Secure Service Composition with Symbolic Effects

Gabriele Costa, Pierpaolo Degano, Fabio Martinelli

Secure Service Composition with Symbolic Effects

Gabriele Costa
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche (CNR)
and
Dipartimento di Informatica, Università di Pisa
Pisa, Italy
gabriele.costa@iit.cnr.it

Pierpaolo Degano
Dipartimento di Informatica
Università di Pisa
Pisa, Italy
degano@di.unipi.it

Fabio Martinelli
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche (CNR)
Pisa, Italy
fabio.martinelli@iit.cnr.it

***Abstract*—Local policies represent security properties that are applied to (parts of) programs or services. They are amenable for developers since they provide for a full compositionality (through scope nesting), for a simple, automaton-like structure and for a direct enforcing through a corresponding execution monitor. Compliance w.r.t. local policies is statically verified against a safe over-approximation of all the possible execution traces, namely a history expression. Given a service, a safe type and effect system extracts a history expression, from which a viable composition plan can be automatically produced. Viable plans drive executions that never rise policy exceptions. Our main contribution consists in defining a type and effect system that also deals with open systems. We extend the syntax of a service-oriented version of the λ-calculus, namely λ^{req}, with resources and external branching operators. Then, we safely over-approximate the possible run-time behaviour of services collecting partial information on the relationship between the program flow and the actual resources. Indeed, the history expressions obtained in this way are compact, rather accurate and able to derive viable plans in most cases.**

I. Introduction

The main purpose of the history-based approach to security is to discriminate between legal and illegal execution traces [1]. We can classify most history-based techniques in two different groups: static verification and run-time enforcement. In the first case, the compliance with the security policy is checked on a model describing (a suitable approximation of) all the possible behaviours of a system. The other approach mainly consists in deploying a monitor that is responsible for preventing the actual execution trace from violating the security policy. Both of them have been widely studied (e.g. see [16], [7], [15], [12]) and many results on their features are present in the literature. An approach based on *local policies* [3] candidates to be a mixed one to the problem of security. These policies are defined through *usage automata*, namely a variant of non-deterministic finite-state automata (NFAs) parameterized over system's resources. Usage automata are intuitive, can be defined via visual editing and do not require specific technical skills. Policies compose each other through scope nesting and apply also to higher-order terms (e.g. mobile applications an service composition). The compliance with policies is statically verified through model checking. If a system passes this static check, then it can run with no security enforcement. Otherwise, the necessary policies give automatically rise to a reference monitor associated with the guarded code (e.g. via instrumentation). In [2] Java has been extended with local policies showing the feasibility of this approach on a real-world system.

This research has been partially supported by EU-FETPI Global Computing Project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers), EU project FP7-214859 CONSEQUENCE (Context-aware data-centric information sharing) and EU project FP7-231167 CONNECT (Emergent Connectors for Eternal Software Intensive Networked Systems).

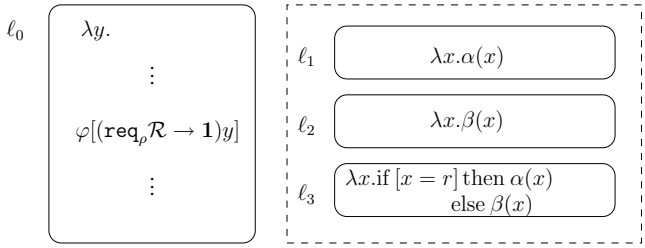

Figure 1. An open service network. Request ρ can be satisfied by three services (ℓ_1, ℓ_2 and ℓ_3)

The benefits of verifying history expressions extend to the orchestration of services in a closed world [4], [5]. The orchestrator produces a plan, with an associated all-inclusive history expression. If this approximation of the whole service network is statically proved to comply with the required policies, then the composed service is secure.

In many cases we would like to also analyse *open* networks, i.e. networks having unspecified participants. We give an intuition through an example.

Example 1. Consider the four-services network in Figure I. A service is represented by a rounded box, and is hosted at location ℓ_i. During its execution, the service at ℓ_0 calls another service, with parameter y. The call is subject to a

policy φ. Suppose that the three services ℓ_1, ℓ_2 and ℓ_3 are compatible with the request ρ. Imagine now that the policy φ says: *"callee must never do action β on the (globally known) resource r and can do α only on r"*. Completing the network with clients binding y to some actual resource we can check which services can be safely associated with ρ for each possible composition plan. In this example it is easy to verify that ℓ_3 always complies with φ. Exploiting this observation we can define a viable plan without any knowledge about the structure of clients (that could be statically unpredictable). Note that this plan is *partial* because it only concerns the four services considered, regardless of the environment in which they will operate.

