Secure Service Orchestration

Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino

Dipartimento di Informatica, Università di Pisa

Abstract. We present a framework for designing and composing services in a secure manner. Services can enforce security policies locally, and can invoke other services in a "call-by-contract" fashion. This mechanism offers a significant set of opportunities, each driving secure ways to compose services. We discuss how to correctly plan service orchestrations in some relevant classes of services and security properties. To this aim, we propose both a core functional calculus for services and a graphical design language. The core calculus is called λ^{req} [10]. It features primitives for selecting and invoking services that respect given behavioural requirements. Critical code can be enclosed in security framings, with a possibly nested, local scope. These framings enforce safety properties on execution histories. A type and effect system over-approximates the actual run-time behaviour of services. Effects include the actions with possible security concerns, as well as information about which services may be selected at run-time. A verification step on these effects allows for detecting the viable plans that drive the selection of those services that match the security requirements on demand.

1 Introduction

The Web service protocol stack (WSDL, UDDI, SOAP, WSBPEL) offers basic support for the development of service-oriented architectures, including facilities to publish, discover and orchestrate services. Although this has been extremely valuable to highlight the key innovative features of the service-oriented approach, experience has singled out several limiting factors of the service protocol stack, mainly because of the purely "syntactic" nature of standards. This has lead to the idea of extending the stack with higher level, "semantic" functionalities. For instance, the design and exploitation of service ontologies is a first attempt to address these concerns.

A challenging issue of the service approach is how to *orchestrate* existing services into more complex ones, by properly selecting and configuring services so to guarantee that their composition enjoys some desirable properties. These properties may involve *functional* aspects, speaking about the goals attained by a service, and also *non-functional* aspects, like e.g. security, availability, performance, transactionality, quality of service, etc. [45].

In this paper we describe a semantics-based framework to model and orchestrate services in the presence of both functional and non-functional constraints, with a special concern for security properties. The formal foundation of our work

A. Aldini and R. Gorrieri (Eds.): FOSAD 2006/2007, LNCS 4677, pp. 24–74, 2007.

is λ^{req} [10,6], a core calculus for securely orchestrating services. The λ^{req} calculus extends the λ-calculus with primitive constructs to describe and invoke services in a *call-by-contract* fashion. Services are modelled as functions with side effects. These side effects represent the action of accessing security-critical resources, and they are logged into *histories*. A run-time security monitor may inspect histories, and forbids those executions that would violate the prescribed policies.

Unlike standard discovery mechanisms that match syntactic signatures only, ours also implements a matchmaking algorithm based on service behaviour. This algorithm exploits static analysis techniques to resolve the call-by-contract involved in a service orchestration. The published interface of a service takes the form of an annotated type, which represents both the signature of the service (i.e. its input-output information) and a suitable semantic abstraction of the service behaviour. In our call-by-contract selection, the client is required to know neither the service name nor its location. Operationally, the service registry is searched for a service with a functional type (the service signature) matching the request type; also, the semantic abstraction must respect the non-functional constraints imposed by the request. Our orchestration machinery constructs a *plan* for the execution of services, e.g. a binding between requests and service locations, guaranteeing that the properties on demand are always satisfied.

We envisage the impact of our approach on the service protocol stack as follows. First, it requires extending services description languages: besides the standard WSDL attributes, service description should include semantic information about service behaviour. Moreover, the call-by-contract invocation mechanism adds a further layer to the standard service protocol stack: the *planning* layer. This layer provides the orchestrator with the plans guaranteeing that the orchestrated services always respect the required properties. Hence, before starting the execution of the orchestration, the orchestrator engines collects the relevant service plans by inquiring the planning layer. These plans enable the orchestration engine to resolve all the requests in the initiator service, as well as those in the invoked services.

