Enforcing Secure 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 software. We extend the λ-calculus with primitives for 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 of execution histories. The actual histories that can occur at runtime are over-approximated by a type and effect system. These approximations are model-checked to verify policy framings within their scopes. This allows for removing any runtime execution monitor, and for selecting those services that match the security requirements.

1. Introduction

Service-oriented computing (SOC) is emerging as an evolutionary paradigm to design open-ended distributed applications [28, 27, 17]. In this paradigm, applications are built by assembling together independent computational units, called *services*, distributed over a network infrastructure. A service is a stand-alone component available over the network through standard interaction mechanisms to meet interoperability demands. An important, challenging aspect is that such services are open, in that they are built with little or no knowledge about their operating environment, their clients, and other services. 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, 32, 36] built upon XML technologies are possibly the most illustrative and well developed example of the SOC paradigm.

So-called *orchestration* languages offer mechanisms to coordinate the execution of services according to expressive interaction patterns. Indeed, XML-based technologies already provide a variety of mechanisms to describe, discover, invoke and coordinate the behaviour of web services [15, 12, 3, 37]. There are also several standards for defining and enforcing non-functional requirements of services, e.g. WS-Security [4], WS-Trust [2] and WS-Policy [13] among the others. The SOC scenario is also interesting from a theoretical point of view. For instance, web service authentication have been formally modelled and analyzed in [9, 10] by exploiting process calculi enriched with cryptographic primitives.

However, composing services in a secure manner is still a major challenge, from both viewpoints of foundational models and programming language design. Indeed, services may be offered by different providers, which only partially trust each other. On the one hand, the provider has to guarantee a 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, the client may want to protect its sensible data from the service invoked.

In this paper, we tackle the problem of modelling composition of services in the presence of security constraints. In our proposal, a security policy is a property over a statically determined abstraction of the behaviour of a service. We are interested in enforcing *safety* and *liveness* properties, that have shown effective to reason about concurrent systems and security. For example, history-based access control can be modelled in terms of safety properties (e.g. as in [6, 31]), while liveness properties can be exploited to formalize denial-of-service and brute-force attacks of cryptographic keys [19]. More generally, a suitable combination of safety and liveness properties can express contract agreements in terms of an *enforce and guarantee* paradigm.

After an example in Section 2, in Section 3 we introduce a typed extension of the λ-calculus to describe services as program expressions, and to compose them under security constraints. Our calculus assumes a set of primitive *access events*, that abstract from activities with possible security concerns. The security poli-

cies 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. We shall exploit a static analysis technique to ensure the desired behaviour, checking both kinds of properties at compile-time. Note that, while safety properties can be enforced by an execution monitor, liveness properties cannot [29]. Also, liveness cannot be reduced to safety in general. Indeed, 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 consideration above further supports the use of static techniques.

Our static analysis over-approximates program behaviour through *history expressions*. These represent sequences of access events together with the scope of safety and liveness framings. A history expression is *valid* when all the histories it denotes respect the security policies – according to the scopes and the intended semantics of safety and liveness framings.

We model services as expressions with a functional type $\tau \xrightarrow{H} \tau'$. Intuitively, when supplied with an argument of type τ, the service evaluates to a value of type τ', and it generates a history belonging to the statically inferred history expression H.

A service invocation is modelled by an expression $\texttt{req}\ \tau \xrightarrow{\varphi[\,],\psi\langle\rangle} \tau'$. It means that we are looking for a service of type $\tau \xrightarrow{H} \tau'$, where H satisfies both the safety framing enforcing φ and the liveness framing guaranteeing ψ. Intuitively, the liveness framing $\psi\langle\rangle$ can be seen as the duties the invoked service must fulfill. Instead, the safety framing $\varphi[\,]$ says how the caller protects itself from the service.

We assume that a typed service $e : \tau \xrightarrow{H} \tau'$ is published in a trusted repository, which collects the type, in particular guaranteeing that H represents all the possible behaviour of e.

Our second technical contribution is the definition in Section 4 of a type and effect system [20, 26, 33] that extracts from a well-typed program a sound approximation of its possible runtime histories, represented by a history expression. Remarkably enough, we exploit the information about types and effects in order to detect and select those services only, that match the security constraints required by the service invocation. In this way we model the fulfillment of a contract agreement. Our type and effect system enjoys the following type safety property: if the history expression of a program is valid, then there will be no runtime errors. Also, only the services that respect the security properties required will be chosen. Therefore, execution monitoring is needed no longer.

Our third contribution is in Section 5, where we develop a static technique to verify validity of history expressions. This is done by model checking Basic Process Algebras (BPAs) with Büchi automata. A history expressions is rendered as a BPA, while an automaton models the security properties subject to the scope of safety and liveness framings. 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. Nevertheless, we extend the proposal in [6], which only considered safety framings and no service requests. Actually, we transform history expressions so to make model checking feasible through specially-tailored Büchi automata.

Summing up, a user can invoke services and put over them constraints that enforce and guarantee security. Our type and effect system predicts the actual behaviour of programs, including the security framings they must respect. Validity of behaviour is model-checked over BPAs and Büchi automata. If the effect of a program e is proved valid, then e can be executed with no runtime monitoring and it will never go wrong.

2. A motivating example

To illustrate our approach, consider a simple certification service c 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*.

To assert its willing to provide its 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 (represented by α_{rvk}).

Formally, our policy providers are characterized by a distinguished event α_p. They return a safety policy φ, in the form of a closure $\lambda x.\,\varphi[x]$, since policies are not first class objects in our model. The policy φ has to be enforced in the subsequent execution of c, i.e. $(\lambda x.\,\varphi[x])e$ will evolve to $\varphi[e]$. This paradigm can be seen as a form of *dynamic sandboxing*.