Here, we extend the results of [4], [5] on networks that are open in the sense mentioned above. In particular, we will single out *partial* plans that involve parts of the known network and that can be safely adopted within any operating context. We say that a plan is partial if it is partially defined over the set of existing requests. For simplicity, here we restrict our analysis to *stateless* security policies, i.e. they do not depend on the previous execution history, but only speak about the intended behaviour of called services.

The paper is structured as follows. In Section II we give the syntax and semantics of λ^{req}. In Section III we introduce history expressions and our type and effect system. Finally, Section IV concludes the paper.

II. Service structure

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

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

A. Syntax

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

Function abstraction and application follow the classical syntax (we use z in $\lambda_z x.e$ as a binding to the function in e). Security framing applies the scope of a policy φ to a program e. Service request requires more attention. We state that services can not be directly accessed (for instance using a public name or address). Clients invoke services through their public interface, i.e. their type. The label ρ is a unique identifier associated with the request. Since both τ and τ' can be higher-order types, we can model simple value-passing interaction, as well as mobile code scenarios.

Table I
Syntax of λ^{req}

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

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

Here we make explicit the conditions of branching, similarly to [8]. Briefly, we check equality between resources, and we have negation and conjunction.

Definition 1. (Syntax of guards)

$$g, g' ::= true \mid [x^* = y^*] \mid \neg g \mid g \wedge g'$$

where x^*,y^* range over variable and resource names.

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

We now introduce our two working examples.

Example 2. Let e be a service that inputs an address taken from the set of network addresses $\mathtt{Addr} = \{a_1, \ldots, a_n\}$. Then e verifies that the current address is a trusted one, i.e. checks if it belongs to a set $\mathtt{TAddr} \subseteq \mathtt{Addr}$. If so, e sends some private message to the trusted address, otherwise e terminates. We model the service e in this way:

$$e = \lambda x.\mathtt{if}\ [x \in \mathtt{TAddr}]\ \mathtt{then}\ \mathtt{send}(x)\ \mathtt{else}\ *$$

where $[x \in \mathtt{TAddr}] = \bigvee_{a \in \mathtt{TAddr}} [x = a]$

Example 3. Consider now a slight variant of the previous service. Now e only sends messages to clients having a sponsor inside the service network. A sponsor is a service

e' that, given an address $a \in \texttt{Addr}$, answers with a value in $\mathcal{S} = \{A, R\}$ where A (R resp.) means that e' accepts (refuses) the sponsorship for a. Then a possible implementation for e is:

$$e = \lambda x.(\lambda y.\texttt{if } [y = A] \texttt{ then } \texttt{send}(x) \texttt{ else } *)((\texttt{req}_\rho \texttt{ Addr} \to \mathcal{S})x)$$

Note that e needs to know nothing about sponsors and their implementation.

B. Operational semantics

Clearly, the run-time behaviour of a network of services depends on the way they interact. As we already mentioned, requests do not directly refer to a specific service that will be actually invoked during the execution, but to its type, only. A *plan* resolves the requests by giving a finite, total mapping from requests to services locations. Needless to say, different plans lead to different executions. A plan is said to be *valid* if and only if the execution it drives complies with all the active security policies. A service network can have many, one or even no viable plans.

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

A computational step of a program is a transition from a source configuration to a target one. In our model, configurations are pairs η, e where η is the execution *history*, and e is the expression under evaluation.

The operational semantics of λ^{req} is given in Table II. Given a plan π we can evaluate λ^{req} expressions, i.e. services, accordingly with these rules.

Briefly, an action $\alpha(r)$ is appended to the current history (possibly after the evaluation of its parameter), a conditional branching chooses between two possible executions (depending on its guard), a security framing checks the current history before allowing the next step, an application works as usual and a service request retrieves the service that the current plan π associates with ρ.

Example 4. Consider the service introduced in Example 2. If it is invoked with parameter $a \in \texttt{TAddr}$ starting from an empty trace, its execution is:

$$\begin{aligned}\varepsilon, (e\ a) &\to \varepsilon, \texttt{if } [a \in \texttt{TAddr}] \texttt{ then } \texttt{send}(a) \texttt{ else } * \\ &\to \varepsilon, \texttt{send}(a) \to \texttt{send}(a), *\end{aligned}$$

III. Type and effect system

We now introduce the type and effect system for λ^{req}. Our system inherits its structure from [4] introducing two main improvements: guarded effects and a new typing rule, namely *strengthening*.

