\{r[\ell_1] \rhd \psi\langle\beta_1\rangle,\ r[\ell_2] \rhd \psi\langle\beta_2\rangle\}]\!]^\pi \\ \qquad = (\{\varphi[\alpha_1\gamma]\} \cup \bot) \cdot (\{\psi\langle\beta_1\rangle\} \cup \bot) \;=\; \{\varphi[\alpha_1\gamma]\ \psi\langle\beta_1\rangle\} \end{array} \qquad \square$$

A plan is *well-typed* if, when it associates a service to a request $\mathtt{req}_r\tau$, then the type of the service is compatible with the type of the request:

$$\pi = r[\ell] \mid \pi' \ \wedge \ e_\ell : \tau_\ell \in \mathsf{Srv} \implies \tau_\ell \approx \tau$$

The next theorem states that our type and effect system over-approximates the actual run-time histories, for a given expression and a given service repository Srv. If H is the effect of e, then the histories η generated when evaluating e under the plan π are the prefixes of the histories in $[\![H]\!]^\pi$, stripped of all the framing events, i.e. $\eta \in ([\![H]\!]^\pi)^{\flat\partial}$. As usual, precision is lost when reducing the conditional construct to non-determinism, and when dealing with recursive functions (see Example 5). Additionally, we over-approximate the set of services satisfying the calling requirements.

Theorem 1. Given a service repository Srv, if $\Gamma, H \vdash_{\mathsf{Srv}} e : \tau$ and $\varepsilon, e \to_\pi^* \eta, e'$ (with π well-typed), then $\eta \in ([\![H]\!]^\pi)^{\flat\partial}$.

Example 8. Let $e = \alpha_0; \varphi[\mathtt{if}\ b\ \mathtt{then}\ \alpha_1\ \mathtt{else}\ \alpha_2]$, with φ requiring "never α_2". Assume that the guard b always evaluates to true. Then, for any plan π, we have the following computation:

$$\varepsilon, e \to_\pi \alpha_0, \varphi[\mathtt{if}\ b\ \mathtt{then}\ \alpha_1\ \mathtt{else}\ \alpha_2] \to_\pi \alpha_0, \varphi[\alpha_1] \to_\pi \alpha_0\alpha_1, \varphi[*] \to_\pi \alpha_0\alpha_1, *$$

The history expression extracted from e is $H = \alpha_0 \cdot \varphi[\alpha_1 + \alpha_2]$. Then, for all plans π, $[\![H]\!]^\pi = \{\alpha_0[_\varphi\alpha_1]_\varphi, \alpha_0[_\varphi\alpha_2]_\varphi\}$, and:

$$(([\![H]\!]^\pi)^\flat)^\partial = \{\alpha_0\alpha_1, \alpha_0\alpha_2\}^\partial = \{\varepsilon, \alpha_0, \alpha_0\alpha_1, \alpha_0\alpha_2\}$$

which (strictly) contains all the run-time histories in the computation above. □

We can now state the type safety property. A plan π is *viable* for e when the reduction of e with plan π does not go wrong, i.e. $\varepsilon, e \to_\pi^* \eta', e'$, and either e' is a value, or there exists a transition $\eta', e' \to_\pi \eta'', e''$ for some η'', e''. For example, a computation goes wrong when attempting to execute an event forbidden by a currently active policy, or when the plan π offers no choices for a request.

Theorem 2 (Type Safety)**.** Given a service repository Srv, let $\Gamma, H \vdash_{\mathsf{Srv}} e : \tau$, with e closed. If H is π-valid for some well-typed plan π, then π is viable for e.

Example 9. Let e and Srv as in Example 6, and let $e_{\ell_i} = \lambda x.\,(\alpha_i; (x*); (\lambda.\beta_i))$ for $i \in 1..2$. Assume the constraints φ, ψ on the service request in e are such that φ is true, and ψ requires "eventually β_1". Consider now the expression $e' = \varphi_0[(e(\lambda.\gamma))*]$, where φ_0 requires "never α_2". Let $\pi = r[\ell_1]$. The history expression $\varphi_0[H]$ of e' (where H has been inferred for (e $(\lambda.\gamma))*$ in Example 7) is π-valid, because $[\![\varphi_0[H]]\!]^\pi = \{\varphi_0[\varphi[\alpha_1\gamma]\psi\langle\beta_1\rangle]\}$ contains exactly one valid history. We have exactly one computation of e' with plan π, and, as predicted by Theorem 2, it does not go wrong:

$$\begin{aligned}
\varepsilon, \varphi_0[(e(\lambda.\gamma))*] &\to_\pi \varepsilon, \varphi_0[(e_{\ell_1}(\lambda.\gamma))*] \ \to_\pi \ \varepsilon, \varphi_0[(\alpha_1; (\lambda.\gamma)*; (\lambda.\beta_1))*] \\
&\to_\pi \alpha_1, \varphi_0[(\lambda.\gamma)*; (\lambda.\beta_1))*] \ \to_\pi \ \alpha_1, \varphi_0[(\gamma; (\lambda.\beta_1))*] \\
&\to_\pi \alpha_1\gamma, \varphi_0[(\lambda.\beta_1)*] \ \to_\pi \ \alpha_1\gamma, \varphi_0[\beta_1] \ \to_\pi \ \alpha_1\gamma\beta_1, \varphi_0[*] \\
&\to_\pi \alpha_1\gamma\beta_1, *
\end{aligned}$$

Consider now the plan $\pi' = r[\ell_2]$. Then H is not π'-valid, because e.g. the event α_2 violates φ_0. In this case the computation of e':

$$\varepsilon, \varphi_0[(e(\lambda.\gamma))*] \to_{\pi'} \varepsilon, \varphi_0[(e_{\ell_2}(\lambda.\gamma))*] \ \to_{\pi'} \ \varepsilon, \varphi_0[(\alpha_2; (\lambda.\gamma)*; (\lambda.\beta_1))*]$$

is correctly aborted, because $\alpha_2 \not\models \varphi_0$. So, π' is not viable for e'. □

Validity of history expressions also guarantees an additional correctness property about the "liveness" of a service. Roughly, it says that if $\psi\langle e\rangle$ has a valid effect, then e will eventually escape the liveness framing, i.e. the policy ψ will be eventually obeyed. We need the notion of evaluation context $\mathcal{C}(\bullet)$, defined by the following grammar: $\bullet \mid v\,\mathcal{C}(\bullet) \mid \mathcal{C}(\bullet)\,e \mid \varphi[\mathcal{C}(\bullet)] \mid \psi\langle\mathcal{C}(\bullet)\rangle$. Additionally, a λ-abstraction $\lambda_z x.\, e$ is *guarded* if $e = \alpha; e'$ for some event α and expression e'.

Theorem 3. Let $\Gamma, H \vdash_{\mathsf{Srv}} e : \tau$, with e closed, and H valid for some well-typed plan π. Let $\varepsilon, e \to_\pi^* \eta, \mathcal{C}(\psi\langle e'\rangle)$, with e' having guarded λ-abstractions only. Then, there exists k such that, for each computation:

$$\eta, \mathcal{C}(\psi\langle e'\rangle) \to_\pi \eta_1, e_1 \to_\pi \cdots \to_\pi \eta_k, e_k$$

there exists $n \leq k$ such that $e_n = \mathcal{C}(\psi\langle e''\rangle)$ and $\eta_n \models \psi$.

Intuitively, if a computation is long enough (i.e. at least k steps), it will always escape all liveness framings in a finite number of steps $n \leq k$ — because, by rule (E-LF2), $\eta_n, \mathcal{C}(\psi\langle e''\rangle) \to_\pi \eta_n, \mathcal{C}(e'')$.