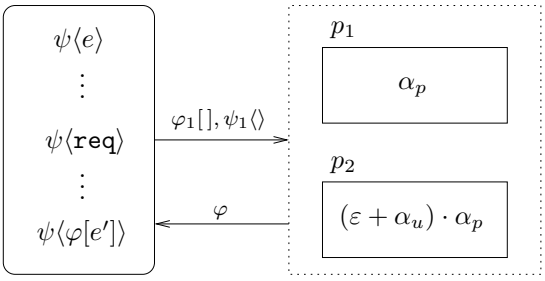
The certification service requests a policy provider through the expression:

$$\texttt{req}\ \tau \xrightarrow{\varphi_1[\,],\psi_1\langle\rangle} \tau'$$

where 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 types τ and τ' are immaterial in this example.

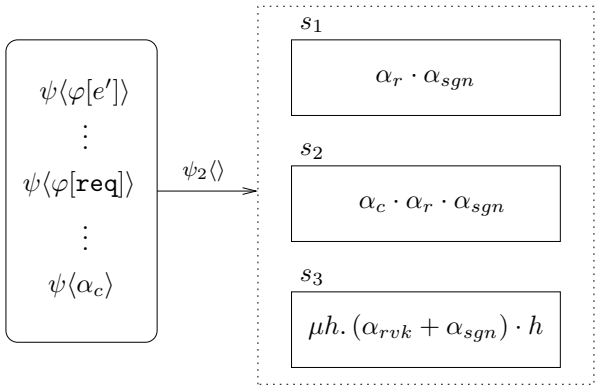
The leftmost part in the diagram below highlights the evolution of the certification service. The rightmost part shows the two policy providers p_1 and p_2 available. The arrows represent the service invocation and the delivered safety policy.



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 operation will be performed. So p_1 can be chosen, because α_p satisfies both φ_1 and ψ_1. 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. Thus, p_2 can generate the histories α_p and $\alpha_u\alpha_p$. The second history does not respect the safety constraint φ_1, so p_2 will be rejected by our static machinery.

The safety policy delivered by p_1 states that connecting to the network (α_c) is prevented after a read (α_r) of local data.

The next diagram depicts the invocation of the external parties. The required guarantee is expressed by the liveness property ψ_2, saying that the certification service will eventually receive back a signed contract. There are three such parties. The service s_1 first reads the contract file (α_r) and then signs it; the service s_2 behaves as s_1 after connecting to the network (α_c); the last service s_3 is a loop of either action of revoking or signing (recursion is written through the μ operator).



Both history expressions $\alpha_r\alpha_{sgn}$ and $\alpha_c\alpha_r\alpha_{sgn}$ clearly satisfy the requested agreement ψ_2, while the history expression of s_3 does not, because the infinite history $(\alpha_{rvk})^\omega$ is possible. Then, the service request can be non-deterministically solved by composing the certification service with either s_1 or s_2. After completion of the service, the safety framing $\varphi[\cdots]$ is left, so the certification service can now connect to the network. The resulting global history expression is obtained by composition:

$$H \;=\; \psi\langle\alpha_p \cdot \varphi[\alpha_r \cdot \alpha_{sgn} + \alpha_c \cdot \alpha_r \cdot \alpha_{sgn}] \cdot \alpha_c\rangle$$

Now we can model-check H, that turns out to be valid. Indeed, both histories $\alpha_p\alpha_r\alpha_{sgn}\alpha_c$ and $\alpha_p\alpha_c\alpha_r\alpha_{sgn}\alpha_c$ do satisfy the properties ψ and φ within the scopes localized by the framings (the last α_c is permitted, because outside of the scope of φ). At this point the service composition can be executed without resorting to any execution monitor.

Assume now to have a fourth service s_4 with history expression $\alpha_r \cdot \alpha_{sgn} \cdot (\varepsilon + \alpha_c)$. Also s_4 satisfies the request agreement ψ_2, but the composed global history expression is no longer valid. Indeed, the history $\alpha_p\alpha_r\alpha_{sgn}\alpha_c\alpha_c$ violates security, because the leftmost α_c in inside the privacy policy φ. A further refinement of our machinery will allow us to efficiently discard services like s_4; note that a trivial static way consists in checking all possible compositions of services.

3. Programming model

To study secure service composition in a pure framework, we consider an explicitly typed, call-by-value λ-calculus enriched with security policies and service requests. An *access event* $\alpha \in \Sigma$ abstracts from a security critical operation (e.g. writing a file, opening a socket connection). A *(plain) history* η is a sequence of access events. A *security policy* $\varphi \in \Pi$ 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 be eventually satisfied while evaluating e. A service request takes the form:

$$\mathtt{req}_\ell\,\tau \xrightarrow{\varphi[\,],\psi\langle\rangle} \tau'$$

Operationally, this results in searching a finite global service environment $\Lambda = \{e_i : \tau_i \xrightarrow{H_i} \tau_i'\}$ for a service with a functional type $\tau \xrightarrow{H} \tau'$, such that the effect H respects the safety framing $\varphi[\cdots]$ and the liveness framing $\psi\langle\cdots\rangle$. The label ℓ will be exploited by our static analysis to optimize the service lookup mechanism (see Section 4).

3.1. Syntax

The syntax of our calculus follows. We omit the definition of policies φ, ψ and of guards b, 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 $e' = \lambda_z x : \tau.\, e$ stands for e' itself within e).

Expressions

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

The values v of our calculus are the variables and the abstractions. We write $*$ for a fixed, closed, event-free value, and $\lambda.\, e$ for $\lambda x.\, e$, for x not free in e. The following abbreviation is standard: $e; e' = (\lambda.\, e')\, e$.

3.2. Operational semantics

We define the behaviour of expressions through the following small-step operational semantics. The configurations are pairs η, e where the history η is a sequence of access events. A transition $\eta, e \to \eta', e'$ means that, starting from a history η, the expression e may evolve to e', possibly extending the history to η'. An expression is initially evaluated starting from the empty history ε. We write $\eta \models \varphi$ when the history η obeys the policy φ. We assume as given a total function $\mathcal{B}$ that evaluates the guards in conditionals.