Table II
Operational semantics of λ^{req}

$$(\texttt{S}-\texttt{Ev}_1)\quad \frac{\eta, e \to_\pi \eta', e'}{\eta, \alpha(e) \to_\pi \eta', \alpha(e')}$$

$$(\texttt{S}-\texttt{Ev}_2)\quad \eta, \alpha(r) \to_\pi \eta\alpha(r), *$$

$$(\texttt{S}-\texttt{If})\quad \eta, \texttt{if } g \texttt{ then } e_{tt} \texttt{ else } e_{ff} \to_\pi \eta, e_{\mathcal{B}(g)}$$

$$(\texttt{S}-\texttt{Frm}_1)\quad \frac{\eta, e \to_\pi \eta', e' \quad \eta \models \varphi}{\eta, \varphi[e] \to_\pi \eta', \varphi[e']}$$

$$(\texttt{S}-\texttt{Frm}_2)\quad \frac{\eta \models \varphi}{\eta, \varphi[v] \to_\pi \eta, v}$$

$$(\texttt{S}-\texttt{App}_1)\quad \frac{\eta, e_1 \to_\pi \eta', e_1'}{\eta, e_1 e_2 \to_\pi \eta', e_1' e_2}$$

$$(\texttt{S}-\texttt{App}_2)\quad \frac{\eta, e_2 \to_\pi \eta', e_2'}{\eta, v e_2 \to_\pi \eta', v e_2'}$$

$$(\texttt{S}-\texttt{App}_3)\quad \eta, (\lambda_z x.e)v \to_\pi \eta, e\{v/x, \lambda_z x.e/z\}$$

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

A. History expressions

History expressions will be used to denote set of histories. They are defined through the following abstract syntax, which extend those of [4].

Definition 2. (Syntax of history expressions)

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

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

Table III
SEMANTICS OF HISTORY EXPRESSIONS

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

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

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

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

given in Def. 1).

The *denotational semantics* of history expressions (given in Table III) maps a history expression H to a set of histories $\mathcal{H}$. The domain $\mathcal{H}$ is the lifted complete partial order of sets of histories [18]. Sets are ordered by (lifted) inclusion $\subseteq_\perp$ (where $\forall \mathcal{H}. \perp \subseteq_\perp \mathcal{H}$ and $\mathcal{H} \subseteq_\perp \mathcal{H}'$ whenever $\eta \in \mathcal{H} \Rightarrow \eta \in \mathcal{H}'$).

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

B. Typing relation

Before introducing a type and effect system for our calculus, we need some auxiliary definitions.

The relation $\sqsubseteq_\sigma$ is a partial order between history expressions defined as:

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

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

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

Definition 3. (Types and type environments)

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

Table IV
TYPING RELATION

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

$$(\texttt{T-Var})\ \Gamma, \varepsilon \vdash_g x : \Gamma(x)$$

$$(\texttt{T-Ev})\ \frac{\Gamma, H \vdash_g e : \mathcal{R}}{\Gamma, H \cdot \sum_{r \in \mathcal{R}} \alpha(r) \vdash_g \alpha(e) : \mathbf{1}}$$

$$(\texttt{T-Abs})\ \frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash_g e : \tau'}{\Gamma, \varepsilon \vdash_g \lambda_z x.e : \tau \xrightarrow{H} \tau'}$$

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

$$(\texttt{T-Frm})\ \frac{\Gamma, H \vdash_g e : \tau}{\Gamma, \varphi[H] \vdash_g \varphi[e] : \tau}$$

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

$$(\texttt{T-Req})\ \frac{I = \{H \mid e_{\ell'} : \tau \xrightarrow{H} \tau' \in \texttt{Srv}\rfloor_\ell\}}{\Gamma, \varepsilon \vdash_g \texttt{req}_\rho\, \tau \to \tau' : \tau \xrightarrow{\sum_{i \in I} H_i} \tau'}$$

$$(\texttt{T-Wkn})\ \frac{\Gamma, H \vdash_g e : \tau \quad H \sqsubseteq H'}{\Gamma, H' \vdash_g e : \tau}$$

$$(\texttt{T-Str})\ \frac{\Gamma, H \vdash_g e : \tau \quad g \Rightarrow g'}{\Gamma, g'H \vdash_g e : \tau}$$

