Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Tesi di Laurea

Secure multi-party contracts for web-services

Davide Basile

Relatori:
Prof. Pierpaolo Degano, Prof. Gian Luigi Ferrari

Controrelatore:
Prof. Giorgio Levi

Anno Accademico 2010/2011

Contents

Secure multi-party contracts for web-services

Davide Basile

Corso di laurea specialistica in Informatica , Università di Pisa

relatori : Prof. Pierpaolo Degano, Prof. Gian Luigi Ferrari

Dipartimento di Informatica, Università di Pisa, Italy

Abstract

We consider two complementary formal approaches for describing services and their interactive behaviour.

The first approach is based on the notion of contracts. Contracts are CCS-like processes that contain a description of the external observable behavior of a service. A notion of compliance has been introduced allowing to check whether the interaction between two parties terminate or gets stuck. The second proposal is based on λ^{req} , a core calculus for services, with primitives for expressing security policies and for composing services in a call-by-contract fashion. In the dissertation we express CCS contracts via λ^{req} expressions and we prove that the proposed transformation preserves compliance of contracts, by exploiting the security mechanism of λ^{req} .

The transformation enjoys further properties. Multi-party and secure contracts are naturally handled. Moreover, the resulting notion of compliance is compositional: one can substitute a service with an equivalent one without breaking the security of the composition.

1 Introduction

In Service-Oriented Computing the applications are built by combining different distributed components, called services. Standard communication protocols are used for the interaction between the parties. Service composition depends on which information about the services is made public. Service can

be discovered according to that information. Security issues can make more complex service composition, since a service can impose constraints on the interactions it can hold; the most common example of the SOC paradigm is Web Services [4], whose languages for the description of their behavior and interactions are WSDL WSCL WSBPEL [5] .
In this paper we take two different paradigms for describe a service. On the one hand there are contracts which are CCS processes [6] that contain a description of the external observable behavior of a service. Contracts can describe if the interaction between two parties terminate or gets stuck, with a mechanism for replacing one service with a more general one. On the other hand we have λ^{req} [2] [7] which is a core calculus for services that extends the original λ-calculus [8] with primitives for composing services in a call-by-contract fashion with the enforcement of safety properties. Indeed there is a mechanism for extracting the abstract behavior of a service (called history expression) in which we impose that policies. An orchestration machinery constructs a plan that is a binding between requests and services guaranteeing that the safety properties are always satisfied.
Our work consists in expressing CCS contracts in λ^{req} formalism. The task is not trivial since circular requests are forbidden in λ^{req} (two services cannot invoke each other to serve a single request). This make it hard describing sessions between two parties, a feature available in contracts instead. In the original proposal [3] polices are regular expression that describe the operations not permitted to services. Instead the contracts of [3] are trees. Here we propose a new type of policies that describe the behavior that a service must respect; another extension concerns defining the notions of external and internal choice in λ^{req} . This operators are used in contracts to describe collaborations between the parties.
Our results are: a formalism to define security policies over contracts and to constrain the structure of interactions between services. Multi-party collaboration between contracts (a feature not enable in [3]). A mechanism to define sessions between λ^{req} services. A mechanism for replace a λ^{req} service with another one without breaking the correctness of the composition, and a rule system for checking if a λ^{req} expression respects a contract. Moreover our solution solved the problem of deciding the service who has the priority in a session where both the parties wait each other.

2 Contracts

In this section we define contracts, which are processes of a calculus introduced in [3] for formalize reasoning on services. Contracts are an abstractions of the interactions of services, used to express when those interactions succeed. Contracts are a subset of CCS processes built with three operators: prefix $\alpha.\sigma$, internal choice $\sigma_1 + \sigma_2$ and external choice $\sigma_1 \oplus \sigma_2$. A contract $\alpha.\sigma$ perform the action α and then continues as σ, where α in an abstraction of an operation that involves two services. We have write or read action with the former ones being topped by a bar, i.e. $\overline{\alpha}$, following the CCS notation. The contract $\sigma_1 + \sigma_2$ describes a service that lets the client decide whether to continue as σ_1 or σ_2. The contract $\sigma_1 \oplus \sigma_2$ describe a service that internally decides whether to continue as σ_1 or σ_2. In [3] there is a semantic model for recursion, where contracts are regular tree, instead we use recursive contracts which represents a syntactic approach with explicit recursion. We must use regular contract since for our transformation we need a finite representation of the infinite tree described by a contract.

Definition 1. *A recursive contract is a finite term generated by the following grammar :*

$$\sigma ::= a.\sigma \mid \overline{a}.\sigma \mid \mathsf{rec}\ x = \sigma \mid (\textstyle\sum_{i\in I} \sigma_i) \mid (\bigoplus_{i\in I} \sigma_i) \mid 0 \mid x \quad a, \overline{a} \in \mathcal{N}$$

where $\mathcal{N}$ is a countable set of names, I is a finite set of index and every variable x is guarded by at least one prefix constructor.

We require contracts to be guarded for ruling out meaningless contracts like $\mathsf{rec}\ x = x + x$. We write $\sum_{i\in I} \sigma_i$ for $\sigma_1 + \sigma_2 + \cdots + \sigma_n$ and $\bigoplus_{i\in I} \sigma_i$ for $\sigma_1 \oplus \sigma_2 \oplus \cdots \oplus \sigma_n$ where $|I| = n$, with $(\sigma, +)$ and $(\sigma, \oplus)$ abelian groups with identity 0 and $+,\oplus$ idempotent; note that $\sum_{i\in I} \sigma_i = \bigoplus_{i\in I} \sigma_i = 0$ if $I = \emptyset$, rec is a binder with variable x, we assume the term are α-convertible.
We introduce some standard notions. We say that a contract σ is closed if $fv(\sigma) = \emptyset$ where $fv(\sigma)$ is the set of free variables and we write $\sigma\{^{\sigma'}/_x\}$ for the contract obtained by replacing σ' for every free occurrence of x within σ. Contracts are used to ensure that interactions betweens client and service always succeeds. This happens when a service offers some set of actions and the client can synchronize with one of them performing the corresponding co-actions or terminating. Given a service contract, it is possible to determinate the set of clients that complies (written $\dashv$) with it, that are those that successfully terminates the interaction with the service. For example $a \oplus \overline{b} \dashv \overline{a} + b$ and also $a \oplus \overline{b} \dashv \overline{a} + b + c$, moreover $a \dashv \overline{a} + b$ but $a \not\dashv \overline{a} \oplus b$ since

$\overline{a} \oplus b$ can internally decide to perform b and the interaction gets stuck . As seen before, a service can comply with another one that offers more actions that those the service need. There is then a sub-contract relation, written $\sigma_1 \preceq \sigma_2$, expressing that the services that comply with σ_1 also comply with σ_2. We can replace a service with a one that offers more choices, allowing more composition and reuse of services. For example $a \preceq a + b$ (width sub-typing) or $a \preceq a.b$ (depth sub-typing), and with a one that is more deterministic, for example $a \oplus b \preceq a$,
For safely replacing a service with a more general one , we use a filter that blocks some not needed actions. For example, we have $a \oplus b.\overline{c} \preceq a$ and $a \preceq a + b.\overline{d}$, but $a \oplus b.\overline{c} \not\preceq a + b.\overline{d}$: if the service synchronize on b than he wait for the action $\overline{c}$ while the other party is ready to perform $\overline{d}$. This problem is solving by applying a filter on the new service that only permits the a action. We conclude with an example and we refer to [3] for more details of contracts and filter:

Example 2.0.1
We model an authentication on a service via login. After the service received the login, internal checks if the user name is valid and return the message of Valid login or Invalid login. In the case the user is authenticated the service continues with σ_1' :

$$\sigma_1 = Login.(\overline{ValidLogin}.\sigma_1' \oplus \overline{InvalidLogin})$$

The following service sends the login action and then waits for the authentication message. If the login is valid then it continues with $\overline{\sigma_1'}$:

$$\sigma_2 = \overline{Login}.(ValidLogin.\overline{\sigma_1'} + InvalidLogin)$$

3 Call-by-contract and secure service orchestration with λ^{req}

λ^{req} is a typed extension of the λ-calculus for service orchestrations, services are modelled as expressions of λ^{req} , where types are an abstractions of the behaviors of services. We assume as given a set of primitive *access events*, that abstract from activities with possible security concerns. Security policies are regular properties of execution histories (i.e. sequences of access events). There is a type and effect system for the calculus; types are standard, while

effects, called *history expressions*, represent all the possible behavior of services, also there is a mechanism to extract the abstract behavior of services that must enforce safety properties; given an expression e, a *safety framing* $\varphi[e]$ enforces the policy φ at each step of the execution of e. We have plans which are possible orchestrations, there is a static analysis technique to determine plans that drive service executions enjoying safety properties. A service is modelled as a λ^{req} expressions with a functional type of the form $\tau_1 \xrightarrow{H} \tau_2$; when supplied with an argument of type τ_1, the service evaluates to a value of type τ_2, and the side effect of the invocation is an execution history among those represented by the history expression H. A service request is modelled by an expression $\mathtt{req}_r\tau$, where r uniquely identifies the request, and τ is the type of the requested service, including the safety properties that explain how the caller protects itself from the service. We assume here that services are published in a global trusted repository, i.e. a set of typed expressions $\{e_1 : \tau_1 \xrightarrow{H_1} \tau_1' \cdots e_k : \tau_k \xrightarrow{H_k} \tau_k'\}$. Types are the semantic information made visible about services. Operationally, a service request $\mathtt{req}_r\tau$ results in a sort of "call-by-contract": the repository is searched for a service with a functional type matching the request type τ; additionally, its effect H should respect the safety constraints in τ. The effect of a service invocation $\mathtt{req}_r\tau$ has the form $\{r[\ell_1] \rhd H_1 \cdots r[\ell_k] \rhd H_k\}$, where $r[\ell_i]$ resolves the request r with the service e_i in the repository. We say that H_i is *valid* when it respects the safety constraints in τ, as well as all the security framings within H_i itself. The effect H of a service composition e is obtained by suitably assembling the effects of the component services, and of those services they may invoke in a nested fashion. The validity of the effect H of e depends thus on the global orchestration that selects a service for each request. It is convenient to lift all the service choices $r[\ell]$ to the top-level of H, collecting them in a set π, called *plan*, with a semantic-preserving transformation that generates effects of the form $\{\pi_1 \rhd H_1 \cdots \pi_n \rhd H_n\}$, where each H_i is free of further choices. Its intuitive meaning is that, under the plan π_i, the effect of the overall service composition e is H_i. If some H_i is valid, then the plan π_i will safely drive the execution of e, without resorting to any run-time monitor, and guaranteeing all the safety properties required. Validity of history expressions is ascertained by model checking Basic Process Algebras with finite state automata; a history expressions H is naturally rendered as a BPA process, while a finite state automaton models the validity of H.

3.1 λ^{req} expressions

In λ^{req} an *access event* $\alpha \in \mathsf{Ev}$ abstracts from a security critical operation (e.g. writing a file, opening a socket connection). A *history* η is a sequence of access events. A *security policy* $\varphi \in \mathsf{Pol}$ is a regular property of histories. A *safety framing* $\varphi[e]$ enforces the policy φ at each step of the evaluation of e. Services $e : \tau$ are typed λ^{req} expressions, collected in a trusted, finite, and global repository Srv . The types τ are annotated with *history expressions* that over-approximate the possible run-time histories. The repository Srv guarantees that H represents all the possible histories of e.

A *service request* has the form $\mathtt{req}_r\tau$. The label r uniquely identifies the request in an expression; the syntax of request types τ is defined as follows:

$$\tau \quad ::= \quad 1 \mid \tau \xrightarrow{\varphi} \tau$$

where 1 is the singleton type, and the annotations φ on the arrow are the safety constraints imposed on the service. Operationally, $\mathtt{req}_r\tau$ drives a search in the repository Srv for a service with a functional type τ' "compatible" with τ and such that τ' respects the constraints imposed by τ. Only functional types are allowed in a request: this models services being considered as remote functions; if the type of a returned value is functional, then the request can be seen as a code download, moreover no constraints should be imposed over the type τ_0 of a request type $\tau_0 \xrightarrow{\varphi} \tau_1$, i.e. in τ_0 there are no annotations. This is because the constraints on the selected service cannot affect its argument. It follows the syntax of a λ^{req} expression, we abstract from the language for express φ and guards b:

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

The values v are the variables, the abstractions, the requests, and the distinguished element $*$. The following abbreviation is standard: $e; e' = (\lambda.\, e')\, e$, the variable z in $e' = \lambda_z x.\, e$ stands for e' itself within e.
A plan formalises how a request is resolved into an actual service, and takes the form of an injective mapping from request labels to services. Plans have the following syntax:

$$
\begin{array}{rcll}
\pi, \pi' & ::= & 0 & \text{empty} \\
 & & r[\ell] & \text{service choice} \\
 & & \pi \mid \pi' & \text{composition}
\end{array}
$$

The empty plan 0 has no choices; the plan $r[\ell]$ associates the service $e_\ell : \tau_\ell$ with the request labelled r. Now we define the behaviour of expressions through the following small-step operational semantics. The configurations are pairs η, e. Transition have the form $\eta, e \to_\pi \eta', e'$ to mean that, starting from a history η, the plan π allows the expression e to evolve to e' and to extend η to η'. An expression is initially evaluated starting from the empty history ε. We write $\eta \models \varphi$ when the history η obeys the (safety/liveness) policy φ. We assume as given a total function $\mathcal{B}$ that evaluates the guards in conditionals.

Operational semantics of λ^{req}

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

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

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

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

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

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

To evaluate a safety framing $\varphi[e]$, we must consider two cases. If, starting from the current history η, e may evolve to e' and extend the history to η', then the whole framing $\varphi[e]$ may evolve to $\varphi[e']$, provided that η' satisfies φ. Otherwise, if e is a value and the current history satisfies φ, then the scope

of the framing is left. In both cases, as soon as a history is found not to respect φ, the evaluation gets stuck, to model a security exception.

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

3.2 History expression

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

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

Safety framings are the abstract counterparts of the analogous constructs in λ^{req}. Given a plan π, a planned selection $\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$ chooses those H_i such that π includes π_i. Intuitively, the history expression $H = \{r[\ell_1] \triangleright H_1, r[\ell_2] \triangleright H_2\}$ is associated with a request r that can be resolved into either e_{ℓ_1} or e_{ℓ_2}. The histories denoted by H depend on the given plan π: if π chooses ℓ_1 (resp. ℓ_2) for r, then H denotes one of the histories represented by H_1 (resp. H_2); otherwise, H denotes no histories.
The denotational semantics of H is the set of histories $\mathcal{H}$ that represents all the possible computations of the λ^{req} expression . An history expressions is valid if none of the possible computations violate some policy φ, where the policy can inspect the whole history generated so far, a history-expression is π valid if under that plan H is valid.

3.3 Type and effect system and planning

We now introduce a type and effect system for our calculus. Types and type environments, ranged over by τ and Γ, are mostly standard and are

defined in the following table. The history expression H in the functional type $\tau \xrightarrow{H} \tau'$ describes the latent effect associated with an abstraction, i.e. one of the histories represented by H is generated when a value is applied to an abstraction with that type. Note that we overload the symbol τ to range over both expression types and request types $\tau \xrightarrow{\varphi[\bullet]} \tau'$.

Types and Type Environments

$$\tau, \tau' ::= 1 \mid \tau \xrightarrow{H} \tau' \qquad \Gamma ::= \emptyset \mid \Gamma; x : \tau \quad (x \notin dom(\Gamma))$$

Typing judgment are in the form $\Gamma, H \vdash_{\mathsf{Srv}} e : \tau$ meaning that, given a service repository Srv, the expression e evaluates to a value of type τ, and produces a history represented by the effect H.
The typing relation $\Gamma, H \vdash_{\mathsf{Srv}} e : \tau$ is defined as the least relation closed under the rules below. Typing judgments are similar to those of the simply-typed λ-calculus. The effects in the rule for application are concatenated according to the evaluation order of the call-by-value semantics (function, argument, latent effect). The actual effect of an abstraction is the empty history expression, while the latent effect is equal to the actual effect of the function body. The rule for abstraction constraints the premise to equate the actual and latent effects, up to associativity, commutativity, and idempotency of $+$, associativity and zero (ε) of $\cdot$, α-conversion, unfolding of recursion, and elimination of vacuous μ-binders. The last rule allows for *weakening* of effects. A service invocation $\mathtt{req}_r\tau$ has an empty actual effect, and a functional type τ', whose latent effect is a planned selection that picks from Srv those services matching the constraints on the request type τ. To give a type to requests, we need to define the auxiliary operators $\approx$, $\oplus$ and $\Cup$. We write $\tau \approx \tau'$, and say τ, τ' *compatible*, whenever, omitting the annotations on the arrows, τ and τ' are equal. The operator $\oplus_{r[\ell]}$ combines a request type τ and a service type τ', when they are compatible. Eventually, the operator $\Cup$ combines the types obtained by combining the request type with the service types.