Reduction rules

$$\frac{\eta, e_1 \to \eta', e_1'}{\eta, e_1 e_2 \to \eta', e_1' e_2} \qquad \frac{\eta, e_2 \to \eta', e_2'}{\eta, v e_2 \to \eta', v e_2'}$$

$$\frac{}{\eta, (\lambda_z x : \tau.\, e) v \to \eta, e\{v/x, \lambda_z x : \tau.\, e/z\}}$$

$$\frac{}{\eta, \mathtt{if}\, b\, \mathtt{then}\, e_{true}\, \mathtt{else}\, e_{false} \to \eta, e_{\mathcal{B}(b)}}$$

$$\frac{}{\eta, \alpha \to \eta\alpha, *} \qquad \frac{e : \tau \xrightarrow{H} \tau' \in \Lambda \quad \varphi[H], \psi\langle H\rangle \text{ valid}}{\eta, \mathtt{req}\ \tau \xrightarrow{\varphi[\,],\psi\langle\rangle} \tau' \to \eta, e}$$

$$\frac{\eta, e \to \eta', e' \quad \eta, \eta' \models \varphi}{\eta, \varphi[e] \to \eta', \varphi[e']} \qquad \frac{\eta \models \varphi}{\eta, \varphi[v] \to \eta, v}$$

$$\frac{\eta, e \to \eta', e' \quad \eta \not\models \psi}{\eta, \psi\langle e\rangle \to \eta', \psi\langle e'\rangle} \qquad \frac{\eta \models \psi}{\eta, \psi\langle e\rangle \to \eta, e}$$

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 $\lambda_z x : \tau.\, e$ replaces the self variable z after the substitution, so giving an explicit copy-rule semantics to recursive functions. A conditional $\mathtt{if}\, b\, \mathtt{then}\, e\, \mathtt{else}\, e'$ evaluates to e (resp. e') if b evaluates to *true* (resp. *false*).

The evaluation of an event α consists in appending α to the current history, and producing the value $*$, that stands for a no-operation.

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 both η and η' satisfy φ. Otherwise, if e is a value and the current history satisfies φ, then the framing is discarded. 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.

A liveness framing $\varphi\langle e\rangle$ is evaluated as follows. The expression e evolves within the framing until the property φ is not satisfied. Of course, if e cannot be further reduced, the execution gets stuck. As soon as the current history obeys φ, the framing is discarded. Note that we cannot operationally guarantee that φ will eventually hold: indeed, this is a liveness property, so

it cannot be enforced by execution monitoring alone. Therefore, we delegate our static analysis for discovering whether a liveness framing will be eventually discarded or not.

The rule for service invocation non-deterministically chooses from Λ a service that respects the types and the required safety and liveness properties, if any. Note that this rule takes advantage of the verification technique in Section 5 to decide about the validity of the history expressions $\varphi[H]$ and $\psi\langle H\rangle$.

3.3. History expressions

To statically predict the histories generated by programs at runtime, as well as the scopes of policies, we introduce *history expressions* with the following abstract syntax. History expressions are much alike regular expressions, 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 $\varphi\langle H\rangle$, and recursion $\mu h.H$ (μ binds the occurrences of the variable h in H). Free variables fv and closed expressions are defined as expected. We assume that the operator $\cdot$ has precedence over $+$.

History Expressions

$$H ::= \varepsilon \mid h \mid \alpha \mid H \cdot H \mid H + H \mid \varphi[H] \mid \varphi\langle H\rangle \mid \mu h.H$$

To define the semantics of history expressions, we enrich histories with a family Σ_Π of special *framing events*, parametrized by policies in Π. The events $[_\varphi$ and $]_\varphi$ denote the opening and closing of a safety framing $\varphi[\cdots]$, while $\langle_\varphi$ and $\rangle_\varphi$ play the same role for liveness framings. Formally, a *history* η is a (possibly infinite) sequence $(\beta_1, \beta_2, \ldots)$ where $\beta_i \in \Sigma \cup \Sigma_\Pi$, $\Sigma_\Pi = \{\, [_\varphi,]_\varphi, \langle_\varphi, \rangle_\varphi \mid \varphi \in \Pi \,\}$, and $\Sigma \cap \Sigma_\Pi = \emptyset$. Note that, if η is infinite, then $\eta\eta' = \eta$, for each η'.

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

We say that a history η is *balanced* when $\eta = \varepsilon$, or η is either $[_\varphi \eta']_\varphi$ or $\langle_\varphi \eta'\rangle_\varphi$ and η' is balanced, or $\eta = \eta'\eta''$, and and both η' and η'' are balanced. For example, $\alpha[_\varphi \alpha'[_{\varphi'} \alpha'']_{\varphi'}]_\varphi$ and $\langle_\varphi \alpha^\omega = \langle_\varphi \alpha^\omega\rangle_\varphi$ are balanced, while $\alpha[_\varphi \alpha'[_{\varphi'} \alpha'']_\varphi$ is not. 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 $\varphi\langle\mathcal{H}\rangle$ as the set $\{\, \langle_\varphi \eta\rangle_\varphi \mid \eta \in \mathcal{H} \,\}$.

The *denotational semantics* of history expressions is defined over the complete lattice $(2^{(\Sigma \cup \Sigma_\Pi)^*}, \subseteq)$. The environment ρ used below maps variables to sets of (finite) histories. We stipulate that concatenation and union of sets of histories are defined only if both their operands are defined. Hereafter, we feel free to omit curly braces, when unambiguous.