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

Rules for $*$, resources and variables are straightforward. An event has type $\mathbf{1}$ and produces a history that is the one obtained from the evaluation of its parameter increased with the event itself. Note that, since the class of resources is finite, we can only have a finite number of instantiations of an event[1]. An abstraction has an empty effect and a functional type carrying a latent effect, i.e. the effect that will be produced when the function is actually applied. The application moves the latent effect (labelling the function type) to the actual history expression and concatenates it with the actual effects according the call-by-value semantics. Security framing extends the scope of the property φ to

[1]If we have more different types of resources, i.e. $\mathcal{R} = \mathcal{R}_1 \cup \cdots \cup \mathcal{R}_k$, we can restrict the number of choices to the resources that are compatible with the event (e.g. $\mathcal{R} = \texttt{Socket} \cup \texttt{File}$ and the action `accept_connection`).

the effect of its target. The rule for conditional branching is new. It says that if we can type e and e' to the same τ generating the same effect H, then we can extend τ and H to be the type and effect of the whole expression. Moreover, in typing the sub-expressions we take into account g' and its negation, respectively. Similarly to abstractions, service requests have an empty effect. However, the type of a request is obtained as the composition of all the types of the possible servers. In particular, the resulting latent effect is the (unguarded) non-deterministic choice among them. Observe that we only accept exact matching for input/output types (see [4] for a different composition of types). The last two rules are for weakening and strengthening. The first states that is always possible to make a generalisation of the effect inferred from an expression e. Finally, $(\mathtt{T{-}Str})$ applies a guard g' to an effect H provided that $g \Rightarrow g'$ (if and only if $\forall\ \sigma.\sigma \models g \Rightarrow \sigma \models g'$). This rule says that we can use the information stored in g for wrapping an effect in a guarded context.

Example 5. Let e be the service introduced in Example 2, then the following derivation is possible (dots stands for trivial or symmetrical derivations)

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x : \mathtt{Addr}, \mathtt{send}(\mathtt{Addr}) \vdash_{[x \in \mathtt{TAddr}]} \mathtt{send}(x) : \mathbf{1}}}{x : \mathtt{Addr}, [x \in \mathtt{TAddr}]\, \mathtt{send}(\mathtt{Addr}) \vdash_{[x \in \mathtt{TAddr}]} \mathtt{send}(x) : \mathbf{1}} \quad \vdots}{x : \mathtt{Addr}, [x \in \mathtt{TAddr}]\, \mathtt{send}(\mathtt{Addr}) \vdash_{true} \mathtt{IF} : \mathbf{1}}}{\emptyset, \varepsilon \vdash_{true} e : \mathtt{Addr} \xrightarrow{[x \in \mathtt{TAddr}]\mathtt{send}(\mathtt{Addr})} \mathbf{1}}$$

where $\mathtt{send}(\mathtt{Addr}) = \sum_{i=1}^{n} \mathtt{send}(a_i)$ and $\mathtt{IF} = \mathtt{if}\ [x \in \mathtt{TAddr}]\ \mathtt{then}\, \mathtt{send}(x)\, \mathtt{else}\, *$.

Example 6. Consider now e to be the service of Example 3. For the rightmost expression we can derive

$$\dfrac{\dfrac{-}{x : \mathtt{Addr}, \varepsilon \vdash_{true} \mathtt{req}_\rho\ \mathtt{Addr} \to \mathcal{S} : \tau} \qquad \dfrac{-}{x : \mathtt{Addr}, \varepsilon \vdash_{true} x : \mathtt{Addr}}}{x : \mathtt{Addr}, \textstyle\sum_{i \in I} H_i \vdash_{true} (\mathtt{req}_\rho\ \mathtt{Addr} \to \mathcal{S}) x : \mathcal{S}}$$

where $\tau = \mathtt{Addr} \xrightarrow{\sum_{i \in I} H_i} \mathcal{S}$. Moreover we can derive