Example 10. Consider the expressions $e_1 = \psi\langle(\lambda_z x.\, \mathtt{if}\ b\ \mathtt{then}\ \beta\ \mathtt{else}\ \alpha; \beta; zx)*\rangle$ and $e_2 = \psi\langle(\lambda_z x.\, \mathtt{if}\ b\ \mathtt{then}\ \beta\ \mathtt{else}\ \alpha; zx; \beta)*\rangle$, where ψ ask "eventually β". Let $H_1 = \psi\langle\mu h.\, \alpha \cdot \beta \cdot h + \beta\rangle$ and $H_2 = \psi\langle\mu h.\, \alpha \cdot h \cdot \beta + \beta\rangle$ be the history expressions associated with e_1 and e_2, respectively. Since H_1 is valid, Theorem 3 guarantees that *all* the computations of e_1 will satisfy the policy ψ. Instead, since H_2 is *not* valid, the expression e_2 might never obey ψ, as shown by the following computation (where $b = \mathit{ff}$ and $Z = \lambda_z x.\, \mathtt{if}\ b\ \mathtt{then}\ \beta\ \mathtt{else}\ \alpha; zx; \beta$):

$$\begin{aligned}
\varepsilon, e_2 &\to_\pi \varepsilon, \psi\langle \mathtt{if}\ b\ \mathtt{then}\ \beta\ \mathtt{else}\ \alpha; Z*; \beta\rangle \\
&\to_\pi \varepsilon, \psi\langle \alpha; Z*; \beta\rangle \ \to_\pi \ \alpha, \psi\langle Z*; \beta\rangle \\
&\to_\pi \alpha, \psi\langle \mathtt{if}\ b\ \mathtt{then}\ \beta\ \mathtt{else}\ \alpha; Zx; \beta; \beta\rangle \\
&\to_\pi \alpha, \psi\langle \alpha; Z*; \beta; \beta\rangle \ \to_\pi \ \alpha\alpha, \psi\langle Z*; \beta; \beta\rangle \ \to_\pi \cdots
\end{aligned}$$

□

Note however that, since our notion of validity is extensional rather than intensional, we still do not have a decision procedure to tell us for which plans π (if any) a history expression is π-valid. This problem is addressed by the planning and verification methods presented in Sections 6 and 7.

6 Planning service composition

Once extracted a history expression H from an expression e, we have to analyse H to find if there is any viable plan for the execution of e. This issue is not trivial, because the effect of selecting a given service for a request is not confined to the execution of that service. For instance, the history generated while running a

service may later on violate a policy that will become active after the service has returned, as shown in Example 11 below. Since each service selection affects the *whole* execution of a program, we cannot simply devise a viable plan by selecting services that satisfy the constraints imposed by the requests, only.

Example 11. Let $e = \varphi[(\lambda x.\,(\mathtt{req}_{r_2}\,\tau_2)x)\,((\mathtt{req}_{r_1}\,\tau_1)*)]$, where φ requires "never γ after α", $\tau = \mathit{unit}$, $\tau_1 = \tau \to (\tau \to \tau)$ and $\tau_2 = (\tau \to \tau) \to \tau$. Intuitively, the service selected upon the request r_1 returns a function, which is then passed as an argument to the service selected upon r_2. Assume the repository Srv comprises exactly the following four services:

$$e_{\ell_1} : \tau \xrightarrow{\alpha} (\tau \xrightarrow{\beta} \tau) \qquad e_{\ell_2} : (\tau \xrightarrow{h} \tau) \xrightarrow{h\cdot\gamma} \tau$$
$$e_{\ell'_1} : \tau \xrightarrow{\alpha'} (\tau \xrightarrow{\beta'} \tau) \qquad e_{\ell'_2} : (\tau \xrightarrow{h} \tau) \xrightarrow{\varphi'[h]} \tau$$

where φ' requires "never β'". Since the request type τ_1 imposes no constraints and matches the types of e_{ℓ_1} and $e_{\ell'_1}$, both these services can be selected for the request r_1. Similarly, both e_{ℓ_2} and $e_{\ell'_2}$ can be chosen for r_2. Therefore, we have to consider four possible plans when evaluating the history expression H of e:

$$H = \varphi[\{r_1[\ell_1] \rhd \alpha, r_1[\ell'_1] \rhd \alpha'\}\cdot \\ \{r_2[\ell_2] \rhd \{r_1[\ell_1] \rhd \beta, r_1[\ell'_1] \rhd \beta'\}\cdot\gamma,\ r_2[\ell'_2] \rhd \varphi'[\{r_1[\ell_1] \rhd \beta, r_1[\ell'_1] \rhd \beta'\}]\}]$$

Consider first the plan $\pi_1 = r_1[\ell_1] \mid r_2[\ell_2]$. Then, $[\![H]\!]^{\pi_1} = \varphi[\alpha\beta\gamma]$ is *not* valid and π_1 is not viable for e, because the policy φ is violated. Consider now $\pi_2 = r_1[\ell'_1] \mid r_2[\ell'_2]$. Then, $[\![H]\!]^{\pi_2} = \varphi[\alpha'\varphi'[\beta']]$ is *not* valid and π_2 is not viable for e, because the policy φ' is violated. Instead, the remaining two plans, $r_1[\ell_1] \mid r_2[\ell'_2]$ and $r_1[\ell'_1] \mid r_2[\ell_2]$ are viable for e. □

As shown above, the tree-shaped structure of planned selections makes it difficult to determine the plans π under which a history expression is valid. Things become easier if we "linearize" such a tree structure into a set of history expressions, forming an equivalent planned selection $\{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$, where no H_i has further selections. E.g., the linearization of H in Example 11 is:

$$\{r_1[\ell_1] \mid r_2[\ell_2] \rhd \varphi[\alpha\cdot\beta\cdot\gamma],\quad r_1[\ell_1] \mid r_2[\ell'_2] \rhd \varphi[\alpha\cdot\varphi'[\beta]], \\ r_1[\ell'_1] \mid r_2[\ell_2] \rhd \varphi[\alpha'\cdot\beta'\cdot\gamma],\ r_1[\ell'_1] \mid r_2[\ell'_2] \rhd \varphi[\alpha'\cdot\varphi'[\beta']]\}$$

Formally, we say that H is *equivalent* to H' ($H \equiv H'$ in symbols) when $[\![H]\!]^\pi_\rho = [\![H']\!]^\pi_\rho$, for each ρ and plan π. The following properties of $\equiv$ hold.

Theorem 4. The relation $\equiv$ is a congruence, and it satisfies the equations between planned selections displayed in the following table.

Equational properties of planned selections

$$H \equiv \{0 \rhd H\} \tag{1}$$
$$\{\pi_i \rhd H_i\}_{i\in I} \cdot \{\pi'_j \rhd H'_j\}_{j\in J} \equiv \{\pi_i \mid \pi'_j \rhd H_i \cdot H'_j\}_{i\in I, j\in J} \tag{2}$$
$$\{\pi_i \rhd H_i\}_{i\in I} + \{\pi'_j \rhd H'_j\}_{j\in J} \equiv \{\pi_i \mid \pi'_j \rhd H_i + H'_j\}_{i\in I, j\in J} \tag{3}$$
$$\varphi[\{\pi_i \rhd H_i\}_{i\in I}] \equiv \{\pi_i \rhd \varphi[H_i]\}_{i\in I} \tag{4}$$
$$\psi\langle\{\pi_i \rhd H_i\}_{i\in I}\rangle \equiv \{\pi_i \rhd \psi\langle H_i\rangle\}_{i\in I} \tag{5}$$
$$\mu h.\, \{\pi_i \rhd H_i\} \equiv \{\pi_i \rhd \mu h.\, H_i\}_{i\in I} \tag{6}$$
$$\{\pi_i \rhd \{\pi'_{i,j} \rhd H_{i,j}\}_{j\in J}\}_{i\in I} \equiv \{\pi_i \mid \pi'_{i,j} \rhd H_{i,j}\}_{i\in I, j\in J} \tag{7}$$

Example 12. Consider again the history expression computed in Example 7:

$$H = \{r[\ell_1] \rhd \varphi[\alpha_1 \cdot \gamma],\ r[\ell_2] \rhd \varphi[\alpha_2 \cdot \gamma]\} \cdot \{r[\ell_1] \rhd \psi\langle\beta_1\rangle,\ r[\ell_2] \rhd \psi\langle\beta_2\rangle\}$$