Typing relation

$$\Gamma, \varepsilon \vdash * : 1 \quad \text{(T-Unit)} \qquad \Gamma, \alpha \vdash \alpha : 1 \quad \text{(T-Ev)}$$

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

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

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

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

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

Once extracted a history expression H from an expression e, we have to analyse H to find if there is any viable plan for the execution of e. Here we only note that a plan π is well-typed if there are no circular request, that is given a service at location ℓ, we have that for all $r[\ell']$ in π we have $\ell \prec \ell'$.

3.4 Networks

A service e is plugged into a network by publishing it at a site ℓ, together with its interface τ. Hereafter, $\ell\langle e : \tau \rangle$ denotes such a *published service*. Labels ℓ can be seen as Uniform Resource Identifiers, and they are only known by the orchestrator. We assume that each site publishes a single service, and that interfaces are certified, i.e. they are inferred by the type system presented later. As usual, we assume that services cannot invoke each other circularly. A *client* is a special published service $\ell\langle e : 1 \rangle$, where 1 is the unit type. A *network* is a set of clients and published services.

The *state* of a published service $\ell\langle e : \tau \rangle$ is denoted by:

$$\ell\langle e : \tau \rangle : \pi \rhd \eta, e'$$

$$[\textsc{Sta}] \qquad \frac{\eta, e \to \eta', e'}{\ell : \pi \rhd \eta, e \to \ell : \pi \rhd \eta', e'}$$

$$[\textsc{Net}] \qquad \frac{N_1 \to N_1'}{N_1 \parallel N_2 \to N_1' \parallel N_2}$$

$$[\textsc{Req}] \qquad \begin{array}{c} \ell : (r[\ell'] \mid \pi) \rhd \eta, \mathcal{C}(\mathtt{req}_r \rho\, v) \parallel \ell' \langle e : \tau \rangle : 0 \rhd \varepsilon, * \to \\ \ell : (r[\ell'] \mid \pi) \rhd \eta, \mathcal{C}(\mathtt{wait}\, \ell') \parallel \ell' \langle e : \tau \rangle : (r[\ell'] \mid \pi) \rhd \sigma, e\, v \end{array}$$

$$[\textsc{Ret}] \qquad \begin{array}{c} \ell : \pi \rhd \eta, \mathtt{wait}\, \ell' \parallel \ell' : \pi' \rhd \eta', v \to \\ \ell : \pi \rhd \eta, v \parallel \ell' : 0 \rhd \varepsilon, * \end{array}$$

where π is the plan used by the current instantiation of the service, η is the history generated so far, and e' models the code in execution. When unambiguous, we simply write ℓ for $\ell\langle e : \tau \rangle$ in states.

The syntax and the operational semantics of networks follows; the operator $\parallel$ is associative and commutative. Given a network $\{\ell_i \langle e_i : \tau_i \rangle\}_{i \in 1..k}$, a *network configuration* N has the form:

$$\{\ell_i : \pi_i \rhd \eta_i, e_i'\}_{i \in 1..k} = \ell_1 : \pi_1 \rhd \eta_1, e_1' \parallel \cdots \parallel \ell_k : \pi_k \rhd \eta_k, e_k'.$$

To trigger a computation of the network, we single out a set of *initiators*, and fix the plans π_i for each of them. We associate the empty plan to the other services. Then, for all $i \in 1..k$, the initial configuration has $\eta_i = \varepsilon$, and $e_i' = *$ if ℓ_i is a service, while $e_i' = e_i$ if ℓ_i is an initiator.

We now comment on the semantic rules of networks in λ^{req} . A transition of a stand-alone service is localized at site ℓ (rule Sta), regardless of a plan π. The rule Net specifies the asynchronous behaviour of the network: a transition of a sub-network becomes a transition of the whole network. The rules Req and Ret model successful requests and replies. A request r, resolved by the current plan with the service ℓ', can be served if the service is available, i.e. it is in the state $\ell' : 0 \rhd \varepsilon, *$. In this case, a new activation of the service starts: e is applied to the received argument v, under the plan π', received as well from the invoker. The special event σ signals that the service has started. The invoker waits until ℓ' has produced a value. When this happens, the service becomes idle again. Since we follow here the *stateless* approach, we clear the history of a service at each activation (indeed, statefullness could be easily obtained by maintaining the history η' at ℓ' in the last rule).

Note that each service has a single instance in network configurations. We could easily model replication of services, by creating a new instance for each

request. Note also that a network evolves by interleaving the activities of its components, which only synchronize when competing for the same service. For further informations we refer to [1] [2].

4 Getting into the problem

In this section we discuss the issues of expressing contracts in λ^{req} by developing examples which let emerges the limitations of the approach followed. We first treat the problem of transforming input and output operations of contracts into λ^{req} expressions and policies, and we introduce a mechanism for expressing sessions in λ^{req} . After that we discuss about the interpretation of internal and external choice in λ^{req} . We then define a transformation of contracts for simplifying the successive translations in λ^{req} and ensure that at runtime the λ^{req} expressions behave as described by the corresponding contracts. Finally we discuss the validation of contracts.
In sections 5 and 6 we formally define a method to generate λ^{req} expressions from contracts; this will allow us to introduce multi-party contracts and secure service substitution, composition and reuse, as we will discuss in section 7.

4.1 Input/Output actions

First thing to decide is what is a contract in λ^{req} . We could express a contract as a new type of policy φ_σ, identical to a contract σ and with the same meaning that describe the action that a service must perform. Then we could write $\varphi_\sigma[H]$ to check if the service with effect H satisfies the contract σ, that is H is compliant with σ. One limitation of this approach emerges when we deal with the interplay between read/write actions. A read action represents a data provided in input to the service, while a write action represents a data obtained as output from the service, to clarify this fact let us consider the contracts:

$\sigma_1 = \underline{Login}.(\overline{ValidLogin}.\sigma_1' \oplus \overline{InvalidLogin})$
$\sigma_2 = \overline{Login}.(ValidLogin.\sigma_1' + InvalidLogin)$

Here σ_1 receives in input the login data, checks if this login is valid and then responds with the valid/invalid message, while σ_2 offers the complementary actions. In a λ^{req} expression we have only access events without distinguishing between input and output actions, and policies only check if the history

H fulfills the requirement without performing any action. If we translate a contract completely in a policy φ_σ we loose the run-time actions that a contract is supposed to perform. Therefore we need to translate contracts both as λ^{req} expressions and use φ_σ to check if they are compliant. Moreover, we see that a contract specifies a *session* between a client and a server, i.e. when the server receives the login it sends back the valid/invalid message. Taking into account these considerations we shall write the following λ^{req} service, note that in σ_2 we use a higher-order function f that represents the value returned by req_{r1}[1]:

Example 4.1.1

$$\begin{aligned}
\sigma_2 = (\lambda f.&\text{if } ((f*) == \text{Valid}) \text{ then} \\
&req_{r2}(\cdots)(\overline{\sigma_1'}) \ ) \\
&(req_{r1}(1 \to (1 \xrightarrow{\varphi_\sigma} 1))(Login))
\end{aligned}$$

$$\begin{aligned}
\sigma_1 = &\lambda x.\varphi'_\sigma[x]; \\
&\text{if (x==ValidLogin) then} \\
&\quad \lambda.ValidLogin; \\
&\text{else} \\
&\quad \lambda.InvalidLogin
\end{aligned}$$

$$\varphi'_\sigma = Login$$
$$\varphi_\sigma = ValidLogin + InvalidLogin$$

In this example we transform output actions in λ^{req} expressions and input actions in φ_σ. In σ_2 first the service perform the request req_{r1}, the argument of the request is the *Login* action, which is an output action; the policy φ_σ checks that the selected service complies with the contract. At static time the orchestrator selects the right contract that complies with σ_2 that is σ_1. If the login is valid then the service continue performing another output action, so generating a new request req_{r2} as required by $\overline{\sigma_1'}$.
The service σ_1 is chosen to take the request req_{r1}. After having received the value *Login* (in fact this is an input action in the contract), the service inserts the value inside the frame φ_σ to statically checked that the two services are compliant. This check yelds the $ValidLogin$ or $InvalidLogin$. Both are

[1]We not specify the effects of a λ^{req} expressions if is unneeded

output actions that became read actions by the side of σ_2 and are used in the if guard; the history generated by req_{r1} is:

$$req_{r1} : 1 \xrightarrow{\{r[\ell_{\sigma_1}] \triangleright \ell_{\sigma_1} : \varphi'_\sigma[Login]\}} (1 \xrightarrow{\{r[\ell_{\sigma_1}] \triangleright \varphi_\sigma[(ValidLogin + InvalidLogin)]\}} 1).$$

The histories generated in the locations of the two services are:

$(\ell_{\sigma_1} : \varphi'_\sigma[Login])$
$(\ell_{\sigma_2} : \varphi_\sigma[(ValidLogin + InvalidLogin)] \cdot H_{\overline{\sigma'_1}})$

These histories are valid since the actions do not violate policies.
The main problem of this approach is that circular requests cannot be specified, so contract can only perform a two-message interaction, after received the $ValidLogin$ message, the second request cannot be taken by the service σ_2. Moreover a contract is not completely rendered by a policy φ_σ or by a history expression H, but it is split into both a policy φ_σ, for the input actions, and a history expression H for the output actions. Another issue that emerges in the example is that a contract σ can be translated into a client that performs the request, or into a server that is planned to serve the request. This two roles lead to different translations of a contract. One important result is that in λ^{req} we can perform many request as we want, and even the server side can perform a request while he is serving another contracts. This give us a notion of multi-contract party that is not available in the contracts of [3].

4.2 Sessions

Since we cannot explicitly define a session in λ^{req} , we can model it as a migration of code. The value of the request is the entire code of the client, that is then executed by the server side. For doing so we embed every action of the client contract into a λ-abstraction that the server will apply while interacting with the client. We will also use a λ-abstraction where a local variable will be used to represent the external choice: the server will make its decision by applying that function.
We expect a system for encrypt this code by the orchestrator if one want to protect the code, even if this is not significant since in that code are present only the actions that the client want to execute in the server, that is the output actions and the Server must have the rights to read those informations. Let us see how the previous example becames now:

Example 4.2.1

$$\sigma_2 = req_{r1}(1 \to (1 \to (1 \to 1)) \xrightarrow{\varphi_\sigma} 1)((\lambda cx.\text{if } (cx{=}{=}\text{ValidLogin}) \text{ then } \lambda.\overline{\sigma_1'}) \\ (\lambda.Login))$$

$$\sigma_1 = \lambda f_0.(\lambda f_1.\text{if } (f_1 == Valid) \text{ then } (\lambda f_2.(\sigma_1'))(f_1 ValidLogin) \\ \text{else } (\lambda f_2.(*))(f_1 ValidLogin)) \\ (\varphi'_\sigma[f*])$$

The histories expression generated are :

$(\ell_{\sigma_1} : \varphi_\sigma[\varphi'_\sigma[Login] \cdot (ValidLogin \cdot H_{\sigma_1'} + InvalidLogin)])$
$(\ell_{\sigma_2} : \varepsilon)$

As we see now the frame φ'_σ is nested inside φ_σ. This is not a problem since φ_σ checks only that the action required are performed. In this example we want that $ValidLogin$ or $InvalidLogin$ are executed, and the action $Login$, that is relevant for φ'_σ, is ignored by φ_σ.
By executing the two contracts on the Server side we have not circular request, now not only two-message interactions are allowed, but after validating the login the two services can continue to perform the other actions required by the contract σ_1'.
Note that we can naturally describe multi-party scenario. For example for validating the login the server may need to make a request to an authority for checking that the login is valid, so it behaves from one side as a server and on the other side as a client, performing a request before returning the $ValidLogin$ or $InvalidLogin$ message. In section 7 will examine multi-party scenarios in details.

4.3 Internal and external choice

In the previous example, the contract $ValidLogin \oplus InvalidLogin$ were translated into history expressions of the form $Valid + Login$.
A problem that emerges is that in history expression we have only the non

deterministic choice $H_1 + H_2$, that expresses the history generated by an expression $H_1 + H_2 \vdash \texttt{if}\, b\, \texttt{then}\, e\, \texttt{else}\, e' : \tau$ with $H_1 \vdash e : \tau$ and $H_2 \vdash e' : \tau$. But in contracts we have two different operators, internal and external choice, with different meanings.
In a first attempt to solve this issue we shall say that an history H comply with a contract $\sigma = \sigma_1 \oplus \sigma_2$ if it has at least both H_1 and H_2 in the non deterministic choice, i.e. is in the form $H_1 + H_2$ or $\sum_{i \in I} H_i$ where $|I| > 1$ and $\forall i, j \in I, i \neq j . H_i \dashv \sigma_i \wedge H_i \not\dashv \sigma_j$, for example $H_1 + H_3$ do not comply with σ since if the service internally decide σ_2 the other service described by $H_1 + H_3$ get stuck. We shall say that an history H complies with a contract $\sigma = \sigma_1 + \sigma_2$ if H contains H_1 or H_2 or both, since is the service H that decide whether to continue as H_1 or H_2. A problem of this approach is that there are contracts, like $\sigma = \sigma_1 \oplus \overline{\sigma_2}$ and $\sigma = \overline{\sigma_1} \oplus \sigma_2$, that do not comply since one can internally decides a branch while the other internally decides the other branch and the interaction get stuck. Writing this contract in λ^{req} we have $\varphi_\sigma = \sigma_1 \oplus \sigma_2$ and $H = H_1 + H_2$, so following this approach the services wrongly complies; since in H we don't have the possibility to describe an internal choice, so we must extend history expression introducing internal and external choice if we want H to describe a contract.
We have to refine the if rule of the type and effect system to make history expression able to distinguish between internal and external choice, we now define internal and external choice in history expressions:

Definition 2. *Let H_1 and H_2 be history expressions, we denote $H_1 + H_2$ as the external choice between the two histories, and $H_1 \oplus H_2$ as the internal choice between the two histories.*

We note that in an external choice another service is choosen by a resorting on a value provided by another service; while in the internal choice the service decide itself. Under this assumption we already have the internal choice and we need to extend the rule to the external choice. To avoid confusion from now onwards we follow the notation of contracts and we use the operator $\oplus$ for the internal choice and $+$ for the external choice.
As described above, there are two different transformations of a contract into a λ^{req} expression: we call there the client and the server transformation. We formally set up the framework by giving the definitions of the internal and external choice. In this way we get the λ^{req} expression corresponding to the contract.
We first discuss the λ^{req} transformation for the client side. The external choice is modelled through a λ-abstraction. When evaluating this expression the server makes its choice by giving the value cx, while in the internal choice we abstract from the boolean guard: the client part makes its choice

to continue as σ_1 or σ_2.

Definition 3. *Let σ_1 and σ_2 be contracts and e_1, e_2 the corresponding λ^{req} client transformations where $H_1 \vdash e_1 : \tau$ and $H_2 \vdash e_2 : \tau$, we define the client transformation in λ^{req} for the external and internal choice as:*

$$H_1 + H_2 \vdash_{client} \lambda cx.if^{+}\ (\ cx = \sigma 1)\ then\ e_1\ else\ e_2$$

$$H_1 \oplus H_2 \vdash_{client} if^{\oplus}\ (\cdots)\ then\ e_1\ else\ e_2$$

We give now the λ^{req} expressions for the internal and external choice by the server side:

Definition 4. *Let σ_1 and σ_2 be contracts and e_1, e_2 the corresponding λ^{req} server transformations where $H_1 \vdash e_1 : \tau$ and $H_2 \vdash e_2 : \tau$, we define the server transformation in λ^{req} for the external and internal choice as:*

$$H_1 + H_2 \vdash_{server} \lambda f.if^{+}\ (\ f = \sigma_1)\ then\ e_1\ else\ e_2$$

$$H_1 \oplus H_2 \vdash_{server} if^{\oplus}\ (\cdots)\ then\ e_1\ else\ e_2$$

That transformations resemble those given for the client contract. The main difference lies in the external choice where the server checks the boolean

guard depending on the client choice. We write $f = \sigma 1$ to indicate the abstraction step of this control. Note that the function f is the client code and it is the value of the request.
We write if^{+} and $if^{\oplus}$ to make clearer the history expression generated. It is useful if we want to express in the type and effect system the rule for internal and external choice, however in this work we are only interested in the transformation of contracts into λ^{req} expression and not vice-versa. We feel free to omit the specifications of the type of the if condition.