Semantics of history expressions

$$[\![\varepsilon]\!]_\rho = \varepsilon \qquad [\![\alpha]\!]_\rho = \alpha \qquad [\![h]\!]_\rho = \rho(h)$$

$$[\![H \cdot H']\!]_\rho = [\![H]\!]_\rho\, [\![H']\!]_\rho \qquad [\![H + H']\!]_\rho = [\![H]\!]_\rho \cup [\![H']\!]_\rho$$

$$[\![\varphi[H]]\!]_\rho = \varphi[[\![H]\!]_\rho] \qquad [\![\varphi\langle H\rangle]\!]_\rho = \varphi\langle [\![H]\!]_\rho\rangle$$

$$[\![\mu h.H]\!]_\rho = \textstyle\bigcup_{n\in\omega} f^n(\emptyset) \quad \text{where } f(X) = [\![H]\!]_{\rho\{X/h\}}$$

For example, consider $H = \mu h.\, \alpha + h \cdot h + \varphi[h]$. The semantics of H consists of all the histories having an arbitrary number of occurrences of α, and arbitrarily nested, balanced safety framings of φ. For instance, $\alpha\varphi[\alpha], \varphi[\alpha]\varphi[\alpha\varphi[\alpha]] \in [\![H]\!]_\emptyset$.

3.4. Validity

We now define when a history is valid. Intuitively, valid histories represent viable computations. Instead, invalid ones happen to violate some security constraint, 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 event can be hidden. For example, a history $\alpha_1\varphi[\alpha_2]\alpha_3$ is valid when $\alpha_1 \models \varphi$ and $\alpha_1\alpha_2 \models \varphi$ (even if α_1 is outside of the safety framing), while $\alpha_1\alpha_2\alpha_3$ is not required to satisfy φ any longer.

To give a formal definition of validity, it is convenient to introduce the notion of safe and live sets (S- and L-sets for short). 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 $\varphi[\cdots]$ encloses 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.

Formally, let $\eta = \beta_1\beta_2\cdots$ be a history. Let $\eta^\flat$ be the plain history obtained from η by erasing all the framing events. Let η^π be the set of all the finite prefixes of η, including the empty history ε. For example, $(\eta_0^\flat)^\pi = (\alpha_c\alpha_r\alpha_c)^\pi = \{\varepsilon, \alpha_c, \alpha_c\alpha_r, \alpha_c\alpha_r\alpha_c\}$.

To have a short, inductive definition of the S-sets $S(\eta)$ and the L-sets $L(\eta)$ of η, it is convenient to manipulate η in order to balance all the safety framings at finite depth, i.e. $[_\varphi\alpha$ becomes $[_\varphi\alpha]_\varphi = \varphi[\alpha]$, and all the framings at the infinity, i.e. $[_\varphi\langle_\psi\alpha^\omega$ becomes $\varphi[\psi\langle\alpha^\omega\rangle]$.

$$\begin{aligned} S(\varepsilon) &= \emptyset \\ S(\eta\,\alpha) &= S(\eta\,\langle_\varphi) = S(\eta\,\rangle_\varphi) = S(\eta) \\ S(\eta_0\,\varphi[\eta_1]) &= S(\eta_0\,\eta_1) \cup \varphi[\eta_0^\flat\,(\eta_1^\flat)^\pi] \end{aligned}$$

$$\begin{aligned} L(\varepsilon) &= \emptyset \\ L(\eta\,\alpha) &= L(\eta\,\langle_\varphi) = L(\eta\,[_\varphi) = L(\eta\,]_\varphi) = L(\eta) \\ L(\eta_0\,\varphi\langle\eta_1\rangle) &= L(\eta_0\,\eta_1) \cup \varphi\langle\eta_0^\flat\,(\eta_1^\flat)^\pi\rangle \end{aligned}$$

where η_0 is finite. We say that a history η is *valid* when:

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

A history expression H is *valid* when all the histories in $[\![H]\!]$ are such. As a first example, we have that:

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

Since neither $\alpha \models \psi$ nor $\alpha\alpha \models \psi$, then η_1 is not valid.

As a more involved example, consider the history:

$$\eta = \langle_\psi[_\varphi\alpha_1]_\varphi\langle_\psi\alpha_2\rangle_\psi\alpha_3[_\varphi\alpha_4[_\varphi\alpha_5$$

Let $\eta|_i$ be the prefix of $\eta^\flat$ containing exactly i events. Then, after rewriting η as $\eta]_\varphi]_\varphi$ to balance the safety framings, we have:

$$\begin{aligned} S(\eta) &= \{\ \varphi[\{\varepsilon,\alpha_1\}], \varphi[\{\eta|_3, \eta|_4, \eta|_5\}], \varphi[\{\eta|_4, \eta|_5\}]\ \} \\ L(\eta) &= \{\ \psi\langle\{\alpha_1, \alpha_1\alpha_2\}\rangle\ \} \end{aligned}$$

Finally, consider, the infinite history $\eta' = \langle_\varphi\alpha^\omega$, where φ requires that eventually α'. The history η' is *not* valid, because it has an L-set $\varphi\langle(\alpha^\omega)^\pi\rangle = \varphi\langle\alpha^*\rangle$ (note that we have balanced the liveness framing at the infinity) but no history in α^* obeys φ.

4. Type and effect system

We now introduce a type and effect system for our calculus, building upon [6, 31]. 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 in $[\![H]\!]$ is generated when a value is applied to an abstraction with that type.

Types and Type Environments

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

A typing judgment $\Gamma, H, I \vdash e : \tau$ means that the expression e evaluates to a value of type τ, and produces a history belonging to the effect H. The component I is a mapping from labels ℓ to sets of indexes I_ℓ, that will be exploited in matching each service invocation $\mathtt{req}_\ell$ with a set of services that respect the required security properties.