Applying the equation (2) of Theorem 4, we obtain:

$$\begin{aligned} H &\equiv \{r[\ell_1] \mid r[\ell_1] \rhd \varphi[\alpha_1 \cdot \gamma] \cdot \psi\langle\beta_1\rangle,\ r[\ell_1] \mid r[\ell_2] \rhd \varphi[\alpha_1 \cdot \gamma] \cdot \psi\langle\beta_2\rangle, \\ &\qquad r[\ell_2] \mid r[\ell_1] \rhd \varphi[\alpha_2 \cdot \gamma] \cdot \psi\langle\beta_1\rangle,\ r[\ell_2] \mid r[\ell_2] \rhd \varphi[\alpha_2 \cdot \gamma] \cdot \psi\langle\beta_2\rangle\} \\ &= \{r[\ell_1] \rhd \varphi[\alpha_1 \cdot \gamma] \cdot \psi\langle\beta_1\rangle,\ r[\ell_2] \rhd \varphi[\alpha_2 \cdot \gamma] \cdot \psi\langle\beta_2\rangle\} \end{aligned}$$

In the last step, we have used idempotency of $|$, and we have purged the resulting selection from the plan $r[\ell_1] \mid r[\ell_2]$, which is not injective. □

Example 13. Let $H = \mu h.\, \{r[\ell_1] \rhd \alpha_1,\ r[\ell_2] \rhd \alpha_2\} \cdot h$. Then, using equations (1), (2) and (6) of Theorem 4, and the identity of the plan 0, we obtain:

$$\begin{aligned} H &\equiv \mu h.\, \{r[\ell_1] \rhd \alpha_1,\ r[\ell_2] \rhd \alpha_2\} \cdot \{0 \rhd h\} \\ &\equiv \mu h.\, \{r[\ell_1] \mid 0 \rhd \alpha_1 \cdot h,\ r[\ell_2] \mid 0 \rhd \alpha_2 \cdot h\} \\ &= \mu h.\, \{r[\ell_1] \rhd \alpha_1 \cdot h,\ r[\ell_2] \rhd \alpha_2 \cdot h\} \\ &\equiv \{r[\ell_1] \rhd \mu h.\, \alpha_1 \cdot h,\ r[\ell_2] \rhd \mu h.\, \alpha_2 \cdot h\} \end{aligned}$$

Note that the original H can choose a service among ℓ_1 and ℓ_2 at *each* iteration of the loop. Instead, in the linearization of H, the request r will be resolved into the *same* service at each iteration. □

Example 14. Consider again the history expression of Example 2. Applying equations (1), (2) and (7) of Theorem 4, we obtain:

$$\begin{aligned} H &= \{r[\ell_1] \rhd \alpha_1 \cdot \{r'[\ell'_1] \rhd \beta_1, r'[\ell'_2] \rhd \beta_2\},\ r[\ell_2] \rhd \alpha_2\} \\ &\equiv \{r[\ell_1] \rhd \{0 \rhd \alpha_1\} \cdot \{r'[\ell'_1] \rhd \beta_1, r'[\ell'_2] \rhd \beta_2\},\ r[\ell_2] \rhd \alpha_2\} \\ &\equiv \{r[\ell_1] \rhd \{0 \mid r'[\ell'_1] \rhd \alpha_1 \cdot \beta_1, 0 \mid r'[\ell'_2] \rhd \alpha_1 \cdot \beta_2\},\ r[\ell_2] \rhd \alpha_2\} \\ &= \{r[\ell_1] \rhd \{r'[\ell'_1] \rhd \alpha_1 \cdot \beta_1, r'[\ell'_2] \rhd \alpha_1 \cdot \beta_2\},\ r[\ell_2] \rhd \alpha_2\} \\ &\equiv \{r[\ell_1] \mid r'[\ell'_1] \rhd \alpha_1 \cdot \beta_1,\ r[\ell_1] \mid r'[\ell'_2] \rhd \alpha_1 \cdot \beta_2,\ r[\ell_2] \rhd \alpha_2\} \end{aligned}$$ □

We say that a history expression H is *linear* when $H = \{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$, the plans are pairwise *independent* (i.e. $\pi_i \neq \pi_j|\pi$ for all $i \neq j$ and π) and no H_i has planned selections.

Given a history expression H, we obtain its linearization in three steps. First, we apply the first equation of Theorem 4 to each event, variable and ε in H. Then, we orient the equations of Theorem 4 from left to right, obtaining a rewriting system that is easily proved finitely terminating and confluent – up to the equational laws of the algebra of plans. The resulting planned selection $H' = \{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$ has no further selections in H_i, but there may be non-independent plans $\pi_i \sqsubseteq \pi_j$ (recall that we discard $\pi_i \triangleright H_i$ when π_i is not injective). In the third linearization step, for each such pairs, we update H' by inserting $\pi_i \triangleright H_i + H_j$, and removing $\pi_j \triangleright H_j$.

The following result enables us to detect the viable plans for service composition: executions driven by any of them will never violate the security constraints on demand.

Theorem 5. If $H = \{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$ is linear, and H_i is valid for some $i \in 1..k$, then H is π_i-valid.

Summing up, we extract from an expression e a history expression H, we linearize it into $\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$, and if some H_i is valid, then we can deduce that H is π_i-valid. By Theorem 2, the plan π_i safely drives the execution of e, without resorting to any run-time monitor. It remains then to verify the validity of history expressions that, like the H_i above, have no planned selections.

7 Verifying validity

The last step in our technical development mechanically verifies the validity of history expressions with no planned selections. Our technique is based on model checking Basic Process Algebras (BPAs) with Finite State Automata (FSA). The standard decision procedure for verifying that a BPA process p satisfies a ω-regular property φ amounts to constructing the pushdown automaton for p and the Büchi automaton for the negation of φ. Then, the property holds if it is empty the (context-free) language accepted by the conjunction of the above, which is still a pushdown automaton. This problem is known to be decidable, and several algorithms and tools show this approach feasible [24]. Since our histories are always finite, it turns out that we can use Finite State Automata instead of Büchi Automata.

Recall however that, as it is, our notion of validity is non-regular, because of the arbitrary nesting of framings. As an example, consider again the history expression $H = \mu h.\, \alpha + h \cdot h + \varphi[h]$. The language $[\![H]\!]^\pi$ is context-free and non-regular, because it contains unbounded pairs of balanced $[_\varphi$ and $]_\varphi$ (for all plans π, as H contains no planned selections, due to Theorem 5). Since context-free languages are not closed under intersection, the emptiness problem is undecidable. To apply the procedure sketched above, we will first manipulate history expressions in order to make validity a regular property.

7.1 Redundant framings

History expressions can generate histories with *redundant framings*, i.e. nesting of the same framing. For example, the history $\eta = \alpha\varphi[\alpha'\varphi'[\varphi[\alpha'']]]$ has an inner redundant safety framing φ around α''. Since α'' is already under the scope of the outermost φ-framing, it happens that η is valid if and only if $\alpha\varphi[\alpha'\varphi'[\alpha'']]$

is valid. Formally, the S-sets of η comprise $\varphi[\{\alpha, \alpha\alpha', \alpha\alpha'\alpha''\}]$ for the outer framing, and $\varphi[\{\alpha\alpha', \alpha\alpha'\alpha''\}]$ for the inner one. Validity requires that all the histories in $\{\alpha, \alpha\alpha', \alpha\alpha'\alpha''\}$ and $\{\alpha\alpha', \alpha\alpha'\alpha''\}$ obey φ. Since the second set is strictly included in the first one, then the *inner* safety framing is redundant.