4.4 Dual actions

Assume that the client contract is of the form $a \oplus b$. Intuitively, the client makes an internal choice and decides whether to read from channel a or b. In this case the server (abstracted by a λ^{req} expression) should be able at runtime to see which action the client is ready to receive to perform the required co-action. Hence we cannot only render input actions as policies, since they are only checked at static time. Another problem emerges in the following contract:
$Login.(\overline{ValidLogin}.\sigma_1' \oplus \overline{InvalidLogin})$ here the internal choice depends on the input action $Login$, in the previous example the boolean guard check if $Login$ is valid or not. However in our λ^{req} specification of a contract we abstract from this level of detail and we are only interested in modelling the values exchanged by the two parties when deciding the external choice.
We introduce a transformation of the contract with the aim to erase the difference between an input operation or an output operation. The main idea is to associate with each channel[2] c a dual channel dc where no relevant information is exchanged. For example, a contract like $\overline{a} + b$ is transformed into $\overline{a}.da + \overline{db}.b$. The client contract is the initiator, so his first operation to be performed is always the write action while in the server contract we have the opposite. Also, when a service receives a data on input it always reacts with the dual action on output. Note that when the client is of the form $a \oplus b$ the server is able to decide the right co-action. We now prove that this transformation preserves the compliance:

Lemma 1. *Let σ_{client} and σ_{server} be contracts and σ^d_{client},σ^d_{server} the corresponding transformed contracts with dual actions, we have: $\sigma_{client} \dashv \sigma_{server} \longrightarrow \sigma^d_{client} \dashv \sigma^d_{server}$*

Proof. Since $\sigma_{client} \dashv \sigma_{server}$ then σ_{client} is able to terminate or the two contracts synchronizes on an action α. In the transformed contracts, we only

[2]we refer to channel as the action of a CCS contract

change the prefix operators adding dual actions. If $\sigma_{client} = \alpha.\sigma'_{client}$ and $\sigma_{server} = \overline{\alpha}.\sigma'_{server}$ then $\sigma^d_{client} = \overline{d\alpha}.\alpha.\sigma'^d_{client}$ and $\sigma^d_{server} = d\alpha.\overline{\alpha}.\sigma'^d_{server}$. We have $\sigma^d_{client} \dashv \sigma^d_{server}$.
If $\sigma_{server} = \alpha.\sigma'_{server}$ and $\sigma_{client} = \overline{\alpha}.\sigma'_{client}$ then $\sigma^d_{client} = \overline{\alpha}.d\alpha.\sigma'^d_{client}$ and $\sigma^d_{server} = \alpha.\overline{d\alpha}.\sigma'^d_{server}$. We have $\sigma^d_{client} \dashv \sigma^d_{server}$. □

In the following example we see how the contracts of the previous example are transformed:

Example 4.4.1
$\sigma_{server} = Login.(\overline{ValidLogin} \oplus \overline{InvalidLogin})$
$\Rightarrow Login.\overline{Dlogin}(DvalidLogin.\overline{ValidLogin} \oplus DinvalidLogin.\overline{InvalidLogin})$

$\sigma_{client} = \overline{Login}.(ValidLogin + InvalidLogin)$
$\Rightarrow \overline{Login}.Dlogin.(\overline{DvalidLogin}.ValidLogin + \overline{DinvalidLogin}.InvalidLogin)$

The transformed contracts are compliant, so this is only a syntactic manipulation and the behavior of the contracts remains the same. This transformation has the property that each action is performed both in input and in output, on one side as a dual operation and on the other one as a normal operation. So this action is present in both φ_σ and H.
Continuing the previous example we have:

Example 4.4.2
$H_{server} = Dlogin(ValidLogin \oplus InvalidLogin)$
$\varphi_{server} = Login.(DvalidLogin \oplus DinvalidLogin)$

$H_{client} = Login.(DvalidLogin + DinvalidLogin)$
$\varphi_{client} = Dlogin(ValidLogin + InvalidLogin)$

$H_{server} \models \varphi_{client}$
$H_{client} \models \varphi_{server}$

Note that in H we only have write actions and in φ_σ we only have read actions, hence we can avoid to specify the type of the actions. Also we assume that if the action occurs in H then the dual action occurs in φ_σ and vice-versa, so we abstract from specifying dual actions. With this abstraction H and φ_σ are the same and when we are going to transform a contract

in a λ^{req} expression we do not have to do any extra work for handling the different type of actions.
In the following example we compute the λ^{req} expressions of the previous one. We use the following shorthands $e1; x.e2 = (\lambda x.e2)(e1)$ and $*_e$ for an expression e with no effects.

Example 4.4.3

Client : $Login.(DvalidLogin.\sigma'_1 + DinvalidLogin)$

$$req_{r1}(1 \to (1 \to (1 \to 1)) \xrightarrow{\varphi_{client}} 1)(\lambda.Login;$$
$$\lambda cx.\text{if } (cx == *_{DvalidLogin}) \text{ then}$$
$$DvalidLogin; \lambda.\sigma'_1$$
$$\text{else}$$
$$DinvalidLogin; \lambda.*)$$

Server: $Dlogin(ValidLogin.\sigma'_1 \oplus InvalidLogin)$

$$\lambda f_0.(\varphi^0_{server}[f_0*]; f_1.(Dlogin;$$
$$\text{if } (\cdots) \text{ then}$$
$$\varphi^1_{server}[f_1 *_{DvalidLogin}]; f_2.(ValidLogin; (\sigma'_1))$$
$$\text{else}$$
$$\varphi^1_{server}[f_1 *_{DinvalidLogin}]; f_2.(InvalidLogin; *)))$$

$$\varphi_{client} = Dlogin(ValidLogin + InvalidLogin)$$

$$\varphi^0_{server} = Login$$
$$\varphi^1_{server} = DValidLogin \oplus DInvalidLogin$$

$$H_{server} = \varphi_{client}[\varphi^0_{server}[Login] \cdot Dlogin \cdot$$
$$(\varphi^1_{server}[DvalidLogin + DinvalidLogin] \cdot ValidLogin \cdot \sigma'_1 \oplus$$
$$\varphi^1_{server}[DvalidLogin + DinvalidLogin] \cdot InvalidLogin)]$$

Note that dual actions don't interfere with the validations, since we discard the actions not required by φ_σ. The only if guard specified is the one of the external choice of the client, while the boolean guard that checks if the login is valid is hidden from the specifications. Finally we need to split φ_{server} into different parts. We will discuss this point further on.
Note that in a contract an action can be used both for read and write. For

example in $\sigma = a.b + \overline{a}.c$ we have $H = da.db + a.dc$ and $\varphi_\sigma = a.b + da.c$. We also have solved the problem [3] of who has priority in a session between two party like:

$\sigma_{client} = a + b$
$\sigma_{server} = a + b$

By the contract definitions, each service waits for the other to establish a decision and the interaction gets stuck. As showed in the following example, in a situation like this we determine that the server has the priority to perform the decision:

Example 4.4.4

Client:
$req_r(\cdots)(\lambda cx.\text{if } (cx == a) \text{ then } a$
$\qquad\qquad\qquad \text{else } b$

Server:
$\lambda f.\text{if } (f \cdots) \text{ then } \varphi_s[fa]; a$
$\quad \text{else } \varphi_s[fb]; b$

The Server is able to check in the boolean guard whether the Client has performed a decision or not. In the second case, the Server performs the decision by applying the function Client so the interaction continue without blocking.

4.5 Validating a contract

In λ^{req} a history expression is valid under a plan π [2] if the set of all the possible histories are recognized by the automata that describe the policy φ. Contracts are not strings of a language but they are trees. In particular in the internal choice a service must follow both the branch of the tree to comply with that contract. Hence we can't use automata to check the validity of an history expression under a policy φ_σ. Here, to address this issue we introduce a rule system to check if $H \models \varphi_\sigma$,and we demonstrate that is equivalent to say that $\sigma \dashv \sigma_H$ where φ_σ represent the contract σ and H represent the contract σ_H. A policy φ_σ only describes a set of strings of actions that a service

must performs before terminating, beside these actions it can perform other actions not required by the contract without violating the policy φ_σ. The scope of this kind of policy is only local to his frame and do not check all the past histories as done before by a normal policy φ. In the section 7 we will define the mechanism to validate history expressions over policies φ_σ.

4.6 Types and contracts

In a contract of the form $\sigma = \sigma_1 + \sigma_2$ we have that σ_1 and σ_2 are completely independent. When we translate this contract in λ^{req} we have the constraint that the two branch of the if construct, that are σ_1 and σ_2, must have the same type. In the following sections we define a method that taken a contract return the corresponding balanced contract: a contract with the properties that the translation in λ^{req} is well-typed.

5 Writing contracts in λ^{req}

In this section, we formally define a method to transform contracts into balanced contracts by using an intermediary representation called labeling contracts. Balanced contracts will be transformed into well-typed λ^{req} expressions. We assume that the contract to be transformed contains dual actions, in particular we select only write actions for the transformation in λ^{req} expression, and only read actions for the transformation in policy, as we discussed before.

5.1 Labeling contracts

Our first step is the introduction of a specific form of contracts called labeled contracts.

Definition 5. *Labeled contracts are special type of contracts generated by the following grammar:*[3]

$$\sigma^n ::= a^n.\sigma^{n'} \mid \mathsf{rec}\ x = \sigma^n \mid (\textstyle\sum_{i\in I} \sigma_i^{n_i})^n \mid (\bigoplus_{i\in I} \sigma_i^{n_i})^n \mid 0^n \mid x^n.\sigma^{n'} \quad n, n' \in \mathbb{N}$$

In a labeled contract each node is associated with a number that expresses the depth of the node. We can avoid labeled contract and calculate the depth directly, but we prefer this way for clearing the separation of concerns and trying to reduce the complexity. We have two different types of contracts:

[3]We use $x.\sigma$ instead of $x.0$ (even if σ is unreachable) because then we can add the unit event * after x if needed

client contracts and server contracts, and two different ways of transforming them into λ^{req} expressions. Accordingly, we define two different functions that compute the labeled client contract and the labeled server contract. In particular, we start by the root with label 0 and we increase the label every time we add a λ-abstractions in the λ^{req} expression that we are constructing. Note that technically the client contract anticipates the λ-abstractions for the actions of an external choice, and consequently the server anticipates the application of the function on input on the internal choice. We use a boolean guard to keep track if we have already set the λ-abstraction or not in the client contract, and if we already applied the function or not in the server contract. The function $\mathcal{L}$ take in input the contract to transform, the boolean guard and the temporary label, these two parameters are initially set to zero.

Definition 6 (Labeled Client Contract)**.** *Given a contract σ we define the corresponding labeled Client contract σ^n_{client} as : $\sigma^n_{client} = \mathcal{L}^c[\![\sigma]\!](0,0)$ where :*

$$\mathcal{L}^c : \sigma \longrightarrow \{0,1\} \times \mathbb{N} \longrightarrow \sigma^n_{client}$$

$\mathcal{L}^c[\![\sum_{i\in I} \sigma_i]\!](b,j) = (\sum_{i\in I} \mathcal{L}^c[\![\sigma_i]\!](1,j+1))^j \quad b \in \{0,1\}$
$\mathcal{L}^c[\![\bigoplus_{i\in I} \sigma_i]\!](b,j) = (\bigoplus_{i\in I} \mathcal{L}^c[\![\sigma_i]\!](b,j))^j \quad b \in \{0,1\}$
$\mathcal{L}^c[\![a.\sigma]\!](0,j) = a^j.\mathcal{L}^c[\![\sigma]\!](0,j+1)$
$\mathcal{L}^c[\![a.\sigma]\!](1,j) = a^j.\mathcal{L}^c[\![\sigma]\!](0,j)$
$\mathcal{L}^c[\![x.\sigma]\!](0,j) = x^j.\mathcal{L}^c[\![\sigma]\!](0,j+1)$
$\mathcal{L}^c[\![x.\sigma]\!](1,j) = x^j.\mathcal{L}^c[\![\sigma]\!](0,j)$
$\mathcal{L}^c[\![\mathsf{rec}\ x = \sigma]\!](b,j) = (\mathsf{rec}\ x = \mathcal{L}^c[\![\sigma_i]\!](b,j)) \quad b \in \{0,1\}$
$\mathcal{L}^c[\![0]\!](0,j) = 0^j$

Definition 7 (Labeled Server Contract)**.** *Given a contract σ we define the corresponding labeled Server contract σ^n_{server} as : $\sigma^n_{server} = \mathcal{L}^s[\![\sigma]\!](0,0) where$:*

$$\mathcal{L}^s : \sigma \longrightarrow \{0,1\} \times \mathbb{N} \longrightarrow \sigma^n_{server}$$

$\mathcal{L}^s[\![\sum_{i\in I} \sigma_i]\!](b,j) = (\sum_{i\in I} \mathcal{L}^s[\![\sigma_i]\!](b,j))^j \quad b \in \{0,1\}$
$\mathcal{L}^s[\![\bigoplus_{i\in I} \sigma_i]\!](b,j) = (\bigoplus_{i\in I} \mathcal{L}^s[\![\sigma_i]\!](1,j+1))^j \quad b \in \{0,1\}$
$\mathcal{L}^s[\![a.\sigma]\!](0,j) = a^j.\mathcal{L}^s[\![\sigma]\!](0,j+1)$
$\mathcal{L}^s[\![a.\sigma]\!](1,j) = a^j.\mathcal{L}^s[\![\sigma]\!](0,j)$
$\mathcal{L}^s[\![x.\sigma]\!](0,j) = x^j.\mathcal{L}^s[\![\sigma]\!](0,j+1)$
$\mathcal{L}^s[\![x.\sigma]\!](1,j) = x^j.\mathcal{L}^s[\![\sigma]\!](0,j)$
$\mathcal{L}^s[\![\mathsf{rec}\ x = \sigma]\!](b,j) = (\mathsf{rec}\ x = \mathcal{L}^s[\![\sigma_i]\!](b,j)) \quad b \in \{0,1\}$
$\mathcal{L}^s[\![0]\!](0,j) = 0^j$

For readability, we feel free to omit the type of the function $\mathcal{L}$ since the following results hold both for the client and the server. We will only use type annotations when expressing a contract in λ^{req}.
The following examples show the steps of a computations of a labeled client contract and a labeled server contract. In particular, we will see that starting from the same contract the functions generate different labeled contracts, because of the behavior in the internal/external choice.