Typing relation

$$\frac{}{\Gamma, \varepsilon \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma, \alpha \vdash \alpha : \mathit{unit}} \qquad \frac{}{\Gamma, \varepsilon \vdash * : \mathit{unit}}$$

$$\frac{\Gamma; x:\tau; z:\tau \xrightarrow{H} \tau', H \vdash e : \tau'}{\Gamma, \varepsilon \vdash \lambda_z x{:}\,\tau.e : \tau \xrightarrow{H} \tau'}$$

$$\frac{\Gamma, H \vdash e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash e\,e' : \tau'}$$

$$\frac{\Gamma, H \vdash e : \tau}{\Gamma, \varphi[H] \vdash \varphi[e] : \tau} \qquad \frac{\Gamma, H \vdash e : \tau}{\Gamma, \varphi\langle H\rangle \vdash \varphi\langle e\rangle : \tau}$$

$$\frac{I_\ell = \{\, i \mid e_i : \tau \xrightarrow{H_i} \tau' \in \Lambda \ \wedge\ \varphi[\psi\langle H_i\rangle] \text{ valid} \,\}}{\Gamma, \varepsilon, I_\ell \vdash \mathtt{req}_\ell\,\tau \xrightarrow{\varphi[],\psi\langle\rangle} \tau' : \tau \xrightarrow{\Sigma_{i\in I_\ell} H_i} \tau'}$$

$$\frac{\Gamma, H \vdash e : \tau \quad \Gamma, H \vdash e' : \tau}{\Gamma, H \vdash \mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' : \tau} \qquad \frac{\Gamma, H \vdash e : \tau}{\Gamma, H + H' \vdash e : \tau}$$

The relation $\Gamma, H, I \vdash e : \tau$ is defined as the least relation closed under the rules above; we have omitted the component I when unneeded.

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. The rule for abstraction constraints the premise to equate the effect and the latent effect of functional type. The rule for a service invocation $\mathtt{req}_\ell$ constructs an index set I_ℓ that picks from Λ exactly those services whose history expression

H makes valid both $\varphi[H]$ and $\psi\langle H\rangle$. The last rule allows for *weakening* of effects.

As an example, consider the following expression:

$$e \;=\; \texttt{if}\ b\ \texttt{then}\ \lambda_z x{:}\mathit{unit}.\,\alpha\ \texttt{else}\ \lambda_z x{:}\mathit{unit}.\,\alpha'$$

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

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

Note that we can equate the history expressions $\alpha+\alpha'$ and $\alpha'+\alpha$, because they have the same semantics. The typing derivation above shows the use of the weakening rule to unify the latent effects on arrow types.

Consider now the following expression:

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

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:

$$\dfrac{\dfrac{}{\Gamma,\varepsilon \vdash w : \tau \xrightarrow{H} \tau} \quad \dfrac{\Gamma,\varepsilon \vdash e : \tau \xrightarrow{\alpha+\alpha'} \tau \quad \Gamma,\varepsilon \vdash x : \tau}{\Gamma,\alpha+\alpha' \vdash e\,x : \tau}}{\Gamma,(\alpha+\alpha')\cdot H \vdash w(e\,x) : \tau}$$

The typing derivation proceeds as follows:

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

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:

$$\begin{aligned}\llbracket H \rrbracket &= \llbracket \varepsilon + \varphi[(\alpha+\alpha')\cdot h] \rrbracket_{\{\llbracket H \rrbracket / h\}} \\ &= \{\varepsilon\} \cup \varphi[(\alpha+\alpha')\cdot \llbracket H \rrbracket]\end{aligned}$$

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 [31] suffices for solving constraints as the one above.

The next theorem states that our type and effect system over-approximates the actual runtime histories. As usual, precision is lost when reducing the if-then-else construct to non-determinism, and when dealing with recursive functions (see the examples above). Additionally, we over-approximate the set of services satisfying the calling requirements.

Theorem 1. If $\Gamma, H, I \vdash e : \tau$ and $\varepsilon, e \to^* \eta, e'$, then $\eta \in (\llbracket H \rrbracket^{\flat})^{\pi}$.

To state the type safety property, it is convenient to introduce an alternative operational semantics for our calculus. This new semantics, with transition relation $e \twoheadrightarrow e'$, discards the history component from configurations, and removes all the safety and liveness framings as soon as they are opened. A new reduction rule is provided for service invocation. It exploits the index sets I_ℓ computed by our type and effect system to choose a service that match the security requirements:

$$\dfrac{e \in \{\, e_i \in \Lambda \mid i \in I_\ell \,\}}{\texttt{req}_\ell\,\tau \xrightarrow{\varphi[\,],\psi\langle\rangle} \tau' \twoheadrightarrow e}$$

Unlike $\to$, the transition relation $\twoheadrightarrow$ is subject to no execution monitor (indeed, it does not even track the history η). Also, resolving service invocations involves no validity checks in $\twoheadrightarrow$. The following type safety result states that a well-typed expression with a valid effect can be evaluated with $\twoheadrightarrow$, and will never go wrong.

Theorem 2 (Type Safety). Let $\Gamma, H, I \vdash e : \tau$, with e closed, H valid, and $I_\ell \neq \emptyset$ for each label ℓ in e. Then, each computation of e in $\to$ can be replayed in $\twoheadrightarrow$, and vice versa.

5. Verifying validity

We now extend the procedure in [6] to verify the validity of history expressions. Our technique is based on model checking Basic Process Algebras (BPAs) with Büchi automata. 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 the (context-free) language accepted by the conjunction of the above, which is still a pushdown automaton, is empty. This problem is known to be decidable, and several algorithms and tools show this approach feasible [18].

Recall that 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 $\llbracket H \rrbracket$ is context-free and non-regular, because it contains unbounded pairs of balanced $[_\varphi$ and $]_\varphi$. Since context-free languages are not closed under intersection, the emptiness problem is

undecidable. To apply the procedure sketched above, we then need to manipulate history expressions in order to make validity a regular property.

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 Büchi automata suffices.