Similarly, consider the history $\eta' = \alpha\psi\langle\alpha'\psi\langle\alpha''\rangle\rangle$. The L-sets of η' are $\psi[\{\alpha, \alpha\alpha', \alpha\alpha'\alpha''\}]$ for the outer framing and $\psi[\{\alpha\alpha', \alpha\alpha'\alpha''\}]$ for the inner one. Since the first set includes the second one, and validity requires that there exists a history in the L-set satisfying ψ, then the *outer* framing is redundant.

Removing redundant framings from a history preserves its validity. But one needs the expressive power of a pushdown automaton, because framings openings and closings are to be matched in pairs. For example, consider the following history:

$$\eta \;=\; \alpha \overbrace{[_\varphi \cdots [_\varphi}^{n} \overbrace{]_\varphi \cdots]_\varphi}^{m} [_\varphi$$

The last $[_\varphi$ is redundant if $n > m$, and is *not* if $n = m$.

As a matter of fact, it turns out that it is possible to regularize the safety side of validity, by removing the redundant safety framings. For the liveness side, this approach seems not feasible, but, surprisingly enough, a tailored construction of finite state automata suffices.

7.2 Regularization of safety framings

Below, we define a transformation that, given a history expression H, yields a H' that does not generate redundant safety framings, and H' is valid if and only if H is such. Recall that there is no need for regularizing planned selections, because, by Theorem 5, we will always verify the validity of history expressions with no selections.

Let $\tilde{H}$ be a history expression with a hole $\bullet$, and let $H = \tilde{H}\{H'/\bullet\}$ be a history expression, for some H'. We say that H' *is guarded by* φ *in* H when $\varphi \in \mathit{guard}(\tilde{H})$, defined as the smallest set satisfying the following equations.

Guards

$$\begin{array}{rcll}
\mathit{guard}(H_0 \cdot H_1) & = & \mathit{guard}(H_i) & \text{if } \bullet \in H_i \\
\mathit{guard}(H_0 + H_1) & = & \mathit{guard}(H_i) & \text{if } \bullet \in H_i \\
\mathit{guard}(\varphi[H]) & = & \{\varphi\} \cup \mathit{guard}(H) & \\
\mathit{guard}(\psi\langle H\rangle) & = & \mathit{guard}(H) & \\
\mathit{guard}(\mu h.\ H) & = & \mathit{guard}(H) &
\end{array}$$

Example 15. Let $H = \varphi[\alpha{\cdot}h_0{\cdot}\varphi'[\alpha'{+}h_1]]{\cdot}h_2$, and let $\tilde{H} = \varphi[\alpha{\cdot}h_0{\cdot}\varphi'[\alpha'{+}\bullet]]{\cdot}h_2$. Then, $H = \tilde{H}\{h_1/\bullet\}$, and so h_1 is guarded by $\mathit{guard}(\tilde{H}) = \{\varphi, \varphi'\}$. Similarly, h_0 is guarded by $\{\varphi\}$, and h_2 is unguarded (i.e. guarded by $\emptyset$). □

Let H be a (possibly non-closed) history expression. Without loss of generality, assume that all the variables in H have distinct names. We define below $H \downarrow_{\Phi,\Omega}$, the history expression produced by the *regularization* of H against a set of policies Φ and a mapping Ω from variables to history expressions.

Regularization of safety framings

$$\varepsilon\downarrow_{\Phi,\Omega} = \varepsilon \qquad h\downarrow_{\Phi,\Omega} = h \qquad \alpha\downarrow_{\Phi,\Omega} = \alpha$$

$$(H \cdot H')\downarrow_{\Phi,\Omega} = H\downarrow_{\Phi,\Omega} \cdot H'\downarrow_{\Phi,\Omega} \qquad (H + H')\downarrow_{\Phi,\Omega} = H\downarrow_{\Phi,\Omega} + H'\downarrow_{\Phi,\Omega}$$

$$\varphi[H]\downarrow_{\Phi,\Omega} = \begin{cases} H\downarrow_{\Phi,\Omega} & \text{if } \varphi \in \Phi \\ \varphi[H\downarrow_{\Phi\cup\{\varphi\},\Omega}] & \text{otherwise} \end{cases}$$

$$\psi\langle H\rangle\downarrow_{\Phi,\Omega} = \begin{cases} H\downarrow_{\Phi,\Omega} & \text{if } \psi \in \Phi \\ \psi\langle H\downarrow_{\Phi,\Omega}\rangle & \text{otherwise} \end{cases}$$

$$(\mu h.\, H)\downarrow_{\Phi,\Omega} = \mu h.\, (H'\sigma'\downarrow_{\Phi,\Omega\{(\mu h.H)\Omega/h\}} \sigma)$$

where $H = H'\{h/h_i\}_i$, h_i fresh, $h \notin fv(H')$, and

$$\sigma(h_i) \;=\; (\mu h.H)\Omega\downarrow_{\Phi\cup guard(H'\{\bullet/h_i\}),\Omega}$$

$$\sigma'(h_i) \;=\; \begin{cases} h & \text{if } guard(H'\{\bullet/h_i\}) \subseteq \Phi \\ h_i & \text{otherwise} \end{cases}$$

Intuitively, $H\downarrow_{\Phi,\Omega}$ results from H by eliminating all the redundant safety framings, and all the framings in Φ. The environment Ω is needed to deal with free variables in the case of nested μ-expressions. We feel free to omit the component Ω when unneeded, and, when H is closed, we abbreviate $H\downarrow_{\emptyset,\emptyset}$ with $H\downarrow$.

The last three regularization rules would benefit from some explanation. Consider first a history expression of the form $\varphi[H]$ to be regularized against a set of policies Φ. To eliminate the redundant safety framings, we must ensure that H has neither φ-framings, nor redundant safety framings itself. This is accomplished by regularizing H against $\Phi \cup \{\varphi\}$.

A history expression $\psi\langle H\rangle$ is dealt with by removing the liveness framing if $\psi \in \Phi$ (because there is an outer safety framing enforcing ψ), otherwise the framing remains. Note that we could end up with redundant liveness framings, but they will not prevent us from verifying validity of history expressions.

Consider a history expression of the form $\mu h.H$. Its regularization against Φ and Ω proceeds as follows. Each free occurrence of h in H guarded by some $\Phi' \not\subseteq \Phi$ is unfolded and regularized against $\Phi \cup \Phi'$. The substitution Ω is used to bind the free variables to closed history expressions. Technically, the i-th free occurrence of h in H is picked up by the substitution $\{h/h_i\}$, for h_i fresh. Note also that $\sigma(h_i)$ is computed only if $\sigma'(h_i) = h_i$. As a matter of fact, regularization is a total function, and its definition above can be easily turned into a finitely terminating rewriting system.

Example 16. Consider the history expression $H_0 = \mu h.\, H$, where $H = \alpha + h \cdot h + \varphi[h]$. Then, H can be written as $H'\{h/h_i\}_{i\in 0..2}$, where $H' = \alpha + h_0 \cdot h_1 + \varphi[h_2]$.