$$\dfrac{\dfrac{\dfrac{\vdots}{x : \mathtt{Addr}; y : \mathcal{S}, [y = A]\, \mathtt{send}(\mathtt{Addr}) \vdash_{[y=A]} \mathtt{send}(x) : \mathbf{1}} \quad \vdots}{x : \mathtt{Addr}; y : \mathcal{S}, [y = A]\, \mathtt{send}(\mathtt{Addr}) \vdash_{true} \mathtt{IF} : \mathbf{1}}}{x : \mathtt{Addr}, \varepsilon \vdash_{true} \lambda_y.\mathtt{IF} : \mathcal{S} \xrightarrow{[y=A]\mathtt{send}(\mathtt{Addr})} \mathbf{1}}$$

with $\mathtt{IF} = \mathtt{if}\ [\mathtt{y} = \mathtt{A}]\ \mathtt{then}\, \mathtt{send}(\mathtt{x})\, \mathtt{else}\, *$. Combining the two we obtain

$$\emptyset, \varepsilon \vdash_{true} e : \mathtt{Addr} \xrightarrow{\bar{H}} \mathbf{1}$$

where $\bar{H} = \sum_{i \in I} H_i \cdot [y = A]\, \mathtt{send}(\mathtt{Addr})$.

Example 7. Consider now the term

$$e = \lambda x.\mathtt{if}\ [x = r]\ \mathtt{then}\, \lambda y.\alpha(y)\, \mathtt{else}\, \lambda y.\beta(y)$$

and let $\mathcal{R} = \{r, s\}$ be the set of resources compatible with actions α and β. Then the following derivation is possible

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : \tau; y : \mathcal{R}, \alpha(\mathcal{R}) \vdash_{[x=r]} \alpha(y) : \mathbf{1}}{x : \tau; y : \mathcal{R}, [x = r]\, \alpha(\mathcal{R}) \vdash_{[x=r]} \alpha(y) : \mathbf{1}}}{x : \tau; y : \mathcal{R}, [x = r]\, \alpha(\mathcal{R}) + [x \neq r]\, \beta(\mathcal{R}) \vdash_{[x=r]} \alpha(y) : \mathbf{1}} \quad \vdots}{x : \tau, \varepsilon \vdash_{[x=r]} \lambda y.\alpha(y) : \mathcal{R} \xrightarrow{[x=r]\alpha(\mathcal{R}) + [x \neq r]\beta(\mathcal{R})} \mathbf{1}}}{x : \tau, \varepsilon \vdash_{true} e' : \mathcal{R} \xrightarrow{[x=r]\alpha(\mathcal{R}) + [x \neq r]\beta(\mathcal{R})} \mathbf{1}}}{\emptyset, \varepsilon \vdash_{true} e : \tau \xrightarrow{\varepsilon} (\mathcal{R} \xrightarrow{[x=r]\alpha(\mathcal{R}) + [x \neq r]\beta(\mathcal{R})} \mathbf{1})}$$

where $e' = \mathtt{if}\ [x = r]\ \mathtt{then}\, \lambda y.\alpha(y)\, \mathtt{else}\, \lambda y.\beta(y)$.

Our type and effect system produces history expressions that approximate the run-time behaviour of programs. The validity of our approach resides in producing history expressions *safely*. We say a type and effect system to be safe if each trace produced by the execution of a program is always denoted by the history expression obtained typing it. To prove type and effect safety we need the following results. All the proofs of our results can be found in [17].

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

In words, Lemma 1 states that type and effect inference is closed under logical implication for guards.

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

Lemma 2 says that history expressions are preserved by programs execution.

Then, the type safety is guaranteed by the following theorem.

Theorem 1. ***(Type safety)*** *If* $\Gamma, H \vdash_{true} e : \tau$ *and* $\varepsilon, e \to^*_\pi \eta, v$ *then* $\forall \sigma.\eta \in [\![H]\!]^\sigma$.

Unlike the type system of [3], the presence of the rule $(\mathtt{T{-}Str})$ makes it possible to discard some of the denoted traces. These traces correspond to executions that, due to the actual instantiation of formal parameters, can not take place. Consequently, our type and effect system produces more compact history expressions than those of [3]. This has several advantages, and we mention here a couple of them. First, having small history expression reduces the search space contributing to speed up verification algorithms. Secondly, it removes possible false negatives raising from traces that violate the active policy but will never be produced at runtime.

IV. Conclusion and related work

We extended the type system of λ^{req}, a service-oriented version of the call-by-value λ-calculus, with rules for inferring guarded history expressions. Our approach produces a finer, but still safe, static approximation of the possible run-time behaviours than those of [3]. Although our type system performs similarly to others in case of closed service networks, we showed that it can be successfully applied to partially undefined networks of services. This assumption seems to be realistic. Indeed an "a priori" knowledge of clients is not always available to services at deploy-time. On the contrary, a service could be interested in verifying which compositions are viable only considering the part of the service network it is aware of at a certain time.