1.1 Service Interfaces and Contracts

In our approach, the published interface of a service is an annotated functional type, of the form $\tau_1 \xrightarrow{H} \tau_2$. When supplied with an argument of type τ_1, the service evaluates to an object of type τ_2. The annotation H is a *history expression*, a sort of context-free grammar that abstractly describes the possible run-time histories of the service. Thus, H will be exploited to guide the selection of those services that respect the requested properties about security or other non-functional aspects. Since service interfaces are crucial in the implementation of the call-by-contract primitive, they have to be *certified* by a trusted party, which guarantees that the abstract behaviour is a sound over-approximation of the actual service behaviour. For instance, service interfaces can be mechanically inferred through a type and effect system, as shown in Section 8.

A *contract* φ is a regular property of execution histories. We express contracts as languages accepted by finite state automata. Although in this paper we mainly

focus on security policies, in the general case contracts can be arbitrary safety properties (e.g. resource usage constraints [7]).

To select a service matching a given contract φ, and with functional type $\tau_1 \to \tau_2$, a client issues a request of the form $\texttt{req}\,(\tau_1 \xrightarrow{\varphi} \tau_2)$. The call-by-contract mechanism ensures that the selected service, with interface $\tau_1 \xrightarrow{H} \tau_2$, will always respect the contract φ, i.e. that all the histories represented by H are recognized by the automaton defining φ.

Since service interactions may be complex, it might be the case that a local choice for a service is not secure in a broader, "global" context. For instance, choosing a low-security e-mail provider might prevent you from using a home-banking service that exchanges confidential data through e-mail. In this case, you should have planned the selection of the e-mail and bank services so to ensure their compatibility. To cope with this kind of issues, we define a static machinery that determines the *viable plans* for selecting services that respect all the contracts, both locally and globally. A plan resolves a call-by-contract into a standard service call, and it is formalized as a mapping from requests to services.

1.2 Planning Service Composition

Our planning technique acts as a *trusted orchestrator* of services. It provides a client with the viable plans guaranteeing that the invoked services always respect the required properties. Thus, in our framework the only trusted entity is the orchestrator, and neither clients nor services need to be such. In particular, the orchestrator infers functional and behavioural types of each service. Also, it is responsible for certifying the service code, for publishing its interface, and for guaranteeing that services will not arbitrarily change their code on the fly: when this happens, services need to be certified again. When an application is injected in the network, the orchestrator provides it with a viable plan (if any), constructed by composing and analysing the certified interfaces of the available services. The trustworthiness of the orchestrator relies upon formal grounds, i.e. the soundness of our type and effect system, and the correctness of the static analysis and model-checking technique that infers viable plans.

As said above, finding viable plans is not a trivial task, because the effect of selecting a given service for a request is not always confined to the execution of that service. Since each service selection may affect the *whole* execution, we cannot simply devise a viable plan by selecting services that satisfy the constraints imposed by the requests, only. We have then devised a two-stage construction for extracting viable plans from a history expression. Let H be the history expression inferred for a client. A first transformation of H, called linearization, lifts all the service choices to the top-level of H. This isolates from H the possible plans, that will be considered one by one in the second stage: model-checking for validity. Projecting the history expression H on a given plan π gives rise to another history expression H', where all the service choices have been resolved according to π. Validity of H' guarantees that the chosen plan π will drive executions that never go wrong at run-time (thus run-time security monitoring becomes unneeded). To verify the validity of H', we first smoothly transform it

into a Basic Process Algebra. We then model-check this Basic Process Algebra with a finite state automaton, specially tailored to recognize validity. The correctness of all these steps (type safety, linearization, model-checking) has been formally proved in [6].

1.3 Contributions

We briefly summarize the key features of our approach.