Since $guard(H'\{\bullet/h_2\}) = guard(\alpha + h_0 \cdot h_1 + \varphi[\bullet]) = \{\varphi\} \not\subseteq \emptyset$, then:

$$\begin{aligned} H_0\downarrow_\emptyset &= \mu h.\, H'\{h/h_0, h/h_1\}\downarrow_\emptyset \{H_0\downarrow_\varphi / h_2\} \\ &= \mu h.\, \alpha + h \cdot h + \varphi[H_0\downarrow_\varphi] \end{aligned}$$

To compute $H_0\downarrow_\varphi$, note that no occurrence of h is guarded by $\Phi \not\subseteq \{\varphi\}$. Then:

$$H_0\downarrow_\varphi \;=\; \mu h.\, (\alpha + h \cdot h + \varphi[h])\downarrow_\varphi \;=\; \mu h.\, \alpha + h \cdot h + h$$

Since $[\![H_0\downarrow_\varphi]\!] = \{\alpha\}^*$ has no φ-framings, we have that $[\![H_0\downarrow]\!] = \big(\{\alpha\}^*\varphi[\{\alpha\}^*]\big)^*$ has no redundant framings. □

Example 17. Let $H_0 = \mu h.\, H_1$, where $H_1 = \mu h'.\, H_2$, and $H_2 = \alpha + h \cdot \varphi[h']$. Since $guard(H_1\{\bullet/h\}) = \emptyset$, we have that:

$$H_0\downarrow_{\emptyset,\emptyset} \;=\; \mu h.\, (H_1\downarrow_{\emptyset,\{H_0/h\}})$$

Note that H_2 can be written as $H_2'\{h/h_0\}$, where $H_2' = \alpha + h \cdot \varphi[h_0]$. Since $guard(H_2'\{\bullet/h_0\}) = \{\varphi\} \not\subseteq \emptyset$, it follows that:

$$\begin{aligned} H_1\downarrow_{\emptyset,\{H_0/h\}} &= \mu h'.\, H_2'\downarrow_{\emptyset,\{H_0/h, H_1\{H_0/h\}/h'\}} \{H_1\{H_0/h\}\downarrow_{\varphi,\{H_0/h\}} / h_0\} \\ &= \mu h'.\, \alpha + h \cdot \varphi[h_0]\ \{(\mu h'.\, \alpha + H_0 \cdot \varphi[h'])\downarrow_{\varphi,\{H_0/h\}} / h_0\} \\ &= \mu h'.\, \alpha + h \cdot \varphi[H_3\downarrow_{\varphi,\{H/h\}}] \\ &= \alpha + h \cdot \varphi[H_3\downarrow_{\varphi,\{H/h\}}] \end{aligned}$$

where $H_3 = \mu h'.\, \alpha + H_0 \cdot \varphi[h']$, and the last step is possible because the outermost μ binds no variable. Since $guard(\alpha + H_0 \cdot \varphi[\bullet]) = \{\varphi\} \subseteq \{\varphi\}$:

$$H_3\downarrow_\varphi \;=\; \mu h'.\, (\alpha + H_0 \cdot \varphi[h'])\downarrow_\varphi \;=\; \mu h'.\, \alpha + H_0\downarrow_\varphi \cdot h'$$

Since $\{\varphi\}$ contains both $guard(H_1\{\bullet/h\}) = \emptyset$, and $guard(H_2\{\bullet/h'\}) = \{\varphi\}$:

$$\begin{aligned} H_0\downarrow_\varphi &= \mu h.\, (\mu h'.\, \alpha + h \cdot \varphi[h'])\downarrow_\varphi = \ \mu h.\, \mu h'.\, (\alpha + h \cdot \varphi[h'])\downarrow_\varphi \\ &= \mu h.\, \mu h'.\, \alpha + h \cdot h' \end{aligned}$$

Summing up, we have that:

$$\begin{aligned} H_0\downarrow_\emptyset &= \mu h.\, \alpha + h \cdot \varphi[H_3\downarrow_\varphi] \\ H_3\downarrow_\varphi &= \mu h'.\, \alpha + \big(\mu h.\, \mu h'.\, \alpha + h \cdot h'\big) \cdot h' \end{aligned}$$ □

We now establish the following basic properties of regularization, stating its correctness.

Theorem 6. For any history expression H:

- $H\downarrow$ has no redundant safety framings.
- $H\downarrow$ is valid if and only if H is valid.

7.3 From history expressions to Basic Process Algebras

Basic Process Algebras [7] (BPAs) provide a natural characterization of histories. A *BPA process* is given by the following abstract syntax, where ε denotes the terminated process, $\alpha \in \mathsf{Ev} \cup \mathsf{Frm}$, $\cdot$ denotes sequential composition, $+$ represents (nondeterministic) choice, and X is a variable.

Syntax of BPA processes

$$p, p' \ ::= \ \varepsilon \mid \alpha \mid p \cdot p' \mid p + p' \mid X$$

A BPA process p is *guarded* if each variable occurrence in p is within a subexpression $\alpha \cdot q$ of p. We assume a finite set $\Delta = \{X \triangleq p\}$ of definitions: for each variable X, there exists a single, guarded p such that $\{X \triangleq p\} \in \Delta$. As usual, we consider the process $\varepsilon \cdot p$ to be equivalent to p.

The operational semantics of BPAs is given by the following labelled transition system, in the SOS style. We denote with $[\![p_0, \Delta]\!]$ the set of the strings that label finite computations, i.e. $\{(a_i)_i \,|\, p_0 \xrightarrow{a_1} \cdots \xrightarrow{a_i} p_i\}$. Note that we only consider finite computations, since our histories are always such.

Operational Semantics of BPA processes

$$\frac{}{\alpha \xrightarrow{\alpha} \varepsilon} \qquad \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'} \qquad \frac{p \xrightarrow{\alpha} p'}{p \cdot q \xrightarrow{\alpha} p' \cdot q} \qquad \frac{p \xrightarrow{\alpha} p' \quad X \triangleq p \in \Delta}{X \xrightarrow{\alpha} p'}$$

We now introduce a mapping from history expressions to BPAs, in the line of [4, 44]. Again, note that there is no need for transforming planned selections into BPAs, because we are only interested in the validity of history expressions with no selections.

The mapping takes as input a history expression H and an injective function Θ from history variables h to BPA variables X, and it outputs a BPA process p and a finite set of definitions Δ. Without loss of generality, we assume that all the variables in H have distinct names.

The rules that transform history expressions into BPAs are rather natural. Events, variables, concatenation and choice are mapped into the corresponding BPA counterparts. A history expression $\varphi[H]$ is mapped to the BPA for H, surrounded by the opening and closing of the φ-framing; similarly for $\psi\langle H \rangle$. A history expression $\mu h.H$ is mapped to a fresh BPA variable X, bound to the translation of H in the set of definitions Δ. To avoid the problem of unguarded BPA processes, we assume a standard preprocessing step, that inserts a dummy event before each unguarded occurrence of a variable in a history expression. Dummy events are eventually discarded before the verification phase.

Mapping history expressions to BPAs

$$
\begin{array}{rcl}
BPA(\varepsilon,\Theta) & = & (\varepsilon,\emptyset)\\
BPA(\alpha,\Theta) & = & (\alpha,\emptyset)\\
BPA(h,\Theta) & = & (\Theta(h),\emptyset)\\
BPA(H_0\cdot H_1,\Theta) & = & (p_0\cdot p_1,\Delta_0\cup\Delta_1),\text{where } BPA(H_i,\Theta)=(p_i,\Delta_i)\\
BPA(H_0+H_1,\Theta) & = & (p_0+p_1,\Delta_0\cup\Delta_1),\text{where } BPA(H_i,\Theta)=(p_i,\Delta_i)\\
BPA(\varphi[H],\Theta) & = & ([_\varphi\cdot p\cdot]_\varphi,\Delta),\text{where } BPA(H,\Theta)=(p,\Delta)\\
BPA(\psi\langle H\rangle,\Theta) & = & (\langle_\psi\cdot p\cdot\rangle_\psi,\Delta),\text{where } BPA(H,\Theta)=(p,\Delta)\\
BPA(\mu h.H,\Theta) & = & (X,\Delta\cup\{X\triangleq p\}),\text{where } BPA(H,\Theta\{X/h\})=(p,\Delta)
\end{array}
$$

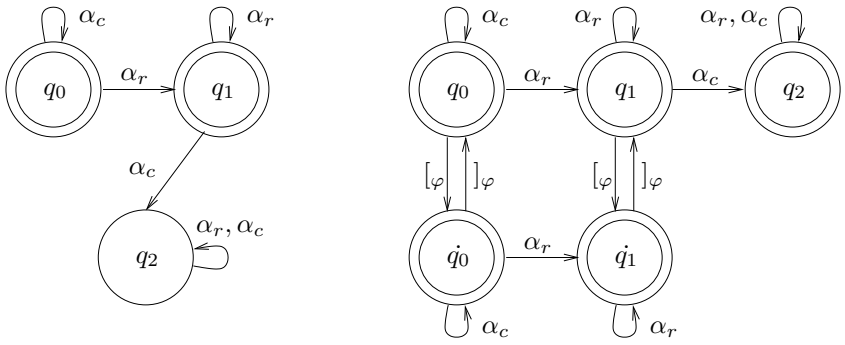
We now state the correspondence between history expressions and BPAs. The prefixes of the histories generated by a history expression H (i.e. $[\![H]\!]^\partial$, where we omit the plan π because immaterial) are all and only the strings that label the finite computations of $BPA(H)$.