Example 5.1.1
$\sigma = ((a + b) \oplus c) + d.e$

$$\begin{aligned}
\sigma^n_{client} = {} & \mathcal{L}^c[\![((a + b) \oplus c) + d.e]\!](0, 0) = \\
& (\mathcal{L}^c[\![((a + b) \oplus c)]\!](1, 1) + \mathcal{L}^c[\![d.e]\!](1, 1))^0 = \\
& ((\mathcal{L}^c[\![a + b]\!](1, 1) \oplus \mathcal{L}^c[\![c]\!](1, 1))^1 + d^1.\mathcal{L}^c[\![e]\!](0, 1))^0 = \\
& (((\mathcal{L}^c[\![a]\!](1, 2) + \mathcal{L}^c[\![b]\!](1, 2))^1 \oplus c^1.\mathcal{L}^c[\![0]\!](0, 1))^1 + d^1.e^1.\mathcal{L}^c[\![0]\!](0, 2))^0 = \\
& (((a^2.\mathcal{L}^c[\![0]\!](0, 2) + b^2.\mathcal{L}^c[\![0]\!](0, 2))^1 \oplus c^1.0^1)^1 + d^1.e^1.0^2)^0 = \\
& (((a^2.0^2 + b^2.0^2)^1 \oplus c^1.0^1)^1 + d^1.e^1.0^2)^0
\end{aligned}$$

$$\begin{aligned}
\sigma^n_{server} = {} & \mathcal{L}^s[\![((a + b) \oplus c) + d.e]\!](0, 0) = \\
& (\mathcal{L}^s[\![((a + b) \oplus c)]\!](0, 0) + \mathcal{L}^s[\![d.e]\!](0, 0))^0 = \\
& ((\mathcal{L}^s[\![a + b]\!](1, 1) \oplus \mathcal{L}^s[\![c]\!](1, 1))^0 + d^0.\mathcal{L}^s[\![e]\!](0, 1))^0 = \\
& (((\mathcal{L}^s[\![a]\!](1, 1) + \mathcal{L}^s[\![b]\!](1, 1))^1 \oplus c^1.\mathcal{L}^s[\![0]\!](0, 1))^0 + d^0.e^1.\mathcal{L}^s[\![0]\!](0, 2))^0 = \\
& (((a^1.\mathcal{L}^s[\![0]\!](0, 1) + b^1.\mathcal{L}^s[\![0]\!](0, 1))^1 \oplus c^1.0^1)^0 + d^0.e^1.0^2)^0 = \\
& (((a^1.0^1 + b^1.0^1)^1 \oplus c^1.0^1)^0 + d^0.e^1.0^2)^0
\end{aligned}$$

The following examples show the steps of the computation of recursive contracts

Example 5.1.2
$\sigma = \mathsf{rec}\ x = a.x \oplus b.c.x$

$$\begin{aligned}
\sigma^n_{client} = {} & \mathcal{L}^c[\![\mathsf{rec}\ x = a.x \oplus b.c.x]\!](0, 0) = \\
& \mathsf{rec}\ x = \mathcal{L}^c[\![a.x \oplus b.c.x]\!](0, 0) = \\
& \mathsf{rec}\ x = (\mathcal{L}^c[\![a.x]\!](0, 0) \oplus \mathcal{L}^c[\![b.c.x]\!](0, 0))^0 = \\
& \mathsf{rec}\ x = (a^0.\mathcal{L}^c[\![x]\!](0, 1) \oplus b^0.\mathcal{L}^c[\![c.x]\!](0, 1))^0 = \\
& \mathsf{rec}\ x = (a^0.x^1.\mathcal{L}^c[\![0]\!](0, 2) \oplus b^0.c^1.\mathcal{L}^c[\![x]\!](0, 2))^0 = \\
& \mathsf{rec}\ x = (a^0.x^1.0^2 \oplus b^0.c^1.x^2.\mathcal{L}^c[\![0]\!](0, 3))^0 = \\
& \mathsf{rec}\ x = (a^0.x^1.0^2 \oplus b^0.c^1.x^2.0^3)^0 =
\end{aligned}$$

Example 5.1.3
$\sigma = \mathsf{rec}\ x = a.x + b.c.x$

$$\begin{aligned}
\sigma^n_{server} = {} & \mathcal{L}^s[\![\mathsf{rec}\ x = a.x + b.c.x]\!](0,0) = \\
& \mathsf{rec}\ x = \mathcal{L}^s[\![a.x + b.c.x]\!](0,0) = \\
& \mathsf{rec}\ x = (\mathcal{L}^s[\![a.x]\!](0,0) + \mathcal{L}^s[\![b.c.x]\!](0,0))^0 = \\
& \mathsf{rec}\ x = (a^0.\mathcal{L}^s[\![x]\!](0,1) + b^0.\mathcal{L}^s[\![c.x]\!](0,1))^0 = \\
& \mathsf{rec}\ x = (a^0.x^1.\mathcal{L}^s[\![0]\!](0,2) + b^0.c^1.\mathcal{L}^s[\![x]\!](0,2))^0 = \\
& \mathsf{rec}\ x = (a^0.x^1.0^2 \oplus b^0.c^1.x^2.\mathcal{L}^s[\![0]\!](0,3))^0 = \\
& \mathsf{rec}\ x = (a^0.x^1.0^2 \oplus b^0.c^1.x^2.0^3)^0 =
\end{aligned}$$

Our next step is the calculation of the greatest label of a contract. This intuitively represents the maximum number of λ-abstractions of the function. We will use this information for giving the same maximum label to all the branches of a contract, so the transformations in λ^{req} will be well-typed.
Let $\mathcal{D}(\sigma^n)$ be the maximum depth of σ^n where the function $\mathcal{D}$ is defined as:

$\mathcal{D} : \sigma^n \longrightarrow \mathbb{N}$

$\mathcal{D}(\sum_{i\in I} \sigma_i^{n_i n}) = max \bigcup_{i\in I} \mathcal{D}(\sigma_i^{n_i})$
$\mathcal{D}(\bigoplus_{i\in I} \sigma_i^{n_i n}) = max \bigcup_{i\in I} \mathcal{D}(\sigma_i^{n_i})$
$\mathcal{D}(a^n.\sigma^{n'}) = \mathcal{D}(\sigma^{n'})$
$\mathcal{D}(x^n.\sigma^{n'}) = \mathcal{D}(\sigma^{n'})$
$\mathcal{D}(\mathsf{rec}\ x = \sigma^n) = \mathcal{D}(\sigma^n)$
$\mathcal{D}(0^n) = n$

5.2 Balancing contracts

The function that generates a balanced contract adds the unit event $*$ to each leaf until it reach the maximum label given in input. The function take in input the contract and the maximum label calculated by the function $\mathcal{D}$:[4]

[4]Note that here we only use the labels of the leaf, the other labels will be used for the translation of the λ^{req} Server

Definition 8 (Balanced contract)**.** *Given a labeled contract σ^n we define the corresponding balanced contract σ^b as : $\sigma^b = \mathcal{B}[\![\sigma^n]\!]\mathcal{D}(\sigma^n)$ where:*

$$\mathcal{B} : \sigma^n \longrightarrow \mathbb{N} \longrightarrow \sigma$$

$\mathcal{B}[\![\sum_{i\in I} \sigma_i^{n_i n}]\!]p = (\sum_{i\in I} \mathcal{B}[\![\sigma_i^{n_i}]\!]p)$
$\mathcal{B}[\![\bigoplus_{i\in I} \sigma_i^{n_i n}]\!]p = (\bigoplus_{i\in I} \mathcal{B}[\![\sigma_i^{n_i}]\!]p)$
$\mathcal{B}[\![a^n.\sigma^{n'}]\!]p = a.\mathcal{B}[\![\sigma^{n'}]\!]p$
$\mathcal{B}[\![x^n.\sigma^{n'}]\!]p = x.\mathcal{B}[\![\sigma^{n'}]\!]p$
$\mathcal{B}[\![\mathsf{rec}\ x = \sigma^n]\!]p = (\mathsf{rec}\ x = \mathcal{B}[\![\sigma^n]\!]p)$
$\mathcal{B}[\![0^n]\!]p = \begin{cases} *.\mathcal{B}[\![0^{n+1}]\!]p & \text{if } n < p \\ 0 & otherwise \end{cases}$

The following examples show the steps of the computation of balanced contracts. We take the examples above: after have computed the labeled contracts now this contracts will be balanced.

Example 5.2.1
$\sigma^n_{client} = (((a^2.0^2 + b^2.0^2)^1 \oplus c^1.0^1)^1 + d^1.e^1.0^2)^0$

$\mathcal{D}(\sigma^n_{client}) = 2$

$$\begin{aligned}\sigma^b_{client} =& \mathcal{B}[\![(((a^2.0^2 + b^2.0^2)^1 \oplus c^1.0^1)^1 + d^1.e^1.0^2)^0]\!]2 = \\ & \mathcal{B}[\![((a^2.0^2 + b^2.0^2)^1 \oplus c^1.0^1)^1]\!]2 + \mathcal{B}[\![d^1.e^1.0^2]\!]2 = \\ & (\mathcal{B}[\![(a^2.0^2 + b^2.0^2)^1]\!]2 \oplus \mathcal{B}[\![c^1.0^1]\!]2) + d.\mathcal{B}[\![e^1.0^2]\!]2 = \\ & ((\mathcal{B}[\![a^2.0^2]\!]2 + \mathcal{B}[\![b^2.0^2]\!]2) \oplus c.\mathcal{B}[\![0^1]\!]2) + d.e.\mathcal{B}[\![0^2]\!]2 = \\ & ((a.\mathcal{B}[\![0^2]\!]2 + b.\mathcal{B}[\![0^2]\!]2) \oplus c.*.\mathcal{B}[\![0^2]\!]2) + d.e.0 = \\ & ((a.0 + b.0) \oplus c.*.0) + d.e.0\end{aligned}$$

$\sigma^n_{server} = (((a^1.0^1 + b^1.0^1)^1 \oplus c^1.0^1)^0 + d^0.e^1.0^2)^0$

$\mathcal{D}(\sigma^n_{server}) = 2$

$$\begin{aligned}\sigma^b_{server} =& \mathcal{B}[\![(((a^1.0^1 + b^1.0^1)^1 \oplus c^1.0^1)^0 + d^0.e^1.0^2)^0]\!]2 \\ & \mathcal{B}[\![((a^1.0^1 + b^1.0^1)^1 \oplus c^1.0^1)^0]\!]2 + \mathcal{B}[\![d^0.e^1.0^2]\!]2 = \\ & (\mathcal{B}[\![(a^1.0^1 + b^1.0^1)^1]\!]2 \oplus \mathcal{B}[\![c^1.0^1]\!]2) + d.\mathcal{B}[\![e^1.0^2]\!]2 = \\ & ((\mathcal{B}[\![a^1.0^1]\!]2 + \mathcal{B}[\![b^1.0^1]\!]2) \oplus c.\mathcal{B}[\![0^1]\!]2) + d.e.\mathcal{B}[\![0^2]\!]2 = \\ & ((a.\mathcal{B}[\![0^1]\!]2 + b.\mathcal{B}[\![0^1]\!]2) \oplus c.*.\mathcal{B}[\![0^2]\!]2) + d.e.0 = \\ & ((a.*.\mathcal{B}[\![0^2]\!]2 + b.*.\mathcal{B}[\![0^2]\!]2) \oplus c.*.0) + d.e.0 = \\ & ((a.*.0 + b.*.0) \oplus c.*.0) + d.e.0\end{aligned}$$

Example 5.2.2

$\sigma^n_{client} = \mathsf{rec}\ x = (a^0.x^1.0^2 \oplus b^0.c^1.x^2.0^3)^0$

$$\begin{aligned}\sigma^b_{client} = \mathcal{B}[\![\mathsf{rec}\ x = (a^0.x^1.0^2 \oplus b^0.c^1.x^2.0^3)^0]\!]\mathcal{D}[\![\mathsf{rec}\ x = (a^0.x^1.0^2 \oplus b^0.c^1.x^2.0^3)^0]\!] = \\ \mathsf{rec}\ x = \mathcal{B}[\![a^0.x^1.0^2 \oplus b^0.c^1.x^2.0^3)^0]\!]3 = \\ \mathsf{rec}\ x = \mathcal{B}[\![a^0.x^1.0^2]\!]3 \oplus \mathcal{B}[\![b^0.c^1.x^2.0^3]\!]3 = \\ \mathsf{rec}\ x = a.\mathcal{B}[\![x^1.0^2]\!]3 \oplus b.\mathcal{B}[\![c^1.x^2.0^3]\!]3 = \\ \mathsf{rec}\ x = a.x.\mathcal{B}[\![0^2]\!]3 \oplus b.c.\mathcal{B}[\![x^2.0^3]\!]3 = \\ \mathsf{rec}\ x = a.x.*.\mathcal{B}[\![0^3]\!]3 \oplus b.c.x.\mathcal{B}[\![0^3]\!]3 = \\ \mathsf{rec}\ x = a.x.*.0 \oplus b.c.x.0\end{aligned}$$

Example 5.2.3

$\sigma^n_{server} = \mathsf{rec}\ x = (a^0.x^1.0^2 + b^0.c^1.x^2.0^3)^0$

$$\begin{aligned}\sigma^b_{server} = \mathcal{B}[\![\mathsf{rec}\ x = (a^0.x^1.0^2 + b^0.c^1.x^2.0^3)^0]\!]\mathcal{D}[\![\mathsf{rec}\ x = (a^0.x^1.0^2 + b^0.c^1.x^2.0^3)^0]\!] = \\ \mathsf{rec}\ x = \mathcal{B}[\![a^0.x^1.0^2 + b^0.c^1.x^2.0^3)^0]\!]3 = \\ \mathsf{rec}\ x = \mathcal{B}[\![a^0.x^1.0^2]\!]3 + \mathcal{B}[\![b^0.c^1.x^2.0^3]\!]3 = \\ \mathsf{rec}\ x = a.\mathcal{B}[\![x^1.0^2]\!]3 + b.\mathcal{B}[\![c^1.x^2.0^3]\!]3 = \\ \mathsf{rec}\ x = a.x.\mathcal{B}[\![0^2]\!]3 + b.c.\mathcal{B}[\![x^2.0^3]\!]3 = \\ \mathsf{rec}\ x = a.x.*.\mathcal{B}[\![0^3]\!]3 + b.c.x.\mathcal{B}[\![0^3]\!]3 = \\ \mathsf{rec}\ x = a.x.*.0 + b.c.x.0\end{aligned}$$

Property 1. $\mathcal{D}(\sigma^b) = \mathcal{D}(\sigma^n)$

Proof. We have that $\sigma^b = \mathcal{B}[\![\sigma^n]\!]\mathcal{D}(\sigma^n)$. By applying the rule $\mathcal{B}[\![0^n]\!]p$ we have $p = \mathcal{D}(\sigma^n)$, also we cannot have $n > p$ since by definition of $\mathcal{D}$ p is the maximum depth of the contract σ, also if $n < p$ we apply the rule $\mathcal{B}[\![(0^b)^n]\!]p = *.\mathcal{B}[\![(0^b)^{n+1}]\!]p$ and we stop only when $n = p$, leading to $\mathcal{D}(\sigma^b) = \mathcal{D}(\sigma^n)$. □

Property 2. $(\sigma^b)^b = \sigma^b$

Proof. Let $(\sigma^b)^n = \mathcal{L}[\![\sigma^b\ ]\!](0,0)$ and $(\sigma^b)^b = \mathcal{B}[\![(\sigma^b)^n]\!]\mathcal{D}((\sigma^b)^n)$, by contradiction assume $\mathcal{B}[\![(\sigma^b)^n]\!]\mathcal{D}((\sigma^b)^n) \neq \sigma^b$. We applied at least one time the rule $\mathcal{B}[\![(0^b)^n]\!]p = *.\mathcal{B}[\![(0^b)^{n+1}]\!]p$ with $n < p$, but we have $n = \mathcal{D}(\sigma^n)$ (definition of σ^b) and $p = \mathcal{D}(\mathcal{L}[\![\mathcal{B}[\![\sigma^n]\!]\mathcal{D}(\sigma^n)]\!](0,0))) = D(\sigma^n)$ (definition of σ^b and σ^n) so we obtain the contradiction $\mathcal{D}(\sigma^n) < \mathcal{D}(\sigma^n)$ □

Property 3. $\sigma^b = \sum_{i\in I}\sigma^b_i \vee \sigma^b = \bigoplus_{i\in I}\sigma^b_i \ \rightarrow \forall i.\mathcal{D}((\sigma^b_i)^{n_i}) = \mathcal{D}(\sigma^b)$

Proof. By definition of $\mathcal{B}[\![\sum_{i\in I}\sigma_i^{n_i n}]\!]p = (\sum_{i\in I}\mathcal{B}[\![\sigma_i^{n_i}]\!]p)$ and $\mathcal{B}[\![\bigoplus_{i\in I}\sigma_i^{n_i n}]\!]p = (\bigoplus_{i\in I}\mathcal{B}[\![\sigma_i^{n_i}]\!]p)$ and $p = \mathcal{D}(\sigma^b)$. Then it follows that $\mathcal{D}(\mathcal{B}[\![\sigma_i^{n_i}]\!]\mathcal{D}(\sigma^b)) = \mathcal{D}(\sigma^b)$ □

The last properties show that in a balanced contract every leaf node occurs at the same depth. This will guarantee that the translation in λ^{req} will not lead to any type error.

6 From σ^b to λ^{req}

In this section, we transform balanced contracts into a λ^{req} expressions and policies. First we introduce some shorthands [5] [6] :

1. $e1; e2 = (\lambda.e2)(e1)$
2. $e1; \lambda.e2 = \lambda.(e1; e2)$
3. $e1; x.e2 = (\lambda x.e2)(e1)$
4. $e1; \lambda x.e2 = (\lambda.\lambda x.e2)(e1)$
5. $\mu z.(\lambda y.e) = \lambda_z y.e$
6. $\mu z.e = e\{^{\mu z.e}/_z\}$

6.1 On design the client specifications for a contract

We inductively define a function $\mathcal{C}$ that constructs the λ^{req} client starting from a balanced client contract. The function takes in input a boolean the use of which has the same utility of the one in the labeling function: we shall avoid to add the λ-abstraction to the first action performed after the external choice. The rules of the internal and external choice are equals as in the previous sections[7].