1. *Taxonomy of security aspects.* We discussed some design choices that affect security in Web Services. These choices address rather general properties of systems: whether services maintain a state across invocations or not, whether they trust each other or not, whether they can pass back and forth mobile code, and whether different threads may share part of their state or not. Each of these choices deeply impacts the expressivity of the enforceable security properties, and the compositionality of planning techniques.
2. *Design Methodology.* We introduced a formal modelling language for designing secure services. Our graphical formalism resembles UML activity diagrams, and it is used to describe the workflow of services. Besides the usual workflow operators, we can express activities subject to security constraints. The awareness of security from the early stages of development will foster security through all the following phases of software production. Diagrams have a formal operational semantics, that specifies the dynamic behaviour of services. Also, they can be statically analysed, to infer the contracts satisfied by a service. Our design methodology allows for a fine-grained characterization of the design choices that affect security (see Section 2). We support our approach with the help of some case study scenarios. The design of UML profiles is currently under development.
3. *Planning and recovering strategies.* We identified several cases where designers need to take a decision before proceeding with the execution. For instance, when a planned service disappears unexpectedly, one can choose to replan, so to adapt to the new network configuration. Depending on the boundary conditions and on past experience, one can choose among different tactics. We comment on the feasibility, advantages and costs of each of them.
4. *Core calculus for services.* We extended the λ-calculus with primitives for selecting and invoking services that respect given security requirements. Service invocation is implemented in a call-by-contract fashion, i.e. you choose a service for its (certified) behaviour, not for its name. Security policies are arbitrary safety properties on execution histories. A key point is that our policies are applied within a given scope, so we called them *local policies.* They are more general than traditional global policies. Instead of having a single, large, monolithic policy, simple requirements on security can be naturally composed. Also, local policies are better than local checks. Programmers are not required to foresee the exact program points where security violations may occur.
5. *Planning secure orchestration.* We defined a three-step static analysis that makes secure orchestration feasible. An abstraction of the program behaviour

is first extracted, through a type and effect system. This abstract behaviour is a history expression that over-approximates the possible run-time histories of all the services involved in an orchestration. The second and third steps put this history expression in a special form, and then model-checks it to construct a correct orchestrator that securely coordinates the running services. Studying the output of the model-checker may highlight design flaws, suggesting how to revise the call-by-contract and the security policies. All the above is completely mechanizable, and we have implemented a prototype to support our methodology. The fact that the tool is based on firm theoretical grounds (i.e. λ^{req} type inference and verifier) positively impacts the reliability to our approach.

The paper is organized as follows. In Section 2 we introduce a taxonomy of security aspects in service-oriented applications. Sections 3, 4 and 5 present our design methodology. In particular, Section 3 introduces our design notation and the operational semantics of diagrams; Section 4 presents service contracts, and outlines how they can be automatically inferred; Section 5 illustrates how to select services under the call-by-contract assumption, and discusses some planning and recovering strategies. A car repair scenario for secure service composition is presented in Section 6. Sections 7, 8 and 9 formally introduce the calculus λ^{req} and the planning machinery. Specifically, Sections 7 formalizes the syntax and the operational semantics of λ^{req} ; Section 8 gives semantics to history expressions, defines a type and effect system for λ^{req} , and states its type safety; Section 9 shows our model-checking technique for planning. We conclude the paper with some remarks (Section 11) about the expected impact of our proposal. Portions of this paper have appeared in [10,6].

2 A Taxonomy of Security Aspects in Web Services

Service composition heavily depends on which information about a service is made public, on how to choose those services that match the user's requirements, and on their actual run-time behaviour. Security makes service composition even harder. Services may be offered by different providers, which only partially trust each other. On the one hand, providers have to guarantee that the delivered service respects a given security policy, in any interaction with the operational environment, and regardless of who actually called the service. On the other hand, clients may want to protect their sensitive data from the services invoked.

In the *history-based* approach to security, the run-time permissions depend on a suitable abstraction of the history of all the pieces of code (possibly partially) executed so far. This approach has been receiving major attention, at both levels of foundations [3,27,43] and of language design/implementation [1,24].

The observations of security-relevant activities, e.g. opening socket connections, reading and writing files, accessing memory critical regions, are called *events*. Sequences of events are called *histories*. The class of policies we are concerned with is that of *safety* properties of histories, i.e. properties that are

expressible through finite state automata. The typical run-time mechanisms for enforcing history-based policies are *reference monitors*, which observe program executions and abort them whenever about to violate the given policy. Reference monitors enforce exactly the class of safety properties [41].