Lemma 7. $[\![H]\!]^\partial = [\![BPA(H)]\!]$.

7.4 Finite State Automata for validity

Given a policy φ, we are interested in defining a formula $\varphi_{[]}$ and a formula $\varphi_{\langle\rangle}$ to be used in verifying the validity of a history η with respect to security policies within their framings.

As mentioned above, since our histories are always finite and our properties are regular, FSA suffice for defining the safety and liveness properties we are using. As an example, let φ be the policy saying that no event α_c can occur after an α_r (see Section 2). The finite state automata A_φ and $A_{\varphi_{[]}}$ are shown below, which define φ and $\varphi_{[]}$, respectively. It is immediate checking that the history $[_\varphi\alpha_r]_\varphi\alpha_c$ is accepted by $A_{\varphi_{[]}}$, while $\alpha_r[_\varphi\alpha_c]_\varphi$ is not.



Intuitively, the automaton $A_{\varphi_{[]}}$ is partitioned into two layers. The first layer is a copy of A_φ, where all the states are final. This models the fact that we are outside the scope of φ, i.e. the history leading to any state in this layer has balanced safety framings of φ (or none). The second layer is reachable from

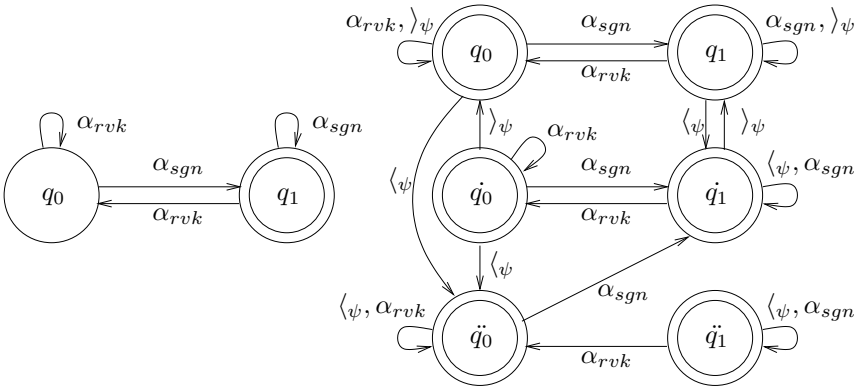
the first one when opening a safety framing for φ, while closing the framing gets back. The transitions in the second layer are a copy of those connecting final states in A_φ. Consequently, the states in the second layer are exactly the final states in A_φ. Since $A_{\varphi_{[]}}$ is only concerned with the verification of φ, the transitions for opening and closing safety framings $\varphi' \neq \varphi$, as well as those for liveness framings ψ, are rendered as self-loops.

We require that the history η to be verified against $\varphi_{[]}$ has no redundant safety framings, i.e. η has been regularized. Hereafter, let the formula φ be defined by the finite state automaton $A_\varphi = (\Sigma, Q, q_0, \rho, F)$, which we assume to have a distinguished non-final sink state. Then, the property $\varphi_{[]}$ is defined through the finite state automaton $A_{\varphi_{[]}}$ defined below.

Finite state automaton for $\varphi_{[]}$

$$
\begin{aligned}
A_{\varphi_{[]}} &= (\Sigma', Q', q_0, \rho', F') \\
\Sigma' &= \Sigma \cup \{\, [_{\varphi'},]_{\varphi'}, \langle_\psi, \rangle_\psi \mid \varphi', \psi \in \mathsf{Pol} \,\} \\
Q' &= F' = Q \cup \{\, \dot{q} \mid q \in F \,\} \\
\rho' &= \rho \cup \{\, (q, [_\varphi, \dot{q}) \mid q \in F \,\} \cup \{\, (\dot{q},]_\varphi, q) \mid q \in Q \,\} \\
&\quad \cup \{\, (\dot{q}_i, \alpha, \dot{q}_j) \mid (q_i, \alpha, q_j) \in \rho \wedge q_j \in F \,\} \\
&\quad \cup \{\, (q, [_{\varphi'}, q) \cup (q,]_{\varphi'}, q) \mid q \in Q' \wedge \varphi' \neq \varphi \,\} \\
&\quad \cup \{\, (q, \langle_\psi, q) \cup (q, \rangle_\psi, q) \mid q \in Q' \,\}
\end{aligned}
$$

Likewise, we introduce in a while the finite state automaton $A_{\psi_{\langle\rangle}}$ that defines the property $\psi_{\langle\rangle}$. However, in this case we allow histories to have redundant framings. As an example, let ψ be the policy saying that α_{sgn} eventually occurs with no subsequent α_{rvk} (see Section 2). The finite state automata A_ψ and $A_{\psi_{\langle\rangle}}$ for ψ and $\psi_{\langle\rangle}$, respectively, are shown below. The history $\alpha_{rvk}\langle_\psi \alpha_{sgn}\rangle_\psi$ is accepted by $A_{\psi_{\langle\rangle}}$, while $\alpha_{sgn}\alpha_{rvk}\langle_\psi \alpha_{rvk}\rangle_\psi$ is not. Also, note that $\langle_\psi \alpha_{sgn}\alpha_{rvk}\rangle_\psi$ is accepted, and indeed it represents a computation that leaves the liveness framing as soon as α_{sgn} has occurred.



The automaton $A_{\psi_{\langle\rangle}}$ consists of three layers. The first layer is a copy of A_ψ, and it represents being outside of the liveness framing $\psi\langle\cdots\rangle$. The second and

the third layer model being inside the framing. If you are in the second layer, then you have already found a history that satisfies the property ψ, while if you are in the third layer, you are still looking for. Suppose now that a new framing $\psi\langle\cdots\rangle$ is opened when you are in the second layer, so this is a redundant liveness framing. If you were in a state that was final in A_ψ, then you remain in the second layer; otherwise, you go to the corresponding state in the third layer. If the redundant framing is opened when you are in the third layer, then you stay there. If a framing is closed when you are in the second layer, then you can go back to the first layer, but if you are in the third layer, then you get stuck. The automaton $A_{\psi\langle\,\rangle}$ is defined in the following table.

Finite state automaton for $\psi_{\langle\,\rangle}$

$$
\begin{aligned}
A_{\psi\langle\,\rangle} &= (\Sigma', Q', q_0, \rho', F') \\
\Sigma' &= \Sigma \cup \{\, [_\varphi,]_\varphi, \langle_{\psi'}, \rangle_{\psi'} \mid \varphi, \psi' \in \mathsf{Pol} \,\} \\
Q' &= F' = Q \cup \{\, \dot{q}, \ddot{q} \mid q \in Q \,\} \\
\rho' &= \rho \cup \{\, (q, \langle_\psi, \dot{q}) \mid q \in F \,\} \cup \{\, (q, \langle_\psi, \ddot{q}) \mid q \notin F \,\} \\
&\quad \cup \{\, (\dot{q}, \langle_\psi, \dot{q}) \mid q \in F \,\} \cup \{\, (\dot{q}, \langle_\psi, \ddot{q}) \mid q \notin F \,\} \cup \{\, (\ddot{q}, \langle_\psi, \ddot{q}) \mid q \in Q \,\} \\
&\quad \cup \{\, (\dot{q_i}, \alpha, \dot{q_j}) \mid (q_i, \alpha, q_j) \in \rho \,\} \\
&\quad \cup \{\, (\ddot{q_i}, \alpha, \dot{q_j}) \mid (q_i, \alpha, q_j) \in \rho \wedge q_j \in F \,\} \\
&\quad \cup \{\, (\ddot{q_i}, \alpha, \ddot{q_j}) \mid (q_i, \alpha, q_j) \in \rho \wedge q_j \notin F \,\} \\
&\quad \cup \{\, (\dot{q}, \rangle_\psi, q), (q, \rangle_\psi, q) \mid q \in Q \,\} \\
&\quad \cup \{\, (q, \langle_{\psi'}, q),\ (q, \rangle_{\psi'}, q) \mid q \in Q' \wedge \psi' \neq \psi \,\} \\
&\quad \cup \{\, (q, [_\varphi, q),\ (q,]_\varphi, q) \mid q \in Q' \,\}
\end{aligned}
$$

Although the policies enforced by the security framings can always inspect the whole past history, we can easily limit the scope from the side of the past. It suffices to mark in the history the point in time β_φ from which checking a policy φ has to start. The corresponding automaton ignores all the events before β_φ, and then behaves like the standard automaton enforcing φ.