Definition 9. *The function $\mathcal{C}$ that generates the λ^{req} expression of client starting from a balanced client contract is defined as:*

$\mathcal{C} : \sigma \longrightarrow \{0,1\} \longrightarrow e$

$$\begin{aligned}\mathcal{C}[\![\textstyle\sum_{i\in I}\sigma_i]\!]b = \lambda cx.\ &\textit{if } (cx{=}\sigma_1) \textit{ then}\\ &\quad \mathcal{C}[\![\sigma_1]\!]1\\ &\textit{else if } (cx{=}\sigma_2) \textit{ then}\\ &\quad \mathcal{C}[\![\sigma_2]\!]1\\ &\cdots\\ &\textit{else}\\ &\quad \mathcal{C}[\![\sigma_n]\!]1 \qquad b \in \{0,1\}\end{aligned}$$

[5] The last case is not a lambda abstraction, since we have only guarded contract we rule out expression like $\mu h.h + h$

[6] Note that $3 \neq 4$

[7] We avoid to specify the type of the contract if it is clear from the context

$$\begin{aligned}
\mathcal{C}[\![\textstyle\bigoplus_{i\in I}\sigma_i]\!]b = \ & \textit{if } (\ldots) \textit{ then} \\
& \quad \mathcal{C}[\![\sigma_1]\!]b \\
& \textit{else if } (\ldots) \textit{ then} \\
& \quad \mathcal{C}[\![\sigma_2]\!]b \\
& \ldots \\
& \textit{else} \\
& \quad \mathcal{C}[\![\sigma_n]\!]b \quad b \in \{0,1\}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{C}[\![a.\sigma]\!]0 = (\lambda.a; \mathcal{C}[\![\sigma]\!]0) \\
&\mathcal{C}[\![a.\sigma]\!]1 = (a; \mathcal{C}[\![\sigma]\!]0) \\
&\mathcal{C}[\![x.\sigma]\!]0 = (\lambda.x; \mathcal{C}[\![\sigma]\!]0) \\
&\mathcal{C}[\![x.\sigma]\!]1 = (x; \mathcal{C}[\![\sigma]\!]0) \\
&\mathcal{C}[\![\mathsf{rec}\ x = \sigma]\!]b = \mu x.\mathcal{C}[\![\sigma]\!]b \ \ b \in \{0,1\} \\
&\mathcal{C}[\![0]\!]b = * \quad b \in \{0,1\}
\end{aligned}$$

We proceed now to compute the type of the lambda expression returned by $\mathcal{C}$. This inference rules are alternative to the Type and Effect system, with more specific types additionally we don't have weakening rule.

$$\begin{aligned}
&\mathcal{C}[\![0]\!]b : 1 \ \ b \in \{0,1\} \\
&\mathcal{C}[\![a.\sigma]\!]0 : 1 \longmapsto \tau_{\mathcal{C}[\![\sigma]\!]0} \\
&\mathcal{C}[\![a.\sigma]\!]1 : \tau_{\mathcal{C}[\![\sigma]\!]0} \\
&\mathcal{C}[\![x.\sigma]\!]0 : 1 \longmapsto \tau_{\mathcal{C}[\![\sigma]\!]0} \\
&\mathcal{C}[\![x.\sigma]\!]1 : \tau_{\mathcal{C}[\![\sigma]\!]0} \\
&\mathcal{C}[\![\mathsf{rec}\ x = \sigma]\!]b : \tau_{\mathcal{C}[\![\sigma]\!]b} \ \ b \in \{0,1\}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{C}[\![\textstyle\sum_{i\in I}\sigma_i]\!]b : 1 \longmapsto \tau_{\mathcal{C}[\![\sigma_1]\!]0} \quad \text{if } \tau_{\mathcal{C}[\![\sigma_i]\!]0} = \tau_{\mathcal{C}[\![\sigma_j]\!]0} \forall i,j \in I \quad b \in \{0,1\} \\
&\mathcal{C}[\![\textstyle\bigoplus_{i\in I}\sigma_i]\!]b : \tau_{\mathcal{C}[\![\sigma_1]\!]b} \quad \text{if } \tau_{\mathcal{C}[\![\sigma_i]\!]0} = \tau_{\mathcal{C}[\![\sigma_j]\!]0} \forall i,j \in I \quad b \in \{0,1\}
\end{aligned}$$

For well-typing the lambda-expression in the internal and external choice all their sub-terms must have the same type, in a balanced contract all the sub-terms have the same depth, it suffice to prove that two such contracts have the same type.

Lemma 2. $\mathcal{D}(\sigma_1^b) = \mathcal{D}(\sigma_2^b) \longrightarrow \tau_{\mathcal{C}[\![\sigma_1^b]\!]0} = \tau_{\mathcal{C}[\![\sigma_2^b]\!]0}$

Proof. By definition of $\mathcal{L}^c$ and $\mathcal{C}$ we observe that the labels of the contract σ^n only increase when we add a λ-abstraction in the λ-expression of σ^b:

$\mathcal{C}[\![0]\!]b : 1 \;\; b \in \{0,1\}$	$\mathcal{L}^c[\![0]\!](0,j) = 0^j$
$\mathcal{C}[\![a.\sigma]\!]0 : 1 \longmapsto \tau_{\mathcal{C}[\![\sigma]\!]0}$	$\mathcal{L}^c[\![a.\sigma]\!](0,j) = a^j.\mathcal{L}^c[\![\sigma]\!](0,j+1)$
$\mathcal{C}[\![a.\sigma]\!]1 : \tau_{\mathcal{C}[\![\sigma]\!]0}$	$\mathcal{L}^c[\![a.\sigma]\!](1,j) = a^j.\mathcal{L}^c[\![\sigma]\!](0,j)$
$\mathcal{C}[\![x.\sigma]\!]0 : 1 \longmapsto \tau_{\mathcal{C}[\![\sigma]\!]0}$	$\mathcal{L}^c[\![x.\sigma]\!](0,j) = x^j.\mathcal{L}^c[\![\sigma]\!](0,j+1)$
$\mathcal{C}[\![x.\sigma]\!]1 : \tau_{\mathcal{C}[\![\sigma]\!]0}$	$\mathcal{L}^c[\![x.\sigma]\!](1,j) = x^j.\mathcal{L}^c[\![\sigma]\!](0,j)$
$\mathcal{C}[\![\mathsf{rec}\ x = \sigma]\!]b : \tau_{\mathcal{C}[\![\sigma]\!]b} \;\; b \in \{0,1\}$	$\mathcal{L}^c[\![\mathsf{rec}\ x = \sigma]\!](b,j) = (\mathsf{rec}\ x = \mathcal{L}^c[\![\sigma_i]\!](b,j))$
$\mathcal{C}[\![\sum_{i\in I}\sigma_i]\!]b : 1 \longmapsto \tau_{\mathcal{C}[\![\sigma_1]\!]0}$	$\mathcal{L}^c[\![\sum_{i\in I}\sigma_i]\!](b,j) = (\sum_{i\in I}\mathcal{L}^c[\![\sigma_i]\!](1,j+1))^j$
$\mathcal{C}[\![\bigoplus_{i\in I}\sigma_i]\!]b : \tau_{\mathcal{C}[\![\sigma_1]\!]b}$	$\mathcal{L}^c[\![\bigoplus_{i\in I}\sigma_i]\!](b,j) = (\bigoplus_{i\in I}\mathcal{L}^c[\![\sigma_i]\!](b,j))^j$

The number of λ-abstractions equal the greatest label calculated by $\mathcal{D}$ (recall that we have balanced contract therefore all the leaf node have the same label and the same number of λ-abstractions); by applying the typing rule below we obtain:

$$\tau_{\mathcal{C}[\![\sigma_1^b]\!]0} = \overbrace{1 \longrightarrow (1 \longrightarrow \cdots (1}^{\mathcal{D}(\sigma_1^b)} \longrightarrow 1)) \cdots)$$

$$\tau_{\mathcal{C}[\![\sigma_2^b]\!]0} = \overbrace{1 \longrightarrow (1 \longrightarrow \cdots (1}^{\mathcal{D}(\sigma_2^b)} \longrightarrow 1)) \cdots)$$

and by hypothesis $\mathcal{D}(\sigma_1^b) = \mathcal{D}(\sigma_2^b)$ □

Definition 10. *The specification in λ^{req} of the Client contract σ^b_{client} is defined as:*
$req_r(\tau_{\mathcal{C}[\![\sigma^b_{client}]\!]0} \overset{\varphi_c}{\longmapsto} \tau_{val})(\mathcal{C}[\![\sigma^b_{client}]\!]0)$ *where:*

$$\tau_{\mathcal{C}[\![\sigma^b_{client}]\!]0} = \overbrace{1 \longrightarrow (1 \longrightarrow \cdots (1}^{\mathcal{D}(\sigma^b_{client})} \longrightarrow 1)) \cdots)$$

Assume $\tau_{val} = 1$. The Server returns the unit value * because the code of the Client migrates to the Server and all the interactions between the two party occur by the Server Side. There is no need for the Client to get some value. In the external choice we abstract from the if condition and we write $cx = \sigma_i$ for check what sub-term σ_i the Server has choose. As discussed in the previous section φ_c is equal to σ_{client} where the outputs action are replaced by the corresponding dual actions.
We conclude by observing that in λ^{req} a service can make more request in cascade, leading to a multi-contract client.
In the following example we show the steps of the computation of a λ^{req} client. The contract in input is the one of the previous example. Note that we do not have type errors since the contract is balanced.

Example 6.1.1
$\sigma^b_{client} = ((a.0 + b.0) \oplus c. * .0) + d.e.0$

$\varphi_c = ((a + b) \oplus c) + d.e$

$$\mathcal{C}[\![((a.0e + b.0) \oplus c. * .0) + d.e.0]\!]0 = \lambda \text{cx.if } (\text{cx} = \cdots) \text{then } \mathcal{C}[\![(a.0 + b.0) \oplus c. * .0]\!]1 \text{ else } \mathcal{C}[\![d.e.0]\!]1$$

$$\mathcal{C}[\![(a.0 + b.0) \oplus c. * .0]\!]1 = \text{if } (\ \cdots\ ) \text{ then } \mathcal{C}[\![a.0 + b.0]\!]1 \text{ else } \mathcal{C}[\![c. * .0]\!]1$$

$$\mathcal{C}[\![d.e.0]\!]1 = d; \mathcal{C}[\![e.0]\!]0 = d; \lambda.e; \mathcal{C}[\![0]\!]0 = d; \lambda.e; *$$

$$\mathcal{C}[\![a.0 + b.0]\!]1 = \lambda \text{cx. if } (\ \text{cx} = \cdots\ ) \text{ then } \mathcal{C}[\![a.0]\!]1 \text{ else } \mathcal{C}[\![b.0]\!]1$$

$$\mathcal{C}[\![c. * .0]\!]1 = c; \mathcal{C}[\![*.0]\!]0 = c; \lambda.*; \mathcal{C}[\![0]\!]0 = c; \lambda.*; *$$

$$\mathcal{C}[\![a.0]\!]1 = a; \mathcal{C}[\![0]\!]0 = a; *;$$

$$\tau_{\mathcal{C}[\![\sigma^b_{client}]\!]0} = 1 \to (1 \to 1)$$

The λ^{req} client of $\sigma^b_{client} = ((a.0 + b.0) \oplus c. * .0) + d.e.0$ is:

$$req_r((1 \to (1 \to 1)) \xrightarrow{\varphi_c} 1)\ \lambda \text{cx.if } (\text{cx} = \cdots) \text{then if } (\ \cdots\ ) \text{ then } \lambda \text{cx. if } (\ \text{cx} = \cdots\ ) \text{ then } a; * \text{ else } b; * \text{ else } c; \lambda.*; * \text{ else } d; \lambda.e; *$$

In the following example we show the steps of the computation of a recursive contract. The contract in input is the one of the previous example.

Example 6.1.2
$\sigma^b_{client} = \mathsf{rec}\ x = a.x. * .0 \oplus b.c.x.0$

$\varphi_c = \mathsf{rec}\ x = a.x \oplus b.c.x$

$\mathcal{C}[\![\mathsf{rec}\ x = a.x. * .0 \oplus b.c.x.0]\!]0 =$
$\mu x.\mathcal{C}[\![a.x. * .0 \oplus b.c.x.0]\!]0 =$
$\mu x.$if $(\cdots)$ then
$\mathcal{C}[\![a.x. * .0]\!]0$
else
$\mathcal{C}[\![b.c.x.0]\!]0 =$
$\mu x.$if $(\cdots)$ then
$\lambda.a; \lambda.x; \lambda.*; *$
else
$\lambda.b; \lambda.c; \lambda.x; *$

$\tau_{\sigma^b_{client}} = 1 \to (1 \to (1 \to 1))$

The λ^{req} client is:

$req_r(1 \to (1 \to (1 \to 1)) \xrightarrow{\varphi_c} 1)\ \mu x.$if $(\cdots)$ then
$\lambda.a; \lambda.x; \lambda.*; *$
else
$\lambda.b; \lambda.c; \lambda.x; *$

6.2 On design the Server specifications for a contract

Having defined the Client now we introduce the function $\mathcal{S}$ that constructs the λ^{req} Server starting from a labeled Server contract. Note that, while in the Client we have a unique φ_c block, in the Server we need to break the φ_s in different blocks, as many as the depth of σ_{server}. We take in input a boolean with the same role of the previous functions. The rule for the internal and external choice equals the ones described in the previous sections. The λ^{req} server first applies the function client and then produces the corresponding co-actions. Note that the history expression of the Server will be inside the

scope of φ_{client}, hence the server will perform the actions requests by φ_{client}:

Definition 11. *The function $\mathcal{S}$ that generates the λ^{req} expression of Server starting from a labeled server contract is defined as:*

$\mathcal{S} : \sigma^n \longrightarrow \{0,1\} \longrightarrow e$

$$\begin{aligned}
\mathcal{S}[\![\textstyle\sum_{i\in I} \sigma_i^{n_i n}]\!]b = \ & \textit{if } (f_n = \sigma_1) \textit{ then} \\
& \quad \mathcal{S}[\![\sigma_1^{n_1}]\!]b \\
& \textit{else if } (f_n = \sigma_2) \textit{ then} \\
& \quad \mathcal{S}[\![\sigma_2^{n_2}]\!]b \\
& \cdots \\
& \textit{else} \\
& \quad \mathcal{S}[\![\sigma_n^{n_n}]\!]b \ \ b \in \{0,1\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\![\textstyle\bigoplus_{i\in I} \sigma_i^{n_i n}]\!]b = \ & \textit{if } (\ldots) \textit{ then} \\
& \quad \varphi_s^n[f_n(*_{\sigma_1})]; f_{n_1}.(\mathcal{S}[\![\sigma_1^{n_1}]\!]1) \\
& \textit{else if } (\ldots) \textit{ then} \\
& \quad \varphi_s^n[f_n(*_{\sigma_2})]; f_{n_2}.(\mathcal{S}[\![\sigma_2^{n_2}]\!]1) \\
& \cdots \\
& \textit{else} \\
& \quad \varphi_s^n[f_n(*_{\sigma_n})]; f_{n_n}.(\mathcal{S}[\![\sigma_n^{n_n}]\!]1) \ \ b \in \{0,1\}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{S}[\![a^n.\sigma^{n'}]\!]0 = \varphi_s^n[f_n(*_a)]; f_{n'}.(a; \mathcal{S}[\![\sigma^{n'}]\!]0) \\
&\mathcal{S}[\![a^n.\sigma^{n'}]\!]1 = a; \mathcal{S}[\![\sigma^{n'}]\!]0 \\
&\mathcal{S}[\![x^n.\sigma^{n'}]\!]0 = \varphi_s^n[f_n(*_x)]; f_{n'}.(x; \mathcal{S}[\![\sigma^{n'}]\!]0) \\
&\mathcal{S}[\![x^n.\sigma^{n'}]\!]1 = x; \mathcal{S}[\![\sigma^{n'}]\!]0 \\
&\mathcal{S}[\![\mathsf{rec}\ x = \sigma^n]\!]b = \mu x.\mathcal{S}[\![\sigma^n]\!]b \ \ b \in \{0,1\} \\
&\mathcal{S}[\![0^n]\!]b = * \ \ b \in \{0,1\}
\end{aligned}$$

In the external choice we abstract from the if condition and we write $f_n = \sigma_i$ for check what sub-term σ_i the client have choose, note that f_n is the code of the Client.
We denote by $*_\sigma$ a σ event passed to f that don't generate any history locally to the Server. Note that all the branches of the if have the same type $\tau_{val} = 1$.