Since histories are the main ingredient of our security model, our taxonomy speaks about how histories are handled and manipulated by services. We focus on the following aspects.

Stateless / Stateful Services

A stateless service does not preserve its state (i.e. its history) across distinct invocations. Instead, a stateful service keeps the histories of all the past invocations. Stateful services allow for more expressive security policies, e.g. they can bound the number of invocations on a per-client basis.

Local / Global Histories

Local histories only record the events generated by a service locally on its site. Instead, a global history may span over multiple services. Local histories are the most prudent choice when services do not trust other services, in particular the histories they generate. In this case, a service only trusts its own history — but it cannot constrain the past history of its callers, e.g. to prevent that its client has visited a malicious site. Global histories instead require some trust relation among services: if a service A trusts B, then the history of A may comprise that of B, and so A may check policies on the behaviour of B.

First Order / Higher Order Requests

A request type $\tau \xrightarrow{\varphi} \tau'$ is first order when both τ and τ' are base types (*Int*, *Bool*, etc.). Instead, if τ or τ' are functional types, the request is higher order. In particular, if the parameter (of type τ) is a function, then the client passes some code to be possibly executed by the requested service. Symmetrically, if τ' is a function type, then the service returns back some code to the caller. Mobility of code impacts the way histories are generated, and demands for particular mechanisms to enforce security on the site where the code is run. A typical protection mechanism is *sandboxing*, that consists in wrapping code within an execution monitor that enforce a given security policy. When there is no mobile code, more efficient mechanisms can be devised, e.g. local checks on security-critical operations.

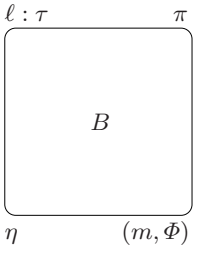
Dependent / Independent Threads

In a network of services, several threads may run concurrently and compete for services. Independent threads keep histories separated, while dependent threads may share part of their histories. Therefore, dependent threads may influence each other when using the same service, while independent threads cannot. For

instance, consider a one-shot service that can be invoked only one time. If threads are independent, the one-shot service has no way to enforce single use. It can only check that no thread uses it more than once, because each thread keeps its own history. Dependent threads are necessary to correctly implement the one-shot service.

3 Designing Secure Services

The basic entity in our design formalism is that of *services*. A service is represented as a box containing its code. The four corners of the box are decorated with information about the service interface and behaviour. The label $\ell : \tau$ indicates the *location* ℓ where the service is made available, and its certified published *interface* τ (discussed later on in Section 4). The other labels instead are used to represent the state of a service at run-time.



$\ell : \tau$	service location ℓ + interface τ
π	orchestration plan
η	event history
(m, Φ)	monitor flag m + sequence Φ of active policies
B	service code

Fig. 1. Execution state of a service

The label $\eta = \alpha_1 \cdots \alpha_k$ is an abstraction of the service execution *history*. In particular, we are concerned with the sequence of security-relevant events α_i happened sometimes in the past, in the spirit of history-based security [1]. The label (m, Φ) is a pair, where the first element is a flag m representing the on/off status of the execution monitor, and the second element is the sequence $\varphi_1 \cdots \varphi_k$ of active *security policies*. When the flag is on, the monitor checks that the service history η adheres to the policy φ_i (written $\eta \models \varphi_i$) for each $i \in 1..k$. Security policies are modelled as regular properties of event histories, i.e. properties that are recognizable by a Finite State Automaton. Since our design notation does not depend on the logic chosen for expressing regular properties of histories, we shall not fix any logic here. However, in our examples (e.g. Fig. 9 in Section 6) we find convenient to describe policies through the *template usage automata* of [7].

The block B inside the box is an abstraction of the service code. Formally, it is a special control flow graph [40] with nodes modelling activities, blocks enclosing sets of nodes, and arrows modelling intra-procedural flow.