We now relate validity of histories with the formulae $\varphi_{[\,]}$ and $\psi_{\langle\rangle}$ for the policies φ, ψ spanning over η.

Lemma 8. Let η be a history with no redundant safety framings. Then, η is valid if and only if $\eta \models \varphi_{[\,]}$ and $\eta \models \psi_{\langle\rangle}$ for all φ, ψ such that $[_\varphi, \langle_\psi \in \eta$.

Since finite state automata are closed under intersection, a valid history η is accepted by the intersection of the automata $A_{\varphi_{[\,]}}$ and $A_{\psi_{\langle\rangle}}$, for all φ, ψ in η.

Validity of a closed history expression H with no planned selections can be decided by showing that the BPA generated by the regularization of H satisfies the given regular formula. Together with Theorem 2, the execution of an expression in our calculus never violates security if its effect is verified valid. Thus we are dispensed from using an execution monitor to enforce the safety properties, and we additionally guarantee the liveness properties on demand.

Theorem 9. A history expression H with no planned selections is valid iff:

$$[\![BPA(H\downarrow)]\!] \models \bigwedge_{\varphi\in H} \varphi_{[\,]} \wedge \bigwedge_{\psi\in H} \psi_{\langle\,\rangle}$$

8 Discussion

The essence of the SOC approach resides in a programming paradigm where loosely coupled, reusable components can be invoked and composed by clients. A key aspect is the use of the "WS-*" standards, e.g. WSDL for service description, WS-BPEL for service orchestration, WS-CDL for choreography, WS-Security for security policies, etc. All these standards feature a call-by-name mechanism for service invocation. Recent initiatives aim at relaxing this *a priori* agreement on the syntax of service interactions, and instead rely on an informal semantics, based on ontologies. The semantic-web initiative is an example of this approach [8].

Our type and effects for services extend the WSDL notion of published interfaces. Besides the standard WSDL attributes, our effects add semantic information about a service behaviour, in the spirit of WSLA proposal [33]. This additional attribute, namely a history expression, formally is a context-free grammar that over-approximates the run-time behaviour of services. Also, this extension impacts on service discovery and selection. In the current Web service technology, these operations use UDDI registries of services or matchmaking algorithms based on semantic ontologies. Our notion of call-by-contract suggests instead to select a service with a matchmaking algorithm based on formal semantic abstractions.

Call-by-contract makes the picture complex, because several components need to cooperate in a trusted manner. First, registries are assumed to be trusted, in that they certify the agreement between a published type and effect and the actual behaviour of the service considered. Formally, a registry statically analyses service code to infer its interface, by the type and effect system of Section 5. Also, the infrastructure running services must only execute the published ones, and code updates require certification before use. The actual mechanism to enforce these aspects of trust are outside the scope of the present paper, which instead concentrates on certification and planning. In our abstract model we assume the interconnecting network is reliable, and we do not address here the classical security problem of confidentiality, identity and availibility.

Our proposal for a semantically-based orchestration outlines the design of an infrastructure for secure service composition. The call-by-contract invocation mechanism adds a further layer to the standard remote procedure call. Before starting the execution of a service, the orchestrator collects the relevant components, by inquiring the (possibly distributed) registries. The plans provided by the orchestrator resolve all the requests in the initiator service, as well as those in the invoked services. This mechanism differs from the standard one, e.g. WS-BPEL, with the advantage of offering a way to enforce all the security policies imposed. The trustfulness of the orchestrator (Theorem 10) follows from our formal approach, in particular from the soundness of the type and effect system (Theorem 2), and the correctness of planning (Theorem 4) and of verification (Theorem 9).

9 Related work

Process calculi techniques have been used to study the foundation of services. The main goal of some of these proposals, e.g. [25, 17, 30, 35] is to formalise various aspects of standards for the description, execution and orchestration of services (WSDL, SOAP and WS-BPEL). The Global Calculus [20] addresses the problem of relating orchestration and choreography. As a matter of fact, our λ^{req} builds over the standard service infrastructure the above calculi formalise. Indeed, our call-by-contract supersedes standard invocation mechanisms and allows for verified planning.

The secure composition of components has been the main concern underlying the design of Sewell and Vitek's box-π [43], an extension of the π-calculus that allows for expressing safety policies in the form of *security wrappers*. These are programs that encapsulate a component to control the interactions with other (possibly untrusted) components. The calculus is equipped with a type system that statically captures the allowed causal information flows between components. Our safety framings are closely related to wrappers, but in [43] there is no analog of our liveness framings.

Gorla, Hennessy and Sassone [29] consider a calculus for mobile agents which may migrate between sites in a controlled manner. Each site has a *membrane*, representing both a security policy and a classification of external sites with respect to their levels of trust. A membrane guards the incoming agents before allowing them to execute. Three classes of membranes are studied, the most complex being the class of policies enforceable by finite state automata. When an agent comes from an untrusted site, *all* its code must be checked. Instead, an agent coming from a trusted site must only provide the destination site with a *digest* of its behaviour, so allowing for more efficient checks.

A different approach is Cook and Misra's Orc [38], a programming model for structured orchestration of services. The basic computational entities orchestrated by Orc expressions are sites. A site computation can start other orchestrations, locally store the effects of a computation, and make them visible to clients. Orc provides three basic composition operators, that can be used to model some common workflow patterns, identified by Van der Aalst et al. [23].

Another solution to planning service composition has been proposed in [36], where the problem of achieving a given composition goal is expressed as a constraint satisfaction problem.

From a technical point of view, the work of Skalka and Smith [44] is the closest to this paper. We share with them the use of a type and effect system and that of model checking validity of effects. In [44], a static approach to history-based access control is proposed. The λ-calculus is enriched with access events and local checks on the past event history. Local checks make validity a regular property, so regularization is unneeded. The programming model and the type system of [44] allow for access events parametrized by constants, and for let-polymorphism. We have omitted these features for simplicity, but they can be easily recovered by using similar techniques.

A related line of research addresses the issue of modelling and analysing resource usage. Igarashi and Kobayashi [32] introduce a type systems to check whether a program accesses resources according to a user-defined usage policy. Our model is less general than the framework of [32], but we provide a

static verification technique, while [32] does not. Colcombet and Fradet [21] and Marriot, Stuckey and Sulzmann [37] mix static and dynamic techniques to transform programs in order to make them obey a given safety property. Besson, de Grenier de Latour and Jensen [9] tackle the problem of characterizing when a program can call a stack-inspecting method while respecting a global security policy. Compared to [21, 37, 9], our programming model allows for local policies, while the other only considers global ones.

Recently, increasing attention has been devoted to express service contracts as behavioural (or session) types. These synthetise the essential aspects of the interaction behaviour of services, while allowing for efficient static verification of properties of composed systems. Session types [31] have been exploited to formalize compatibility of components [47] and to describe adaptation of web services [19]. Security issues have been recently considered in terms of session types, e.g. in [15], which proves the decidability of type-checking in an extension of the π-calculus with session types and correspondence assertions [52]. Our λ^{req} has no explicit primitive for sessions. However, they can be suitably encoded, via higher-order functions.