Definition 12. *The specification in λ^{req} of a Server contract σ^n_{server} is defined as:*
$\lambda f_0.\mathcal{S}[\![\sigma^n_{server}]\!]0$ *where:*

$$\tau_{f_0} = \overbrace{1 \longrightarrow (1 \longrightarrow \cdots (1}^{\mathcal{D}(\sigma^n_{server})} \longrightarrow 1)) \cdots)$$

$$\tau_{server} = \tau_{f_0} \longrightarrow \tau_{val}$$

For computing the type of the argument f_0 we proceed as done before with the client. Note that in the rule of the function $\mathcal{L}^s$ we increase the label of the node when we apply the function f_i in the rule of $\mathcal{S}$. The number of the applications is then equals to $\mathcal{D}(\sigma^n_{server})$.
Note that the service hold the request of the client only if $\tau_{f_0} = \tau_{client}$, but if $\mathcal{D}(\sigma_{server}) > \mathcal{D}(\sigma_{client})$ the interaction does not occur, and the two parties can be compliant since the client can decide to terminate instead of synchronizing with the co-action of the server. To avoid this problem when the orchestrator plans the execution of the services, it can balance the client contract with the depth of the server contract to equalize the type of the two expression.
In the following example we show the steps of the computation of a λ^{req} server. The contract in input is the one of the previous example.

Example 6.2.1
$\sigma^n_{server} = (((a^1.0^1 + b^1.0^1)^1 \oplus c^1.0^1)^0 + d^0.e^1.0^2)^0$

$$\mathcal{S}[\![(((a^1.0^1 + b^1.0^1)^1 \oplus c^1.0^1)^0 + d^0.e^1.0^2)^0]\!]0 = \text{if } (f_0 = \cdots) \text{ then } \mathcal{S}[\![((a^1.0^1 + b^1.0^1)^1 \oplus c^1.0^1)^0]\!]0 \text{ else } \mathcal{S}[\![d^0.e^1.0^2]\!]0$$

$$\mathcal{S}[\![((a^1.0^1 + b^1.0^1)^1 \oplus c^1.0^1)^0]\!]0 = \text{if } (...) \text{ then } \varphi^0_s[f_0(*_{a+b})]; f_1.(\mathcal{S}[\![(a^1.0^1 + b^1.0^1)^1]\!]1) \text{ else } \varphi^0_s[f_0(*_c)]; f_1.(\mathcal{S}[\![c^1.0^1]\!]1)$$

$$\begin{aligned}\mathcal{S}[\![d^0.e^1.0^2]\!]0 &= \varphi^0_s[f_0(*_d)]; f_1.(d; \mathcal{S}[\![e^1.0^2]\!]0) \\ &= \varphi^0_s[f_0(*_d)]; f_1.(d; \varphi^1_s[f_1(*_e)]; f_2.(e; \mathcal{S}[\![0^2]\!]0))\end{aligned}$$

$$
\begin{array}{l}
\quad = \varphi_s^0[f_0(*_d)]; \\
\quad\quad f_1.(d; \varphi_s^1[f_1(*_e)]; \\
\quad\quad f_2.(e; *))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}[\![(a^1.0^1 + b^1.0^1)^1]\!]1 = \text{if } (f_1 = \cdots) then \\
\qquad\qquad\qquad\qquad\qquad \mathcal{S}[\![a^1.0^1]\!]1 \\
\qquad\qquad\qquad\qquad \text{else} \\
\qquad\qquad\qquad\qquad\quad \mathcal{S}[\![b^1.0^1]\!]1
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}[\![c^1.0^1]\!]1 = c; * \\
\mathcal{S}[\![a^1.0^1]\!]1 = a; * \\
\mathcal{S}[\![b^1.0^1]\!]1 = b; *
\end{array}
$$

The λ^{req} Server is:

$$
\begin{array}{l}
\lambda f_0.\text{if } (\ f_0 = \cdots) \text{ then} \\
\qquad \text{if } (...) \text{ then} \\
\qquad\quad \varphi_s^0[f_0(*_{a+b})]; \\
\qquad\quad f_1.(\ \text{if } (\ f_1 = \cdots) \text{ then} \\
\qquad\qquad\qquad\quad a; * \\
\qquad\qquad\quad \text{else} \\
\qquad\qquad\qquad\quad b; *) \\
\\
\qquad \text{else} \\
\qquad\quad \varphi_s^0[f_0(*_c)]; f_1.(c; *) \\
\quad \text{else} \\
\qquad \varphi_s^0[f_0(*_d)]; \\
\qquad f_1.(d; \varphi_s^1[f_1(*_e)]; \\
\qquad f_2.(e; *))
\end{array}
$$

$\tau_{server} = (1 \to (1 \to 1)) \longrightarrow 1$

In the following example we show the steps of a computation of a λ^{req} recursive server. The contract is the one of the previous example.

Example 6.2.2
$\sigma^n_{server} = \mathsf{rec}\ x = (a^0.x^1.0^2 + b^0.c^1.x^2.0^3)^0$

$\mathcal{S}[\![\mathsf{rec}\ x = (a^0.x^1.0^2 + b^0.c^1.x^2.0^3)^0]\!]0 =$

$\mu x.\mathcal{S}[\![(a^0.x^1.0^2 + b^0.c^1.x^2.0^3)^0]\!]0 =$

$\mu x.$ if($f_0 = \cdots$) then
$\mathcal{S}[\![a^0.x^1.0^2]\!]0$
else
$\mathcal{S}[\![b^0.c^1.x^2.0^3]\!]0 =$

The λ^{req} Server is:

$\lambda f_0.\mu x.$ if($f_0 = \cdots$) then
$\varphi^0_s[f_0 *_a]; f_1.(a;$
$\varphi^1_s[f_1 *_x]; f_2.(x, *))$
else
$\varphi^0_s[f_0 *_b]; f_1.(b;$
$\varphi^1_s[f_1 *_c]; f_2.(c;$
$\varphi^2_s[f_2 *_x]; f_3.(x; *)))$

Now we define a formal method to compute φ_{server}. We simply split the policy into more policies, as many as the depth of the contract. Every policy represents a level of the tree, and it will contain in his frame the application of the function client at that level. The function Φ takes in input the contract, a boolean with the same role of the previous functions, and the index of the policy we want to generate. Note that for the recursion we need to make a step of unfolding for avoiding meaningless policies like $\varphi^i_s = x$. The contract in input is first balanced and then labeled, the label is used for check the level of depth in the tree.

Definition 13. *We define the policy φ_s of the server as the set $\varphi_s = \{\varphi^0_s, \cdots, \varphi^{|I|}_s\}$ where:*
$\varphi^i_s = \Phi[\![\mathcal{L}^s[\![\sigma^b_{server}]\!](0,0)]\!](0,i)\ \ i \in I = \{0, \cdots, \mathcal{D}(\sigma^b_{server}) - 1\}$ *where:*

$\Phi : \sigma^n \longrightarrow \{0,1\} \times \mathbb{N} \longrightarrow \sigma$

$\Phi[\![\sum_{i\in I} \sigma_i^{n_i n}]\!](b,z) = \sum_{i\in I} \Phi[\![\sigma_i^{n_i}]\!](b,z)\ \ b \in \{0,1\}$

$$\Phi[\![\bigoplus_{i\in I} \sigma_i^{n_i n}]\!](b,z) = \begin{cases} \bigoplus_{i\in I} \Phi[\![\sigma_i^{n_i}]\!](1,z) & \text{if } z \geq n \\ \varepsilon & otherwise \end{cases} \quad b \in \{0,1\}$$

$$\Phi[\![a^n.\sigma^{n'}]\!](0,z) = \begin{cases} a & \text{if } z = n \\ \Phi[\![\sigma^{n'}]\!](0,z) & \text{if } z > n \\ \varepsilon & otherwise \end{cases}$$

$$\Phi[\![a^n.\sigma^{n'}]\!](1,z) = \begin{cases} a & \text{if } z = n-1 \\ \Phi[\![\sigma^{n'}]\!](0,z) & \text{if } z \geq n \\ \varepsilon & otherwise \end{cases}$$

$$\Phi[\![x^n.\sigma^{n'}]\!](0,z) = \begin{cases} x & \text{if } z = n \\ \Phi[\![\sigma^{n'}]\!](0,z) & \text{if } z > n \\ \varepsilon & otherwise \end{cases}$$

$$\Phi[\![x^n.\sigma^{n'}]\!](1,z) = \begin{cases} x & \text{if } z = n-1 \\ \Phi[\![\sigma^{n'}]\!](0,z) & \text{if } z \geq n \\ \varepsilon & otherwise \end{cases}$$

$$\Phi[\![\mathsf{rec}\ x = \sigma^n]\!](b,z) = \Phi[\![\sigma\{^{\mathsf{rec}\ x=\sigma^n}/_x\}]\!](b,z) \quad b \in \{0,1\}$$

$$\Phi[\![(\mathsf{rec}\ x = \sigma^n)^m]\!](b,z) = \begin{cases} \mathsf{rec}\ x = \sigma & \text{if } z = m \\ \varepsilon & otherwise \end{cases} \quad b \in \{0,1\}$$

$$\Phi[\![0^n]\!](b,z) = \varepsilon \quad b \in \{0,1\}$$

The following examples compute the φ_{server} of the previous example.

Example 6.2.3
$\sigma^b_{server} = ((a.*.0 + b.*.0) \oplus c.*.0) + d.e.0$

$$\begin{aligned}(\sigma^b_{server})^n_{server} =& \mathcal{L}[\![((a.*.0 + b.*.0) \oplus c.*.0) + d.e.0]\!](0,0) = \\ & (((a^1.*^1.0^2 + b^1.*^1.0^2)^1 \oplus c^1.*^1.0^2)^0 + d^0.e^1.0^2)^0\end{aligned}$$

$I = \{0,1\}$

$$\begin{aligned}\varphi^0_s =& \Phi[\![(((a^1.*^1.0^2 + b^1.*^1.0^2)^1 \oplus c^1.*^1.0^2)^0 + d^0.e^1.0^2)^0]\!](0,0) = \\ & \Phi[\![(((a^1.*^1.0^2 + b^1.*^1.0^2)^1 \oplus c^1.*^1.0^2)^0]\!](0,0) + \Phi[\![d^0.e^1.0^2]\!](0,0) = \\ & (\Phi[\![((a^1.*^1.0^2 + b^1.*^1.0^2)^1]\!](1,0) \oplus \Phi[\![c^1.*^1.0^2]\!](1,0)) + d = \\ & ((\Phi[\![a^1.*^1.0^2]\!](1,0) + \Phi[\![b^1.*^1.0^2]\!](1,0)) \oplus c) + d = \\ & ((a+b) \oplus c) + d\end{aligned}$$

$$\begin{aligned}\varphi^1_s =& \Phi[\![(((a^1.*^1.0^2 + b^1.*^1.0^2)^1 \oplus c^1.*^1.0^2)^0 + d^0.e^1.0^2)^0]\!](0,1) = \\ & \Phi[\![(((a^1.*^1.0^2 + b^1.*^1.0^2)^1 \oplus c^1.*^1.0^2)^0]\!](0,1) + \Phi[\![d^0.e^1.0^2]\!](0,1) = \\ & (\Phi[\![((a^1.*^1.0^2 + b^1.*^1.0^2)^1]\!](1,1) \oplus \Phi[\![c^1.*^1.0^2]\!](1,1)) + \Phi[\![e^1.0^2]\!](0,1) = \\ & ((\Phi[\![a^1.*^1.0^2]\!](1,1) + \Phi[\![b^1.*^1.0^2]\!](1,1)) \oplus \Phi[\![*^1.0^2]\!](0,1)) + e = \\ & ((\Phi[\![*^1.0^2]\!](0,1) + \Phi[\![*^1.0^2]\!](0,1)) \oplus \varepsilon) + e =\end{aligned}$$

$$((\varepsilon + \varepsilon) \oplus \varepsilon) + e$$

Example 6.2.4
$\sigma^b_{server} = \mathsf{rec}\ x = a.x. * .0 + b.c.x.0$

$$(\sigma^b_{server})^n_{server} = \mathcal{L}[\![\mathsf{rec}\ x = a.x. * .0 + b.c.x.0]\!](0,0) = \mathsf{rec}\ x = (a^0.x^1. *^2 .0^3 + b^0.c^1.x^2.0^3)^0$$

$I = \{0, 1, 2\}$

$$\begin{aligned}
\varphi^0_s = {} & \Phi[\![\mathsf{rec}\ x = (a^0.x^1. *^2 .0^3 + b^0.c^1.x^2.0^3)^0]\!](0,0) = \\
& \Phi[\![(a^0.(\mathsf{rec}\ x = a.x. * + b.c.x)^1. *^2 .0^3 + \\
& \quad b^0.c^1.(\mathsf{rec}\ x = a.x. * + b.c.x)^2.0^3)^0]\!](0,0) = \\
& \Phi[\![a^0.(\mathsf{rec}\ x = a.x. * + b.c.x)^1. *^2 .0^3]\!](0,0) + \\
& \quad \Phi[\![b^0.c^1.(\mathsf{rec}\ x = a.x. * + b.c.x)^2.0^3]\!](0,0) = \\
& \mathrm{a} + \mathrm{b}
\end{aligned}$$

$$\begin{aligned}
\varphi^1_s = {} & \Phi[\![\mathsf{rec}\ x = (a^0.x^1. *^2 .0^3 + b^0.c^1.x^2.0^3)^0]\!](0,1) = \\
& \Phi[\![(a^0.(\mathsf{rec}\ x = a.x. * + b.c.x)^1. *^2 .0^3 + \\
& \quad b^0.c^1.(\mathsf{rec}\ x = a.x. * + b.c.x)^2.0^3)^0]\!](0,1) = \\
& \Phi[\![a^0.(\mathsf{rec}\ x = a.x. * + b.c.x)^1. *^2 .0^3]\!](0,1) + \\
& \quad \Phi[\![b^0.c^1.(\mathsf{rec}\ x = a.x. * + b.c.x)^2.0^3]\!](0,1) = \\
& \Phi[\![(\mathsf{rec}\ x = a.x. * + b.c.x)^1. *^2 .0^3]\!](0,1) + \\
& \quad \Phi[\![c^1.(\mathsf{rec}\ x = a.x. * + b.c.x)^2.0^3]\!](0,1) = \\
& (\mathsf{rec}\ x = a.x.\varepsilon + b.c.x) + c
\end{aligned}$$

$$\begin{aligned}
\varphi^2_s = {} & \Phi[\![\mathsf{rec}\ x = (a^0.x^1. *^2 .0^3 + b^0.c^1.x^2.0^3)^0]\!](0,2) = \\
& \Phi[\![(a^0.(\mathsf{rec}\ x = a.x. * + b.c.x)^1. *^2 .0^3 + \\
& \quad b^0.c^1.(\mathsf{rec}\ x = a.x. * + b.c.x)^2.0^3)^0]\!](0,2) = \\
& \Phi[\![a^0.(\mathsf{rec}\ x = a.x. * + b.c.x)^1. *^2 .0^3]\!](0,2) + \\
& \quad \Phi[\![b^0.c^1.(\mathsf{rec}\ x = a.x. * + b.c.x)^2.0^3]\!](0,2) = \\
& \Phi[\![(\mathsf{rec}\ x = a.x. * + b.c.x)^1. *^2 .0^3]\!](0,2) + \\
& \quad \Phi[\![c^1.(\mathsf{rec}\ x = a.x. * + b.c.x)^2.0^3]\!](0,2) = \\
& \Phi[\![*^2.0^3]\!](0,2) + \Phi[\![(\mathsf{rec}\ x = a.x. * + b.c.x)^2.0^3]\!](0,2) = \\
& \varepsilon + (\mathsf{rec}\ x = a.x.\varepsilon + b.c.x)
\end{aligned}$$

7 Validation

In this section we define a formal method for checking whether an history H complies with a contract φ_σ. As discussed above we can't use automata since our contract are not languages of strings, so we define a rule system that work on trees and decides the clause $H \models \varphi_\sigma$.
At static time the orchestrator will associate with each request the right service that comply with the client. Note that we have defined a type of policies that can interleave with safety policies, so we can impose safety properties over a CCS contract. The orchestrator checks if the policy φ_σ is valid by the rule system we define below. The first step consists in removing from the history expression H all the safety policies φ, we write $H^\flat$ for an history expression without safety policies, for checking only policies φ_σ. Note that for the safety policies φ the internal and external choice are treated in the same manner. When the orchestrator encounters a history expression $\varphi_\sigma[H]$ it will check if it is valid by computing the value of the clause $H \models \varphi_\sigma$.

Definition 14. *A history expression $\varphi_\sigma[H]$ is valid if and only if $H \models \varphi_\sigma$ where:*
$H \models \varphi_\sigma \longleftrightarrow (H^\flat, \emptyset) \models \varphi_\sigma$

Here we define when a history expression is φ_σ-valid, that is when a history expression respects the contracts which occuring in it. Here we noting that a contract φ_σ only inspects its local histories and not all the past histories as done before by safety policies [2].