Nodes can be of two kinds, i.e. *events* or *requests*. Events $\alpha, \beta, \ldots$ abstract from some security-critical operation. An event can possibly be parametrized,

e.g. $\alpha_w(foo)$ for writing the file *foo*, $sgn(\ell)$ for a certificate signed by ℓ, etc. A service request takes the form $\mathtt{req}_r\tau$. The label r uniquely identifies the request in a network, and the request type τ is defined as:

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

where b is a base type ($Int, Bool, \ldots$). The annotation φ on the arrow is the query pattern (or *"contract"*) to be matched by the invoked service. For instance, the request type $\tau \xrightarrow{\varphi} \tau'$ matches services with functional type $\tau \to \tau'$, and whose behaviour respects the policy φ.

Blocks can be of two kinds: *security blocks* $\varphi[B]$ enforce the policy φ on B, i.e. the history must respect φ at each step of the evaluation of B; *planning blocks* $\{B\}$ construct a plan for the execution of B (see Section 9 for a discussion on some planning strategies). Blocks can be nested, and they determine the scope of policies (hence called *local* policies [5]) and of planning.

The label π is the *plan* used for resolving future service choices. Plans may come in several different shapes [9], but here we focus on a very simple form of plans, mapping each request to a single service. A plan formalises how a call-by-contract $\mathtt{req}_r\tau$ is transformed into a call-by-name, and takes the form of a function from request identifiers r to service locations ℓ. Definition 1 gives the syntax of plans.

Definition 1. Syntax of plans

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

The plan 0 is empty; the plan $r[\ell]$ associates the service published at site ℓ with the request labelled r. The plan $r[?]$ models an unresolved choice for the request r: we call a plan *complete* when it has no unresolved choices. Composition $\mid$ on plans is associative, commutative and idempotent, and its identity is the empty plan 0. We require plans to have a single choice for each request, i.e. $r[\ell] \mid r[\ell']$ implies $\ell = \ell'$.

Note that in this design language, we do not render all the features of λ^{req} (see Section 7). In particular, we neglect variables, conditionals, higher-order functions, and parameter passing. However, we feel free to use these features in the examples, because their treatment can be directly inherited from λ^{req}.

3.1 Graph Semantics

We formally define the behaviour of services through a graph rewriting semantics [4]. In this section, we resort to an *oracle* that provides the initiator of a

computation with a viable plan. The oracle guarantees that the overall execution satisfies all the contracts and the security policies on demand, unless services become unavailable. In the following sections, we will discuss a static machinery that will enable us to correctly implement the oracle, guaranteeing that an expression will never go wrong. We will also show some strategies to adopt when services disappear unexpectedly.

The semantics is defined through graph rewriting. The graph semantics for the case of dependent threads is depicted in Fig. 2 and in Fig. 5. We shall briefly discuss the case of independent threads in Section 3.2. All the remaining axes in the taxonomy are covered by our semantics; in particular, Fig. 3 defines the behaviour of requests and returns according to the possible choices in the taxonomy.

An overlined block $\overline{B}$ means that the first node in B is going to be executed; similarly, an underlined block $\underline{B}$ means that the last node in B has just been executed. A service with a slashed box (rule FAIL) is unavailable, i.e. either is down, unreachable or removed from the directory.

We now briefly discuss the graph rewritings in Fig. 2.