5.1. Redundant framings

History expressions can generate histories with *redundant framings*, i.e. nesting of the same framing. For example, the history $\eta = \varphi[\alpha\varphi'[\varphi[\alpha']]]$ 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 $\varphi[\alpha\varphi'[\alpha']]$ is valid. Formally, the L-sets of η comprise $\varphi[\{\alpha, \alpha\alpha'\}]$ for the outer framing, and $\varphi[\{\alpha\alpha'\}]$ for the inner one. Validity requires that all the histories in $\{\alpha, \alpha\alpha'\}$ and $\{\alpha\alpha'\}$ obey φ. Since the second set is strictly included in the first one, it turns out that 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 $n = m$.

5.2. Regularization of safety framings

Below, we define a transformation that, given a history expression H, yields an H' that does not generate redundant safety framings, and H' is valid if and only if H is such.

Let $h^* \in fv(H)$ be a selected occurrence of h in H. We say that h^* is guarded by $guard(h^*, H)$, defined as the smallest set satisfying the following equations.

Guards

$$\begin{array}{rcll} guard(h^*, h) & = & \emptyset & \\ guard(h^*, H_0 \cdot H_1) & = & guard(h^*, H_i) & \text{if } h^* \in H_i \\ guard(h^*, H_0 + H_1) & = & guard(h^*, H_i) & \text{if } h^* \in H_i \\ guard(h^*, \varphi[H]) & = & \{\varphi\} \cup guard(h^*, H) & \\ guard(h^*, \varphi\langle H\rangle) & = & guard(h^*, H) & \\ guard(h^*, \mu h'.\, H') & = & guard(h^*, H') & \text{if } h' \neq h \end{array}$$

For example, in $\mu h.\, \varphi[\alpha \cdot h \cdot \varphi'[h]] \cdot h$, the first occurrence of h is guarded by $\{\varphi\}$, the second one is guarded by $\{\varphi, \varphi'\}$, and the third one is unguarded.

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 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

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

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

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

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

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

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

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

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

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

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 (see [5] for details and an example). We sometimes omit to write 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 $\varphi\langle H\rangle$ is dealt with by removing the liveness framing if $\varphi \in \Phi$ (because there is an outer safety framing enforcing φ), 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 can be easily turned into a terminating rewriting system.

As an example, 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 $guard(h_2, H') = \{\varphi\} \not\subseteq \emptyset$:

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

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

We now establish the following basic properties of regularization.

Theorem 3. For any history expression H:

(a) $H\downarrow$ has no redundant safety framings.

(b) $H\downarrow$ is valid if and only if H is valid

5.3. Basic Process Algebras

Basic Process Algebras [7] provide a natural characterization of (possibly infinite) histories. A *BPA process* is given by the following abstract syntax:

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

where ε denotes the terminated process, $\alpha \in \Sigma$, X is a variable, $\cdot$ denotes sequential composition, $+$ represents (nondeterministic) choice.

A BPA process p is *guarded* if each variable occurrence in p occurs in a subexpression $\alpha \cdot q$ of p. We assume a finite set $\Delta = \{X \stackrel{def}{=} p\}$ of guarded definitions, such that, for each variable X, there exists a single, guarded p such that $\{X \stackrel{def}{=} p\} \in \Delta$. 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. The set $\{(a_i)_i \,|\, p_0 \xrightarrow{a_1} \cdots \xrightarrow{a_i} p_i\} \cup \{(a_i)_i \,|\, p_0 \cdots \xrightarrow{a_i} \cdots\}$ is denoted by $[\![p_0, \Delta]\!]$, where $[\![p, \Delta]\!]^{fin}$ is the first set, containing the strings that label finite computations. We omit the component Δ, when empty.

Operational Semantics of BPA processes

$$\frac{}{\alpha \xrightarrow{\alpha} \varepsilon} \qquad \frac{p \xrightarrow{\alpha} p'}{p + q \xrightarrow{\alpha} p'} \qquad \frac{q \xrightarrow{\alpha} q'}{p + q \xrightarrow{\alpha} q'}$$

$$\frac{p \xrightarrow{\alpha} p'}{p \cdot q \xrightarrow{\alpha} p' \cdot q} \qquad \frac{p \xrightarrow{\alpha} p' \quad X \stackrel{def}{=} p \in \Delta}{X \xrightarrow{\alpha} p'}$$

We now introduce a mapping from history expressions to BPAs, in the line of [6, 31]. Without loss of generality, we assume that all the variables in H have distinct names. 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.

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.

The rules that transform history expressions into BPAs are rather standard. History events, variables, concatenation and choice are mapped into the corresponding BPA counterparts. 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. An expression $\varphi[H]$ is mapped to the BPA for H, surrounded by the opening and closing of the φ-framing.

Mapping history expressions to BPAs

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

We now state the correspondence between history expressions and BPAs. The prefixes of the histories generated by a history expression H (i.e. $[\![H]\!]^\pi$) are all and only the finite prefixes of the strings that label the computations of $BPA(H)$.

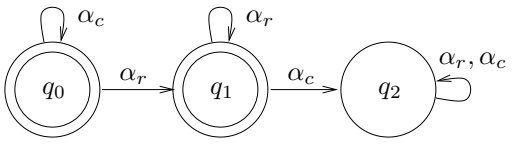
Lemma 4. $[\![H]\!]^\pi = [\![BPA(H)]\!]^{fin}$.

5.4. Büchi Automata

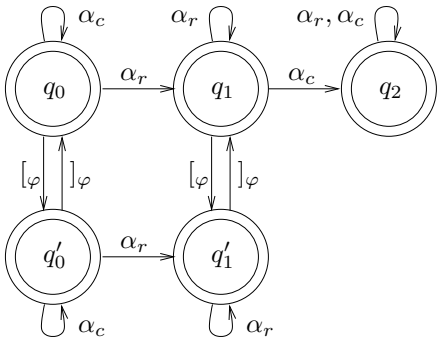
Büchi automata are finite state automata whose acceptance condition roughly says that a computation is accepted if some final state is visited infinitely often; see [35] for details. Since we also need to establish the validity of finite histories, we use the standard trick of padding a finite string with a special symbol \$. Then, each final state has a self-loop labelled with \$. For brevity, we will omit these transitions hereafter.