Definition 15. *A history expression $H^\flat$ is $\varphi_\sigma - valid$ if and only if:*

$H = h$
$H = \varepsilon$
$H = \varphi_\sigma[H'] \Rightarrow H' \models \varphi_\sigma \wedge H'\ is\ \varphi_\sigma - valid$
$H = H' \cdot H'' \Rightarrow H'\ is\ \varphi_\sigma - valid \wedge H''\ is\ \varphi_\sigma - valid$
$H = H' \oplus H'' \Rightarrow H'\ is\ \varphi_\sigma - valid \wedge H''\ is\ \varphi_\sigma - valid$
$H = H' + H'' \Rightarrow H'\ is\ \varphi_\sigma - valid \wedge H''\ is\ \varphi_\sigma - valid$
$H = \mu h.H' \Rightarrow H'\ is\ \varphi_\sigma - valid$
$H = \alpha.H' \Rightarrow H'\ is\ \varphi_\sigma - valid$
$H = \{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\} \Rightarrow H_1\ is\ \varphi_\sigma - valid \wedge \cdots \wedge H_k\ is\ \varphi_\sigma - valid$

The rules for checking wheter $\varphi_\sigma[H]$ is valid are defined below. We use a set Δ of pairs of $H \times \varphi_\sigma$ for the unfolding of recursion and we start with $\Delta = \emptyset$. In $IC1$, φ_σ internally decides to continue as $\varphi_{\sigma i}$ and H must comply

with all the possible choice, so we have all the possible clauses in *and*. In $IC2$, H makes the internal choice, note that we impose $|I| > 1$ since if $|I| = 1$ we use the rule EC. In EC, it suffices that one of the possible choice is valid, so we have an *or* of all the possible clauses. In $ST1$, H and φ_σ are able to synchronize on an action and both made a step. In $ST2$, H and φ_σ cannot synchronize so the action of H is discarded. This rule can be used only for φ_{client}, since inside the history expression of the server there could be other requests or operations different of those specified by the policy. This is not true for φ_{server} since we split the policy into different level and inside each level we require H^i_{client} to perform only the actions of φ^i_{server}. Note that the client service can perform other actions or requests since σ_{client} represent a single request and not all the history expression.
In $ST3$ we skip another frame, this happens when inside a φ_{client} frame there are φ_{server} frames. In $RE1$ and $RE2$, we made a step in the unfolding for H and φ_σ respectively. Note that in φ_σ it is necessary to use a marker ${\varphi_\sigma}^*$ to check that the rule has been applied at least once. In $RJ1$, we reject if H haven't performed all the actions required by φ_σ. In $RJ2$, we reject because we get into an infinite loop where the two services never synchronize. In $AC1$, we accept since all the actions of φ_σ are performed. In $AC2$, we accept since the two services comply infinitely. In $AC3$, we accept since the client has decided to terminate. Note that * does not represent the empty history ε but the special character introduced by the function $\mathcal{B}$ only in σ_{client}. We need rule $AC3$ for accepting φ_{server} on a client that is already terminated.

$\Delta = H \times \varphi_\sigma$
$(H, \Delta) \models \varphi_\sigma :$
$IC1 : (H, \varphi_\sigma) \notin \Delta \wedge \varphi_\sigma = \bigoplus_{i \in I, |I|>1} \varphi_{\sigma_i} \Longrightarrow \bigwedge_{i \in I} (H, \Delta) \models \varphi_{\sigma_i}$
$IC2 : (H, \varphi_\sigma) \notin \Delta \wedge H = \bigoplus_{i \in I, |I|>1} H_i \Longrightarrow \bigwedge_{i \in I} (H_i, \Delta) \models \varphi_\sigma$
$EC : (H, \varphi_\sigma) \notin \Delta \wedge \varphi_\sigma = \sum_{i \in I} \varphi_{\sigma_i} \wedge H = \sum_{j \in J} H_j \Longrightarrow \bigvee_{i \in I, j \in J} (H_j, \Delta) \models \varphi_{\sigma_i}$
$ST1 : (H, \varphi_\sigma) \notin \Delta \wedge \varphi_\sigma = a.\varphi_{\sigma'} \wedge H = b.H' \wedge a = b \Longrightarrow (H', \Delta) \models \varphi_{\sigma'}$
$ST2 : (H, \varphi_\sigma) \notin \Delta \wedge \varphi_\sigma = a.\varphi_{\sigma'} \wedge H = b.H' \wedge a \neq b \wedge \sigma = \sigma_{client} \Longrightarrow (H', \Delta) \models \varphi_\sigma$
$ST3 : (H, \varphi_\sigma) \notin \Delta \wedge H = \varphi'_\sigma[H'] \cdot H'' \Longrightarrow (H'', \Delta) \models \varphi_\sigma$
$RE1 : (H, \varphi_\sigma) \notin \Delta \wedge H = \mu h.H' \Longrightarrow (H'\{^H/_h\}, \Delta \cup \{(H, \varphi_\sigma)\}) \models \varphi_\sigma$
$RE2 : (H, \varphi_\sigma) \notin \Delta \wedge \varphi_\sigma = \mathsf{rec}\ x = \varphi'_\sigma \Longrightarrow (H, \Delta \cup \{(H, \varphi_\sigma)\}) \models \varphi'_\sigma\{^{\varphi^*_\sigma}/_x\}$
$RJ1 : H = \emptyset \wedge \varphi_\sigma \neq 0 \Longrightarrow false$
$RJ2 : (H, \varphi_\sigma) \in \Delta \wedge \neg \varphi^*_\sigma \Longrightarrow false$
$AC1 : \varphi_\sigma = 0 \Longrightarrow true$
$AC2 : (H, \varphi_\sigma) \in \Delta \wedge \varphi^*_\sigma \Longrightarrow true$
$AC3 : H = *.H' \Longrightarrow true$

With this rules the orchestrator binds services and requests only if the compliance is ensured by the policy φ_{client} and φ_{server}. Note that if we only use φ_{client}, two contract like $\sigma_1 = a$ and $\sigma_2 = b.a$ result valid while they are not compliant, since we have the weakening rule $ST2$. To ensure that the contracts synchronize on an action, we need both φ_{client} and φ_{server}. In the following theorem we use unfolded contracts which are infinite terms generated by repeated infinitely the unfolding of recursive contracts. We know that a recursive contract $\sigma = \mathsf{rec}\, x = \sigma'$ equals his unfolded contract $\sigma'\{^{\sigma}/_{x}\}$[8], so we have: $\sigma_{client} \dashv \sigma_{server} \longleftrightarrow \sigma'_{client}\{^{\sigma_{client}}/_{x}\} \dashv \sigma'_{server}\{^{\sigma_{server}}/_{x}\}$. The following results are stated for unfolded contracts, we conjecture that they hold for the corresponding recursive contracts too. In the following theorem we show that two unfolded contracts complies if and only if the respective translation in λ^{req} is valid.

Theorem 1. *Given two unfolded contract σ_{client} and σ_{server} and the associates λ^{req} expressions :*
$e_{client} : req_{r_{client}}(\tau_{client} \overset{\varphi_c}{\longmapsto} 1)(\mathcal{C}[\![\mathcal{B}[\![\sigma_{client}]\!]\mathcal{D}(\sigma_{server})]\!]0)$
with $H_{client} \vdash \mathcal{C}[\![\mathcal{B}[\![\sigma_{client}]\!]\mathcal{D}(\sigma_{server})]\!]0 : \tau_{client}$
$e_{server} : \lambda f_0.\mathcal{S}[\![\sigma^n_{server}]\!]0$ *we have:*

$\sigma_{client} \dashv \sigma_{server} \longleftrightarrow \{r_{client}[\ell_{server}] \rhd H_{server}\}$ *is φ_σ-valid*

Proof. $\longrightarrow$ By structural induction on σ_{client}:
$\sigma_{client} = 0$: we have $\varphi_{client} = \emptyset$ that is compliant with every σ_{server}; by rule $AC1$ $\varphi_{client}[H_{server}]$ evaluates to true. Also since σ_{client} is balanced with $\mathcal{D}(\sigma_{server})$, we have that $\forall i.\varphi^i_s[*]$ evaluates to true by rule $AC3$.

$\sigma_{client} = \alpha.\sigma'_{client}$: since the two contracts are compliant the server contract is able to synchronize with α, so it is in the form $\sigma_{server} = \alpha.\sigma'_{server} + \sigma''_{server}$ and $\sigma'_{client} \dashv \sigma'_{server}$. We have $\varphi_{client} = \alpha.\varphi'_{client}$, $\varphi^i_s = \alpha + \cdots$ and $H_{server} = \varphi_{client}[\varphi^i_s[\alpha] \cdot \alpha \cdot H'_{server} + \varphi^i_s[\alpha] \cdot H''_{server}]$, for φ_{client} by rule EC $ST3$ and $ST1$ we obtain $H'_{server} \models \varphi'_{client} \vee \cdots \Rightarrow true$, for φ^i_s we have $\alpha \models \alpha + \cdots \Rightarrow true$.

$\sigma_{client} = \sum_{i\in I} \sigma^i_{client}$: since the two contracts are compliant the server is able to synchronize with one or more σ^i_{client}, so it is in the form $\sigma_{server} = \sum_{z\in Z} \sigma^z_{server} + \sigma''_{server}$ or $\sigma_{server} = \bigoplus_{j\in J} \sigma^j_{server} + \sigma''_{server}$ where $0 < |J| \leq |I|, |Z| > 0$ and $\forall j \in J, z \in Z.\sigma^j_{client} \dashv \sigma^j_{server} \wedge \sigma^z_{client} \dashv \sigma^z_{server}$. So we have at least one $\sigma^{j/z}_{server}$ that complies with $\sigma^{j/z}_{client}$. We have $\varphi_{client} = \sum_{i\in I} \varphi^i_{client}$ and

[8] a complete proof can be found on [3]

$H_{server} = \varphi_{client}[\sum_{z\in Z} H^z_{server} + H''_{server}]$ or $H_{server} = \varphi_{client}[\bigoplus_{j\in J} H^j_{server} + H''_{server}]$. In the first case by rule EC we obtain $H^z_{server} \models \varphi^z_{client} \vee \cdots \Rightarrow true$ and in the second case by rule EC and $IC2$ we obtain $H^j_{server} \models \bigwedge_{j\in J} \varphi^j_{client} \vee \cdots \Rightarrow true$ by induction hypothesis.

$\sigma_{client} = \bigoplus_{i\in I} \sigma^i_{client}$: since the two contracts comply the server is able to synchronize with all the σ^i_{client}, so it is in the form $\sigma_{server} = \sum_{j\in J} \sigma^j_{server} + \sigma''_{server}$ where $|J| > |I|$ and $\forall i \in I : \sigma^i_{client} \dashv \sigma^i_{server}$. We have $\varphi_{client} = \bigoplus_{i\in I} \varphi^i_{client}$ and $H_{server} = \varphi_{client}[\sum_{j\in J} H^j_{server} + H''_{server}]$ and by rule $IC1$ and EC we obtain $H^i_{server} \models \bigwedge_{i\in I} \varphi^i_{client} \vee \cdots \Rightarrow true$ by induction hypothesis.

$\longleftarrow$ by structural induction on σ_{client}:
$\sigma_{client} = 0$: for every σ_{server} we have $\sigma_{client} \dashv \sigma_{server}$

$\sigma_{client} = \alpha.\sigma'_{client}$: since H_{server} is $\varphi_\sigma-$valid it must be in the form $\alpha.\sigma'_{server} + \sigma''_{server}$ or by contradiction $\varphi^i_s[\alpha]$ is not valid, by induction hypothesis we have $\sigma'_{client} \dashv \sigma'_{server}$, and we conclude $\sigma_{client} \dashv \sigma_{server}$

$\sigma_{client} = \sum_{i\in I} \sigma^i_{client}$: by hypothesis we have that $H_{server} = H'''_{server} \cdot (\sum_{z\in Z} H^z_{server} + H''_{server})$ or $H_{server} = H'''_{server} \cdot (\bigoplus_{j\in J} H^j_{server} + H''_{server})$ where $0 < |J| \leq |I|, |Z| > 0$ and there must be $H'''_{server} = \varepsilon$ or we have $\varphi_{H'''_{server}} \neq 0$ and by contradiction H_{server} is not valid; so we have $\sigma_{server} = \sum_{z\in Z} \sigma^z_{server} + \sigma''_{server}$ or $H_{server} = \bigoplus_{j\in J} \sigma^j_{server} + \sigma''_{server}$ and by induction hypothesis we obtain $\sigma_{client} \dashv \sigma_{server}$

$\sigma_{client} = \bigoplus_{i\in I} \sigma^i_{client}$: by hypothesis we have $H_{server} = H'''_{server} \cdot (\sum_{j\in J} H^j_{server} + H''_{server})$ where $|J| > |I|$ and there must be $H'''_{server} = \varepsilon$ or we have $\varphi_{H'''_{server}} \neq 0$ and by contradiction H_{server} is not valid; so we have $\sigma_{client} = \bigoplus_{i\in I} \sigma^i_{client}$ and $\sigma_{server} = \sum_{j\in J} \sigma^j_{server} + \sigma''_{server}$ and we have $\sigma_{client} \dashv \sigma_{server}$

□

The following example show the use of the marker $\varphi_\sigma{}^*$, if we do not use the marker the two following services would comply:

Example 7.0.5
$\varphi_\sigma = b \quad H = \mu h.ah$

$(\mu h.ah, \emptyset) \models b \stackrel{RE1}{\Longrightarrow} (a.H, \{(H, b)\}) \models b \stackrel{ST2}{\Longrightarrow}$
$(H, \{(H, b)\}) \models b \stackrel{RJ2}{\Longrightarrow} false$

In the following example the two services correctly complies:

Example 7.0.6
$\varphi_\sigma = ((a+b) \oplus c) + d.e \quad H = (b+c) \oplus d.e$

$((b+c) \oplus d.e, \emptyset) \models ((a+b) \oplus c) + d.e \stackrel{IC2}{\Longrightarrow}$

$(b+c, \emptyset) \models ((a+b) \oplus c) + d.e \wedge (d.e, \emptyset) \models ((a+b) \oplus c) + d.e \stackrel{EC}{\Longrightarrow}$

$((b+c, \emptyset) \stackrel{1}{\models} (a+b) \oplus c \vee (b+c, \emptyset) \stackrel{2}{\models} d.e) \wedge$
$((d.e, \emptyset) \stackrel{3}{\models} (a+b) \oplus c \vee (d.e, \emptyset) \stackrel{4}{\models} d.e)$

1 $(b+c, \emptyset) \models (a+b) \oplus c \stackrel{IC1}{\Longrightarrow} ((b+c, \emptyset) \models a+b) \wedge ((b+c, \emptyset) \models c) \stackrel{EC}{\Longrightarrow}$
$((b, \emptyset) \models a \vee (b, \emptyset) \models b) \vee (c, \emptyset) \models a \vee (c, \emptyset) \models b)) \wedge$
$((b, \emptyset) \models c) \vee (c, \emptyset) \models c) \stackrel{ST1,ST2,RJ1,AC1}{\Longrightarrow} \dots$
$(false \vee true \vee false \vee false) \wedge (false \vee true) = true$
2 $(b+c, \emptyset) \models d.e \stackrel{EC}{\Longrightarrow} (b, \emptyset) \models d.e \vee (c, \emptyset) \models d.e \stackrel{ST2}{\Longrightarrow}$
$(0, \emptyset) \models d.e \vee (0, \emptyset) \models d.e \stackrel{RJ1}{\Longrightarrow} false$
3 $(d.e, \emptyset) \models (a+b) \oplus c \stackrel{IC1}{\Longrightarrow} (d.e, \emptyset) \models a+b \wedge (d.e, \emptyset) \models c \stackrel{EC}{\Longrightarrow}$
$((d.e, \emptyset) \models a \vee (d.e, \emptyset) \models b) \wedge (d.e, \emptyset) \models c \stackrel{ST2 \times 2}{\Longrightarrow}$
$((0, \emptyset) \models a \vee (0, \emptyset) \models b) \wedge (0, \emptyset) \models c \stackrel{RJ1}{\Longrightarrow} (false \vee false) \wedge false = false$
4 $(d.e, \emptyset) \models d.e) \stackrel{ST1 \times 2}{\Longrightarrow} (0, \emptyset) \models 0 \stackrel{AC1}{\Longrightarrow} true$

$(true \vee false) \wedge (false \vee true) = true \Longrightarrow H \models \varphi_\sigma : true$

The following examples show the use of the rule $RE2$ and $AC2$:

Example 7.0.7
$\varphi_\sigma = \mathsf{rec}\ x = a.x \oplus b.c.x \quad H = \mu h.a.h + b.c.h$