Many improvements are possible on the present proposal. For instance, exploiting the correspondence between history expressions and basic process algebras (BPAs [6]), we could apply the approach of [8] for defining a symbolic transition system on which some behavioural analysis are available. Additionally, it would be important studying the minimal necessary conditions under which partial plans are fully compositional. Indeed, such a result would allow the orchestration strategy to be distributed, so reducing the complexity of this problem.

Verifying security properties of orchestrations is a major issue in service oriented paradigms and many authors have studied it. However, to the best of our knowledge, little work has been done on partial orchestration.

In [9] the authors present a framework for contract-based creation of choreographies. Roughly, they use a contract system for finding a match between contracts and choreographies. In this way they are able to find whether a given contract, declared by a service joining the network, is consistent with the current choreography. However, this framework exploits a global knowledge about the network structure while our model aims to infer partial compositions, whenever they exist. Actually, we want to verify whether a service can be composed with (a visible part of) the network respecting the active security policies.

For defining contracts, Castagna et al. [11] use a variant of CCS with internal and external choice operators [14]. A subcontract relation guarantees that the choreography always respects the contracts of both client and service. Nevertheless, this approach assumes that clients always know the contract for a request and it is focused on finding a satisfactory composition with some available service. Since the composition depends on the possible instantiations of a service, such a contract could be unavailable during the analysis phase. In this sense, our technique overcomes this limitation by producing symbolic partial plans that will be instantiated over actual resources.

Busi et al. [10] propose an analysis of service orchestration and choreography. In their work, the validity of the orchestration of services is a consequence of its conformance respect to the intended choreography. The main difference with our work consists in the approach to choreography. Indeed, they start from a pre-defined choreography and verify the validity of an orchestration. Instead, our approach aims to check partial service composition without relying on any global orchestrator.

In [13] a framework for the synthesis of orchestrators is presented. This technique consists in automatically producing an orchestrator guaranteeing a service composition to respect the desired security policy. This approach defines a composition that complies with the client's policy. Since our method produces partial compositions that respect all the involved policies (on both clients and servers), it seems to be more general. However, as [13] generates dynamic orchestrators, the two systems work under completely different assumptions.

References

[1] Martín Abadi and Cédric Fournet. Access control based on execution history. In *NDSS*, 2003.

[2] Massimo Bartoletti, Gabriele Costa, Pierpaolo Degano, Fabio Martinelli, and Roberto Zunino. Securing Java with local policies. In *Workshop on Formal Techniques for Java-like Programs*, 2008.

[3] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. History-based access control with local policies. In *FoSSaCS*, pages 316–332, 2005.

[4] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Planning and verifying service composition. *Journal of Computer Security (JCS)*, 17(5):799–837, 2009. (Abridged version In *Proc. 18th Computer Security Foundations Workshop (CSFW)*, 2005).

[5] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Secure service orchestration. In *FOSAD*, pages 24–74, 2007.

[6] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[7] Frédéric Besson, Thomas P. Jensen, and Daniel Le Métayer. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217–250, 2001.

[8] Michele Boreale and Rocco De Nicola. A symbolic semantics for the pi-calculus. *Inf. Comput.*, 126(1):34–52, 1996.

[9] Mario Bravetti, Ivan Lanese, and Gianluigi Zavattaro. Contract-driven implementation of choreographies. In *TGC*, pages 1–18, 2008.

[10] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration: A synergic approach for system design. In *ICSOC*, pages 228–240, 2005.

[11] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5), 2009.

[12] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.

[13] Fabio Martinelli and Ilaria Matteucci. Synthesis of web services orchestrators in a timed setting. In *WS-FM*, pages 124–138, 2007.

[14] Rocco De Nicola and Matthew Hennessy. Ccs without tau's. In *TAPSOFT, Vol.1*, pages 138–152, 1987.

[15] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[16] Christian Skalka and Scott F. Smith. History effects and verification. In *APLAS*, pages 107–128, 2004.

[17] Gabriele Costa's web page. Secure service composition with symbolic effects (extended version), 2009. http://www.di.unipi.it/∼costa/data/papers/ssc_extended.pdf.

[18] Glynn Winskel. *The formal semantics of programming languages*. MIT Press, 1993.