- The evaluation of an event α (rule EV) consists in appending α to the current history. It is also required that the new history obeys all the policies φ in Φ (denoted $\eta\alpha \models \Phi$), if the execution monitor is on.
- The rule SEQ says that, after a block B has been evaluated, the next instruction is chosen non-deterministically among the blocks intra-procedurally connected with B. Note that branching is a special case of SEQ, where the block B is a conditional or a switch.
- Entering a security block $\varphi[B]$ results in appending the policy φ to the sequence of active policies. Leaving $\varphi[B]$ removes φ from the sequence. In both cases, as soon as a history is found not to respect φ, the evaluation gets stuck, to model a security exception (for simplicity, we do not model here exceptions and exception handling. Extending our formalism in this direction would require to define how to compensate from aborted computations, e.g. like in Sagas [28,20]).
- A request $\mathtt{req}_r\tau$ under a plan $r[\ell'] \mid \pi$ looks for the service at site ℓ'. If the service is available (rule REQ), then the client establishes a session with that service (dashed arrow), and waits until it returns. Note that the meaning of the labels η' and Φ' is left undefined in Fig. 2, since it depends on the choice made on the security aspects discussed in Section 2. The actual values for the undefined labels are shown in Fig. 3. In particular, the initial history of the invoked service is: (i) empty, if the service is stateless with local history; (ii) the invoker history, if the service has a global history; (iii) the service past history, if the service is stateful, with local history.
- Returning from a request (rule RET) requires suitably updating the history of the caller service, according to chosen axes in the taxonomy. The actual values for η'' are defined in Fig. 3.

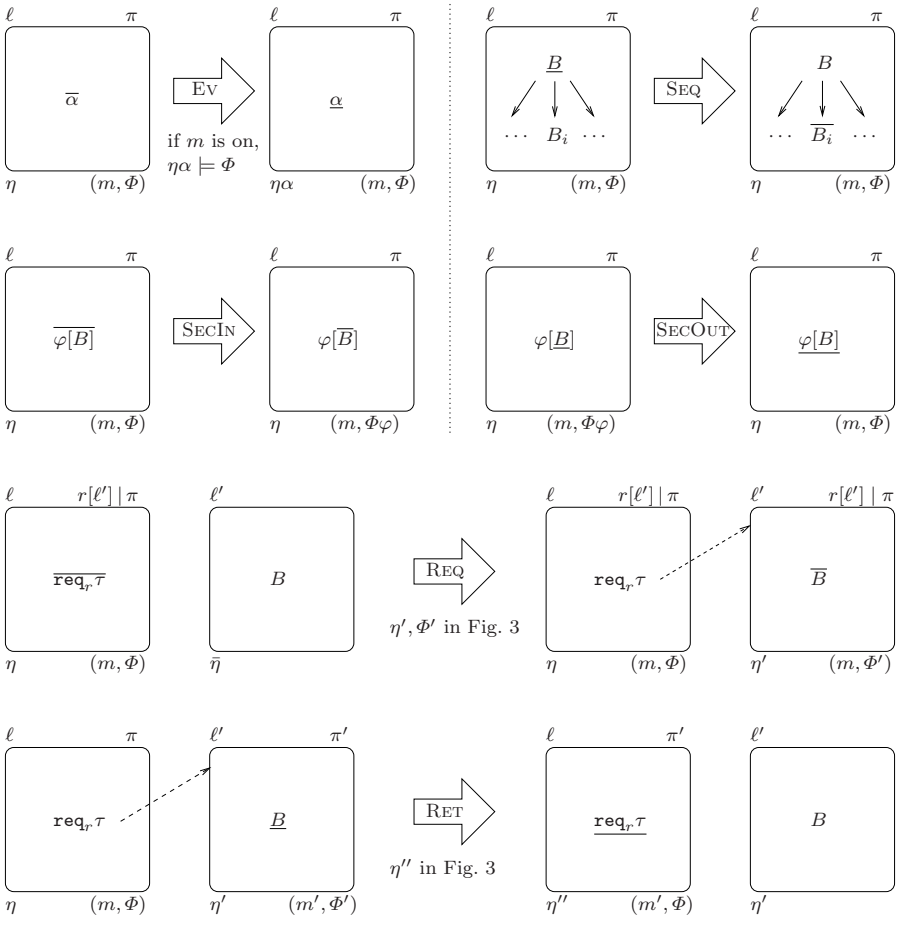


Fig. 2. Semantics of services: events, branches, policies, requests and returns

	Stateless services		Stateful services	
	Local histories	Global histories	Local histories	Global histories
Req	$\eta' = \varepsilon$ $\Phi' = \varepsilon$	$\eta' = \eta$ $\Phi' = \Phi$	$\eta' = \bar{\eta}$ $\Phi' = \varepsilon$	$\eta' = \eta$ $\Phi' = \Phi$
Ret	$\eta'' = \eta$	$\eta'' = \eta$	$\eta'' = \eta$	$\eta'' = \eta'$

Fig. 3. Histories and policies in four cases of the taxonomy

The cases Fail, Plg In and Plg Out are defined in Fig. 5, and they have many possible choices. When no service is available for a request (e.g. because the plan is incomplete, or because the planned service is down), or when you have to construct a plan for a block, the execution may proceed according to one of the strategies discussed in Section 5.