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. In the first case, we require that η has no redundant safety framings. Hereafter, let the formula φ be defined by the Büchi automaton $A_\varphi = (\Sigma, Q, Q_0, \rho, F)$, which we assume to have a non-final sink state.

As an example, let φ be the policy saying that no event α_c can occur after an α_r (see Section 2). The Büchi automaton for φ is shown below.



We define the formula $\varphi_{[]}$ through the Büchi automaton $A_{\varphi_{[]}}$ depicted below. For example, the history $[_\varphi\alpha_r]_\varphi\alpha_c$ is accepted by $A_{\varphi_{[]}}$, while $\alpha_r[_\varphi\alpha_c]_\varphi$ is not (recall that we do not draw the self-loops labelled by \$).



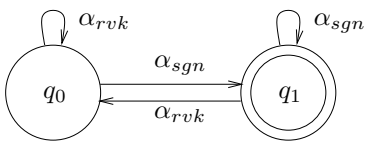
Intuitively, the automaton $A_{\varphi_{[]}}$ is partitioned into two layers. The first layer is a copy of A_φ, where all the states are final. This models the fact that we are outside the scope of φ, i.e. the history leading to any state in this layer has balanced safety framings of φ (or none). The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

Büchi automaton for $\varphi_{[]}$

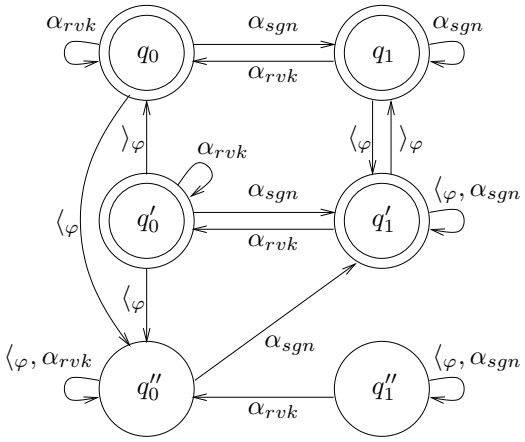
$$
\begin{array}{l}
A_{\varphi_{[]}} = (\Sigma', Q', Q_0, \rho', F')\\
\quad \Sigma' = \Sigma \cup \{\, [_\varphi,]_\varphi, \langle_\varphi, \rangle_\varphi \mid \varphi \in \Pi \,\}\\
\quad Q' = F' = Q \cup \{\, q' \mid q \in F \,\}\\
\quad \rho' = \rho \cup \{\, (q, [_\varphi, q') \mid q \in F \,\} \;\cup\; \{(q',]_\varphi, q)\}\\
\qquad\quad \cup \{\, (q_0', \alpha, q_1') \mid (q_0, \alpha, q_1) \in \rho \wedge q_1 \in F \,\}\\
\qquad\quad \cup \{\, (q, [_{\varphi'}, q) \cup (q,]_{\varphi'}, q) \mid q \in Q' \wedge \varphi' \neq \varphi \,\}\\
\qquad\quad \cup \{\, (q, \langle_{\varphi''}, q) \cup (q, \rangle_{\varphi''}, q) \mid q \in Q' \,\}
\end{array}
$$

Likewise, we define the Büchi automaton $A_{\varphi_{\langle\rangle}}$, but we now allow for redundant liveness framings.

As an example, let φ be the policy saying that α_{sgn} eventually occurs with no subsequent α_{rvk} (see Section 2). The Büchi automaton for φ is shown below.



The Büchi automaton for $\varphi_{\langle\rangle}$ is illustrated below. For example, the history $\alpha_{rvk}\langle_\varphi\alpha_{sgn}$ is accepted by $A_{\varphi_{\langle\rangle}}$, while $\langle_\varphi\alpha_{sgn}\alpha_{rvk}\langle_\varphi$ is not.



The automaton $A_{\varphi_{\langle\rangle}}$ consists of three layers. The first layer is a copy of A_φ, and it represents being outside of the liveness framing $\varphi\langle\cdots\rangle$. 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 φ, while if you are in the third layer, you are still looking for. Suppose now that a new framing $\varphi\langle\cdots\rangle$ 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_φ, 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 $A_{\varphi_{\langle\rangle}}$ is defined in the following table.

Büchi automaton for $\varphi_{\langle\rangle}$

$$
\begin{aligned}
A_{\varphi_{\langle\rangle}} &= (\Sigma', Q', Q_0, \rho', F') \\
\Sigma' &= \Sigma \cup \{\, [_\varphi,]_\varphi, \langle_\varphi, \rangle_\varphi \mid \varphi \in \Pi \,\} \\
Q' &= Q \cup \{\, q', q'' \mid q \in Q \,\} \\
F' &= \{\, q' \mid q \in Q \,\} \\
\rho' &= \rho \cup \{\, (q, \langle_\varphi, q') \mid q \in F \,\} \cup \{\, (q, \langle_\varphi, q'') \mid q \notin F \,\} \\
&\quad \cup \{\, (q'_0, \alpha, q'_1) \mid (q_0, \alpha, q_1) \in \rho \,\} \\
&\quad \cup \{\, (q', \langle_\varphi, q') \mid q \in F \,\} \cup \{\, (q', \langle_\varphi, q'') \mid q \notin F \,\} \\
&\quad \cup \{\, (q''_0, \alpha, q'_1) \mid (q_0, \alpha, q_1) \in \rho \wedge q_1 \in F \,\} \\
&\quad \cup \{\, (q''_0, \alpha, q''_1) \mid (q_0, \alpha, q_1) \in \rho \wedge q_1 \notin F \,\} \\
&\quad \cup \{(q', \rangle_\varphi, q)\} \cup \{(q'', \langle_\varphi, q'')\} \\
&\quad \cup \{\, (q, \langle_{\varphi'}, q) \cup (q, \rangle_{\varphi'}, q) \mid q \in Q' \wedge \varphi' \neq \varphi \,\} \\
&\quad \cup \{\, (q, [_{\varphi''}, q) \cup (q,]_{\varphi''}, q) \mid q \in Q' \,\}
\end{aligned}
$$

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

Lemma 5. 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 $[_\varphi, \langle_\psi \in \eta$.