Other papers have proposed type-based methodologies to check security properties of distributed systems. For instance, Gordon and Jeffrey [28] use a type and effect system to prove authenticity properties of security protocols. Web service authentication has been recently modelled and analysed in [10, 11] through a process calculus enriched with cryptographic primitives. In particular, [12] builds security libraries using the WS-Security policies of [3]. These libraries are then mechanically analysed with ProVerif [13].

10 Conclusions

We have enriched the λ-calculus with primitives to express service composition under security constraints, as a first step towards the design of programming constructs to publish, select and compose services on top of a semantic theory. The security requirements are safety and liveness properties over (an abstraction of) execution histories. These properties have a local scope, possibly nested.

We have used a type and effect system to extract from a given program a history expression, i.e. a safe approximation of its run-time behaviour that also includes plans, i.e. which services can be selected to serve requests. Indeed, plans drive service composition, so to achieve the task assigned while respecting the security constraints. A history expression is valid (under a plan π) when it represents execution histories (driven by π) that never violate the security policies within their scope.

To verify the validity of history expressions, we have exploited model checking over Basic Process Algebras and Finite State Automata. However, nesting of scopes makes validity non-regular, and has required us to transform history expressions (technically, to linearize and regularize them) so that model checking is feasible. When a history expression of a program e is verified valid under a plan π, then e will never go wrong. Therefore, at run-time it suffices to follow the plan π to enforce the required safety and liveness properties without resorting to any run-time monitoring.

The main result of this paper is resumed by the following theorem:

Theorem 10. Let $\Gamma, H \vdash e : \tau$, for e closed, and let $H' = \{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$ be the linearization of H. If H_i is verified valid for some $i \in 1..k$, then the execution of e with plan π_i will never go wrong.

Acknowledgments

We thank Roberto Zunino, Luís Caires and the anonymous referees for their insightful comments. This research has been partially supported by EU-FETPI Global Computing Project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers).

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[2] S. Anderson et al. *Web Services Trust Language (WS-Trust)*, 2005. `http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf`.

[3] B. Atkinson et al. *Web Services Security (WS-Security)*, 2002. `http://www.oasis-open.org`.

[4] M. Bartoletti, P. Degano, and G.L. Ferrari. History based access control with local policies. In *Proc. Foundations of Software Science and Computation Structures (Fossacs)*, volume 3441 of *Springer LNCS*, 2005.

[5] M. Bartoletti, P. Degano, and G.L. Ferrari. Planning and verifying service composition. Technical Report TR-07-02, Dip. Informatica, Univ. of Pisa, 2007. `http://compass2.di.unipi.it/TR/Files/TR-07-02.pdf.gz`.

[6] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Enforcing secure service composition. In *Proc. 18th Computer Security Foundations Workshop (CSFW)*, 2005.

[7] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37, 1985.

[8] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.

[9] F. Besson, T. de Grenier de Latour, and T. Jensen. Interfaces for stack inspection. *Journal of Functional Programming*, 15(2), 2005.

[10] K. Bhargavan, R. Corin, C. Fournet, and A.D. Gordon. Secure sessions for web services. In *Proc. ACM Workshop on Secure Web Services*, 2004.

[11] K. Bhargavan, C. Fournet, and A.D. Gordon. A semantics for web services authentication. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.

[12] K. Bhargavan, C. Fournet, and A.D. Gordon. Verified reference implementations of WS-security protocols. In *WS-FM*, volume 4184 of *Springer LNCS*, 2006.

[13] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop (CSFW)*, 2001.

[14] B. Bloch et al. Web services business process execution language, version 2.0. Technical report, TC OASIS, 2005. `http://www.oasis-open.org`.

[15] E. Bonelli, A. Compagnoni, and E. Gunter. Typechecking safe process synchronization. In *Proc. Foundations of Global Ubiquitous Computing*, volume 138(1) of *ENTCS*, 2005.

[16] D. Booth et al. *Web Service Description Language (WSDL), Version 2.0*, 2006. `http://www.w3.org/TR/wsdl20-primer`.

[17] M. Boreale et al. SCC: a service centered calculus. In *WS-FM*, volume 4184 of *Springer LNCS*, 2006.

[18] D. Box et al. *Simple Object Access Protocol (SOAP) 1.1.* W3C Note, 2000. `http://www.w3.org/TR/soap`.

[19] A. Brogi, C. Canal, and E. Pimentel. Behavioural types and component adaptation. In *Proc. Algebraic Methodology and Software Technology (AMAST)*, volume 3116 of *Springer LNCS*, 2004.

[20] M. Carbone, K. Honda, and N. Yoshida. Structured global programming for communicating behaviour. In *European Symposium in Programming Languages (ESOP)*, volume to appear, 2007.

[21] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2000.

[22] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarane. The next step in web services. *Communications of the ACM*, 46(10), 2003.

[23] W. Van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1), 2003.

[24] J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proc. 19th Int. Colloquium on Trees in Algebra and Programming*, volume 787 of *Springer LNCS*, 1994.

[25] G.L. Ferrari, R. Guanciale, and D. Strollo. JSCL: A middleware for service coordination. In *Proc. FORTE*, volume 4229 of *Springer LNCS*, 2006.

[26] F. C. Gärtner. Revisiting liveness properties in the context of secure systems. In *Proc. FASec*, volume 2629 of *Springer LNCS*, 2002.

[27] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, 1986.

[28] A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, 2002.

[29] D. Gorla, M. Hennessy, and V. Sassone. Security policies as membranes in systems for global computing. *Logical Methods in Computer Science*, 1(3), 2005.

[30] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A calculus for service oriented computing. In *Proc. Service-Oriented Computing (ICSOC)*, volume 4294 of *Springer LNCS*, 2006.

[31] K. Honda, V. Vansconcelos, and M. Kubo. Language primitives and type discipline for structures communication-based programming. In *Proc. European Symposium on Programming Languages (ESOP)*, volume 1381 of *Springer LNCS*, 1998.

[32] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[33] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1), 2003.

[34] R. Khalaf, N. Mukhi, and S. Weerawarana. Service oriented composition in BPEL4WS. In *Proc. World Wide Web Conference (WWW)*, 2003.

[35] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *European Symposium in Programming Languages (ESOP)*, volume to appear, 2007.

[36] A. Lazovik, M. Aiello, and R. Gennari. Encoding requests to web service compositions as constraints. In *Proc. Principles and Practice of Constraint Programming (CP)*, volume 3709 of *Springer LNCS*, 2005.

[37] K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proc. Asian Programming Languages Symposium (APLAS)*, volume 2895 of *Springer LNCS*, 2003.

[38] J. Misra. A programming model for the orchestration of web services. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, 2004.

[39] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, 1999.

[40] M. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Proc. Web Information Systems Engineering (WISE)*, 2003.

[41] M. Papazoglou and D. Georgakopoulos. Special issue on service oriented computing. *Communications of the ACM*, 46(10), 2003.

[42] F.B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1), 2000.

[43] P. Sewell and J. Vitek. Secure composition of untrusted code: box-π, wrappers and causality types. *Journal of Computer Security*, 11(2), 2003.

[44] C. Skalka and S. Smith. History effects and verification. In *Proc. Asian Programming Languages Symposium (APLAS)*, volume 3302 of *Springer LNCS*, 2004.

[45] M. Stal. Web services: Beyond component-based computing. *Communications of the ACM*, 55(10), 2002.

[46] J.P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 2(111), 1994.

[47] A. Vallecillo, V. Vansconcelos, and A. Ravara. Typing the behaviours of objects and components using session types. In *Proc. of FOCLASA*, 2002.

[48] Vedamuthu et al. *Web Services Policy Framework (WS-Policy)*, 2006. `http://www.w3.org/TR/ws-policy`.

[49] W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6), 2003.

[50] W3C. *UDDI Technical White Paper*, 2000.

[51] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

[52] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, 1993.