$(\mu h.a.h + b.c.h, \emptyset) \models \mathsf{rec}\ x = a.x \oplus b.c.x \stackrel{RE1}{\Longrightarrow}$

$(H' = a.H + b.c.H, \{(H, \varphi_\sigma)\}) \models \mathsf{rec}\ x = a.x \oplus b.c.x \stackrel{RE2}{\Longrightarrow}$

$(H', \Delta = \{(H, \varphi_\sigma), (H', \varphi_\sigma)\}) \models a.\varphi_\sigma^* \oplus b.c.\varphi_\sigma^* \stackrel{IC1}{\Longrightarrow}$

$(H', \Delta) \stackrel{1}{\models} a.\varphi_\sigma^* \wedge (H', \Delta) \stackrel{2}{\models} b.c.\varphi_\sigma^*$

1 $(a.H + b.c.H, \Delta) \models a.\varphi_\sigma^* \stackrel{EC}{\Longrightarrow}$
$(a.H, \Delta) \models a.\varphi_\sigma^* \vee (b.c.H, \Delta) \models a.\varphi_\sigma^* \stackrel{ST1}{\Longrightarrow}$
$(H, \Delta) \models \varphi_\sigma^* \vee (b.c.H, \Delta) \models a.\varphi_\sigma^* \stackrel{AC2}{\Longrightarrow}$
$true \vee (b.c.H, \Delta) \models a.\varphi_\sigma^* = true$

2 $(a.H + b.c.H, \Delta) \models b.c.\varphi_\sigma^* \stackrel{EC}{\Longrightarrow}$
$(a.H, \Delta) \models b.c.\varphi_\sigma^* \vee (b.c.H, \Delta) \models b.c.\varphi_\sigma^* \stackrel{ST1\times 2}{\Longrightarrow}$
$(a.H, \Delta) \models b.c.\varphi_\sigma^* \vee (H, \Delta) \models \varphi_\sigma^* \stackrel{AC2}{\Longrightarrow}$
$(a.H, \Delta) \models b.c.\varphi_\sigma^* \vee true = true$

$(\mu h.a.h + b.c.h, \emptyset) \models \mathsf{rec}\ x = a.x \oplus b.c.x : true \wedge true = true$

Example 7.0.8
$\varphi_\sigma = \mathsf{rec}\ x = a.x \quad H = a.\mu h.a.h$

$(a.\mu h.a.h, \emptyset) \models \mathsf{rec}\ x = a.x \stackrel{RE1}{\Longrightarrow}$

$(a.\mu h.a.h, \{(a.\mu h.a.h, \varphi_\sigma)\}) \models a.\varphi_\sigma^* \stackrel{ST1}{\Longrightarrow}$

$(\mu h.a.h, \{(a.\mu h.a.h, \varphi_\sigma)\}) \models \varphi_\sigma^* = (\mathsf{rec}\ x = a.x)^* \stackrel{RE2}{\Longrightarrow}$

$(\mu h.a.h, \{(a.\mu h.a.h, \varphi_\sigma), (\mu h.a.h, \varphi_\sigma^*)\}) \models a.\varphi_\sigma^* \stackrel{RE1}{\Longrightarrow}$

$(a.\mu h.a.h, \Delta = \{(a.\mu h.a.h, \varphi_\sigma), (\mu h.a.h, \varphi_\sigma^*), (\mu h.a.h, a.\varphi_\sigma^*)\}) \models a.\varphi_\sigma^* \stackrel{ST1}{\Longrightarrow}$

$(\mu h.ah, \Delta) \models \varphi_\sigma^* \stackrel{AC2}{\Longrightarrow} true$

$(a.\mu h.a.h, \emptyset) \models \mathsf{rec}\ x = a.x : true$

7.1 Subcontract

In the first section we recalled the subcontract relation $\preceq$. This relation holds in λ^{req} too, so we can replace a service σ_1 with another service σ_2 if $\sigma_1 \preceq \sigma_2$. Before stating the theorem, we give some intuition: for the width sub-typing we note that adding more choices in an external choice(i.e. $a + b \preceq a + b + c$) can never break the validity of the composition since in the validations we have another clause in or(i.e. $a \vee b = true \rightarrow a \vee b \vee c = true$). For a more deterministic contract like $a \oplus b \preceq a$ we note that erasing an internal

choice cannot break validity, since in the validations we erase a clause in *and* with other clauses(i.e. $a \wedge b = true \rightarrow a = true$). For the depth sub-typing like $a \preceq a.b$ we note that $H_{server} \models \varphi_{client}$ is true when all the actions in φ_{client} are performed by H_{server}; validity is preserved also if H_{server} performs other actions. $H^i_{client} \models \varphi^i_s$ is true since the λ^{req} client is balanced with the depth of the server and this will add special actions $*$ that by rule $AC3$ are recognized by the policies φ^i_s.
The following theorem shows that the substitution never breaks validity of compositions.

Theorem 2. *Let σ_c be a client contract and σ_{s1}, σ_{s2} be server contracts where $\sigma_c \dashv \sigma_{s1}$, let e_c be the λ^{req} expression for the client contract and e_{s1}, e_{s2} be the λ^{req} expressions for the server contracts and let $\pi = r_{client}[\ell_{s1}]|\pi''$ be the plan that binds the request of the client e_c to the server e_{s1}, furthermore the history generated is π-valid.*

If $\sigma_{s1} \preceq \sigma_{s2}$ then the history generated under the new plan $\pi' = r_{client}[\ell_{s2}]|\pi''$ is π-valid.

Proof. Since $\sigma_{s1} \preceq \sigma_{s2}$ and $\sigma_c \dashv \sigma_{s1}$ it follows that $\sigma_c \dashv \sigma_{s2}$ by definition of $\preceq$, the thesis follow from Theorem 1. □

7.2 Multi-Party

In λ^{req} we can model multi-party interaction: if we want a service to performs different client contracts $\sigma_{c1} \cdots \sigma_{cn}$ we simply create the λ^{req} client service with all the requests in cascade. In this way each request can be taken by a different service. Note that if we make a single request with the contract $\sigma = \bigoplus_{i \in I} \sigma_{ci}$, the request will be satisfied only if there is a single service able to comply with all the client contracts.

Definition 16. *Let $\sigma_{c1} \cdots \sigma_{cn}$ be client contracts, we define the λ^{req} multi-party client as:*

$req_{r1}(\tau_{val1} \xrightarrow{\varphi_{c1}} 1)(\mathcal{C}[\![\sigma^b_{c1}]\!]0);$
$req_{r2}(\tau_{val2} \xrightarrow{\varphi_{c2}} 1)(\mathcal{C}[\![\sigma^b_{c2}]\!]0);$
$\cdots$
$req_{rn}(\tau_{valn} \xrightarrow{\varphi_{cn}} 1)(\mathcal{C}[\![\sigma^b_{cn}]\!]0)$

with $\mathcal{C}[\![\sigma^b_{c1}]\!]0 : \tau_{val1}, \cdots, \mathcal{C}[\![\sigma^b_{cn}]\!]0 : \tau_{valn}$

If we want a service to perform different service contracts $\sigma_{s1} \cdots \sigma_{sn}$, we can compose the services with the external choice. The service will be planned to hold more requests. Note that all the σ_{si} are balanced with the most general contracts.

Definition 17. *Let $\sigma_{s1} \cdots \sigma_{sn}$ be server contracts, we define the λ^{req} multi-party server as:*

$\lambda f_0.\mathcal{S}[\![((\sum_{i \in I} \sigma_{si})^b)^n]\!]0$

We can also make a service perform a client contract and a server contract. Let $[\sigma]$ be a client contract inside a server contract, we define the rule for the compositions of a client contract and a server contract.

Definition 18. *Let σ_c be a client contract and σ_s a server contract. The λ^{req} server of the composition $[\sigma_c].\sigma_s$ is defined as:*

$$\begin{aligned}\mathcal{S}[\![[\sigma_c].\sigma_s]\!]b =& req_r(\tau_{val} \xrightarrow{\varphi_c} 1)(\mathcal{C}[\![\sigma_c^b]\!]0) \\ & \mathcal{S}[\![\sigma_s]\!]b \quad b \in 0,1\end{aligned}$$

with $\mathcal{C}[\![\sigma_c^b]\!]0 : \tau_{val}$

Here we give the classic example where a client asks a service to book a flight and book an hotel room. The service acts like a broker: by asking an airline to book the flight and an hotel to book a room. Then it answer the client.

Example 7.2.1
The contracts to be transformed are:

$$\begin{aligned}\sigma_{client} =& \overline{FlightRequest}(FlightConfirmed. \\ & .\overline{HotelRoomRequest}(RoomConfirmed + RoomDenied) + FlightDenied)\end{aligned}$$

$\sigma_{broker-server} = FlightRequest.[\sigma_{broker-airline-client}].(\overline{FlightConfirmed}.$
$HotelRoomRequest.[\sigma_{broker-hotel-client}].(\overline{RoomConfirmed} \oplus \overline{RoomDenied})$
$\oplus \overline{FlightDenied})$

$\sigma_{broker-airline-client} = \overline{BookFlight}.(BookFlightConfirmed + BookFlightDenied)$

$\sigma_{broker-hotel-client} = \overline{BookRoom}.(BookRoomConfirmed + BookRoomDenied)$

$\sigma_{airline-server} = BookFlight.(\overline{BookFlightConfirmed} + \overline{BookFlightDenied})$

$\sigma_{hotel-server} = BookRoom.(\overline{BookRoomConfirmed} \oplus \overline{BookRoomDenied})$

Note that the broker service implements three different contracts.
Here we give the λ^{req} expressions:

Client :
$req_{r1}(1 \to (1 \to (1 \to (1 \to 1))) \xrightarrow{\varphi_{client}} 1)$
$(\lambda.FlightRequest;$
$\lambda cx.$if $(cx = *_{FlightConfirmed})$ then
$DflightConfirmed;$
$\lambda.HotelRoomRequest;$
$\lambda cx2.$if $(cx2 = *_{RoomConfirmed})$ then
$DRoomConfirmed;$
else
$DRoomDenied;$
else
$DFlightDenied;$
$\lambda.*;$
$\lambda.*);$

$\varphi_{client} = DFlightRequest(FlightConfirmed.$
$.DHotelRoomRequest(RoomConfirmed + RoomDenied) + FlightDenied)$

AirlineServer :
$\lambda f_0.\varphi^0_{as}[f_0 *_{bookflight}]; f_1.(DBookFlight;$
if $(\cdots)$ then
$\varphi^1_{as}[f_1 *_{bookflightconfirmed}]; f_2.($
$BookFlightConfirmed)$
else
$\varphi^1_{as}[f_1 *_{bookflightdenied}]; f_2.($
$BookFlightDenied))$

$\varphi^0_{as} = BookFlight$
$\varphi^1_{as} = DBookFlightConfirmed \oplus DBookFlightDenied$

HotelServer :

$\lambda f_0.\varphi^0_{hs}[f_0 *_{bookroom}]; f_1.(DBookRoom;$
if $(\cdots)$ then
$\varphi^1_{hs}[f_1 *_{bookroomconfirmed}]; f_2.($
$BookRoomConfirmed)$
else
$\varphi^1_{hs}[f_1 *_{bookroomdenied}]; f_2.($
$BookRoomDenied))$

$\varphi^0_{hs} = BookRoom$
$\varphi^1_{hs} = DBookRoomConfirmed \oplus DBookRoomDenied$

Broker :

$\lambda f_0.\varphi^0_{is}[f_0 *_{flightrequest}]; f_1.(DFlightRequest;$

$(req_{r2}(1 \to (1 \to 1) \xrightarrow{\varphi_{airline-client}} 1)$
$(\lambda.BookFlight;$
$\lambda cx.$if $(cx == *_{bookflightconfirmed})$ then
$DBookFlightConfirmed;$
else
$DBookFlightDenied;))$

if $(\cdots)$ then
$\varphi^1_{is}[f_1 *_{flightconfirmed}]; f_2.(FlightConfirmed;$
$\varphi^2_{is}[f_2 *_{DHotelRoomRequest}]; f_3.(DHotelRoomRequest;$

$(req_{r3}(1 \to (1 \to 1) \xrightarrow{\varphi_{hotel-client}} 1)$
$(\lambda.BookRoom;$
$\lambda cx.$if $(cx == *_{bookroomconfirmed})$ then
$DBookRoomConfirmed;$
else
$DBookRoomDenied;))$

if $(\cdots)$ then
$\varphi^3_{is}[f_3 *_{RoomConfirmed}]; f_4.(RoomConfirmed; *)$
else
$\varphi^3_{is}[f_3 *_{RoomDenied}]; f_4.(RoomDenied; *)))$
else
$\varphi^1_{hs}[f_1 *_{flightdenied}]; f_2.(FlightDenied; *))$

$\varphi_{airline-client} = DBookFlight(BookFlightConfirmed + BookFlightDenied)$
$\varphi_{hotel-client} = DBookRoom(BookRoomConfirmed + BookRoomDenied)$
$\varphi^0_{is} = FlightRequest$
$\varphi^1_{is} = DFlightConfirmed \oplus DFlightDenied$
$\varphi^2_{is} = HotelRoomRequest \oplus \varepsilon$
$\varphi^3_{is} = (DRoomConfirmed \oplus DRoomDenied) \oplus \varepsilon$

$\tau_{broker} = (1 \to (1 \to (1 \to (1 \to 1)))) \longrightarrow 1$
$\tau_{airline-server} = (1 \to (1 \to 1)) \longrightarrow 1$
$\tau_{hotel-server} = (1 \to (1 \to 1)) \longrightarrow 1$

The plan will be :
$\pi = r1[\ell_{broker}]|r2[\ell_{airline-server}]|r3[\ell_{hotel-server}]$

The history expressions generated will be:

$H_{client} = \varepsilon$

$H_{airline} = \varphi_{airline-client}[\varphi^0_{as}[BookFlight] \cdot DBookFlight($
$\varphi^1_{as}[DBookFlightConfirmed + DBookFlightDenied] \cdot BookFlightConfirmed$
$\oplus \varphi^1_{as}[DBookFlightConfirmed + DBookFlightDenied] \cdot BookFlightDenied)]$

$H_{hotel} = \varphi_{hotel-client}[\varphi^0_{hs}[BookRoom] \cdot DBookRoom($
$\varphi^1_{hs}[DBookRoomConfirmed + DBookRoomDenied] \cdot BookRoomConfirmed$
$\oplus \varphi^1_{hs}[DBookRoomConfirmed + DBookRoomDenied] \cdot BookRoomDenied)]$

$H_{broker} = \varphi_{client}[\varphi^0_{is}[FlightRequest] \cdot DFlightRequest($
$\varphi^1_{is}[DFlightConfirmed + DFlightDenied] \cdot FlightConfirmed \cdot$
$\varphi^2_{is}[HotelRoomRequest + \varepsilon] \cdot DHotelRoomRequest($
$\varphi^2_{is}[(DRoomConfirmed + DRoomDenied) + \varepsilon] \cdot RoomConfirmed \oplus$
$\varphi^2_{is}[(DRoomConfirmed + DRoomDenied) + \varepsilon] \cdot RoomDenied)$
$\oplus \varphi^1_{is}[DFlightConfirmed + DFlightDenied] \cdot FlightConfirmed]$

As we can see, all the history expressions are valid. Thus at runtime the services will behave as defined by their contracts, and the interactions terminate successfully.

8 Conclusions

In this dissertation we introduced contracts in λ^{req}. This gave us the advantage of expressing multi-party interactions, which is not possible with the contracts of [3]. It was also possible to express security policies on contracts, sessions in λ^{req}, and the substitution of a service with an equivalent one without breaking the validity of the composition. The transformation of contracts in λ^{req} expressions also provide an executable specification of a service.

Other extensions concerns the extension of the proof of validity from contracts as infinite trees to recursive finite contracts. Furthermore we would like to study if it is possible to avoid dual actions, and checking the compliance with a single policy for both clients and services.

References

[1] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, Roberto Zunino. *Call-by-Contract and Secure Service Orchestration, 2009.*

[2] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari. *Planning and verifying service composition, 2009.*

[3] Giuseppe Castagna, Niel Gesbert, Luca Padovani. *A Theory of Contracts for Web Services, 2008.*

[4] M.Papazoglou and D.Georgakopoulos. *Special issue on service oriented computing, 2003.*

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

[6] R.Milner. *A Calculus of Communicating Systems, 1980*

[7] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari. *Types and effects for secure service orchestration, 2006.*

[8] A. Church. *A Formulation of the Simple Theory of Types, 1940*