Büchi automata are closed under intersection [35]. Therefore, 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.

The last result of our paper follows. Validity of a history expression H can be decided by showing that the BPA generated by the regularization of H satisfies a ω-regular formula. Together with Theorem 2, an expression in our calculus never violates security if its effect is checked valid. Thus we are dispensed from using an execution monitor.

Theorem 6. A history expression H is valid iff:

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

6. Conclusions and related work

We have enriched the λ-calculus with primitives to express service composition under security constraints. The security requirements are safety and liveness properties over execution histories. These properties have a local scope, possibly nested. The policies enforced by the framing can always inspect the whole past history. If of interest, 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 discards all the events before β_φ, and then behaves like the standard automaton enforcing φ.

We have used a type and effect system to extract from a given program a history expression, i.e. a safe approximation of its runtime behaviour. A history expression is valid when it represents execution histories that never violate the security policies within their scope. To verify the validity of history expression, we have exploited model checking over Basic Process Algebras and Büchi automata. However, nesting of scopes makes validity non-regular, and has required us to transform history expressions so that model checking is feasible. When a history expression of a program e is verified valid, then e will not go wrong, so we no runtime monitoring is required. Note that our static analysis enables us to compose at compile-time those services that match the imposed security constraints.

Besides our own work [6], that of Skalka and Smith [31] 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 [31], 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 [31] 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 the same techniques of [31].

The secure composition of components has been the main concern underlying the design of Sewell and Vitek's box-π [30], 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 [30] there is no analog of our liveness framings.

Gorla, Hennessy and Sassone [22] 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.

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 efficient static verification of properties of composed systems. Session types [23] have been exploited to formalize compatibility of components [34] and to describe adaptation of web services [14]. Security issues have been recently considered in terms of session types, e.g. in [11], which proves the decidability of type-checking in an extension of the π-calculus with session types and correspondence assertions [38].

A related line of research addresses the issue of modelling and analysing resource usage. Igarashi and Kobayashi [24] 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 [24], but we provide a static verification technique, while [24] does not. Colcombet and Fradet [16] and Marriot, Stuckey and Sulzmann [25] 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 [8] tackle the problem of characterizing when a program can call a stack-inspecting method while respecting a global security policy. Compared to [16, 25, 8], our programming model allows for local policies, while the other only considers global ones.

Other works have proposed type-based methodologies to check security properties of distributed systems. For instance, Gordon and Jeffrey [21] use a type and effect system to prove authenticity properties of security protocols. Web service authentication has been recently modelled and analysed in [9, 10] through a process calculus enriched with cryptographic primitives.

Acknowledgments

We wish to thank the anonymous referees for their insightful comments and suggestions. This work has been partially supported by EU project DEGAS (IST-2001-32072) and FET project PROFUNDIS (IST-2001-33100).

References

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

[2] S. Anderson et al. *Web Services Trust Language (WS-Trust)*, 2005.

[3] T. Andrews et al. *Business Process Execution Language for Web Services (BPEL4WS), Version 1.1*, 2003.

[4] B. Atkinson et al. *Web Services Security (WS-Security)*, 2002.

[5] M. Bartoletti. *Language-based Security: Access Control and Static Analysis*. PhD thesis, Università di Pisa, 2005.

[6] M. Bartoletti, P. Degano, and G. L. Ferrari. History based access control with local policies. In *Proc. Fossacs*, 2005.

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

[8] F. Besson, T. de Grenier de Latour, and T. Jensen. Interfaces for stack inspection. *Journal of Functional Programming*, 2005. To appear.

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

[10] K. Bhargavan, C. Fournet, and A. D. Gordon. A semantics for web services authentication. In *Proc. ACM Symposium on Principles of Programming Languages*, 2004.

[11] E. Bonelli, A. Compagnoni, and E. Gunter. Typechecking safe process synchronization. In *Proc. Foundations of Global Ubiquitous Computing*, 2004.

[12] D. Box et al. *Simple Object Access Protocol (SOAP) 1.1.* WRC Note, 2000.

[13] D. Box et al. *Web Services Policy Framework (WS-Policy)*, 2002.

[14] A. Brogi, C. Canal, and E. Pimentel. Behavioural types and component adaptation. In *Proc. AMAST*, 2004.

[15] R. Chinnici, M. Gudgina, J. Moreau, and S. Weerawarana. *Web Service Description Language (WSDL), Version 1.2*, 2002.

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

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

[18] 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*, 1994.

[19] F. C. Gärtner. Revisiting liveness properties in the context of secure systems. In *Proc. FASec*, 2002.

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

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

[22] D. Gorla, M. Hennessy, and V. Sassone. Security policies as membranes in systems for global computing. In *Foundations of Global Ubiqitous Computing Workshop*, 2004.

[23] K. Honda, V. Vansconcelos, and M. Kubo. Language primitives and type discipline for structures communication-based programming. In *Proc. ESOP*, 1998.

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

[25] K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proc. First Asian Programming Languages Symposium*, 2003.

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

[27] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE*, 2003.

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

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

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

[31] C. Skalka and S. Smith. History effects and verification. In *Asian Programming Languages Symposium*, 2004.

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

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

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

[35] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. Banff Higher order workshop conference on Logics for concurrency*, 1996.

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

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

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