A plan is *viable* when it drives no stuck computations. Under a viable plan, a service can always proceed its execution without attempting to violate some security policy, and it will always manage to resolve each request.

3.2 Semantics of Independent Threads

To model independent threads, each service must keep separate histories of all the initiators. Therefore, histories take the form $\ell_I : \eta, \{\ell_j : \eta_j\}$, where the first item represents the current thread (initiated at site ℓ_I) and its history η, while $\{\ell_j : \eta_j\}$ is the set of the histories associated with the other threads. The rule depicted in Fig. 4 for the case Req shows that (stateful) services must maintain all the histories of the various threads. The actual value of η' is defined as in Fig. 3. The rule Ret is dealt with similarly.

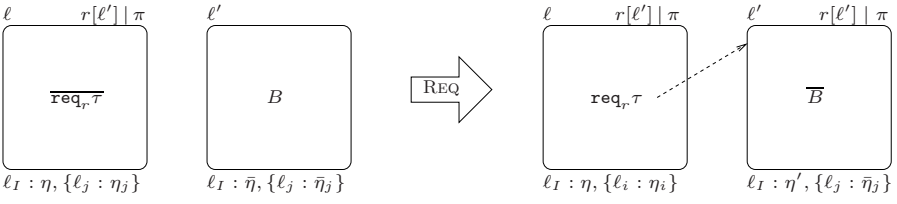


Fig. 4. Maintaining separate histories in the case of independent threads

4 Service Contracts

A service is plugged into a network by publishing it at a site ℓ, together with its interface τ. We assume that each site publishes a single service, and that interfaces are certified, e.g. they are inferred by the type and effect system defined in Section 8. Also, we assume that services cannot invoke each other circularly, since this is quite unusual in the SOC scenario. The functional types are annotated with *history expressions* H that over-approximate the possible run-time histories. When a service with interface $\tau \xrightarrow{H} \tau'$ is run, it will generate one of the histories denoted by H. Note that we overload the symbol τ to range over both service types and request types $\tau \xrightarrow{\varphi} \tau'$. The syntax of types and history expressions is summarized in Definition 2.

History expressions are a sort of context-free grammars. They include the empty history ε, events α, and $H \cdot H'$ that represents sequentialization of code, $H + H'$ for conditionals and branching, security blocks $\varphi[H]$, recursion $\mu h.H$ (where μ binds the occurrences of the variable h in H), localization $\ell : H$, and planned selection $\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$.

A history expression represents a set of histories η, possibly carrying security annotations in the form $\varphi[\eta]$. We formally define the semantics of history expressions in Section 8.1; here we just give some intuition. The semantics of $H \cdot H'$ (denoted by $[\![H \cdot H']\!]$) is the set of histories $\eta\eta'$ such that $\eta \in [\![H]\!]$ and $\eta' \in [\![H']\!]$.

Definition 2. Service interfaces: types and history expressions

τ, τ'	$::=$	types
	b	base type
	$\tau \xrightarrow{H} \tau'$	annotated function
H, H'	$::=$	history expressions
	ε	empty
	h	variable
	α	access event
	$H \cdot H'$	sequence
	$H + H'$	choice
	$\varphi[H]$	security block
	$\mu h.H$	recursion
	$\ell : H$	localization
	$\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$	planned selection

The semantics of $H + H'$ comprises the histories η such that $\eta \in [\![H]\!] \cup [\![H']\!]$. The last three constructs (recursion, localization and planned selection) will benefit of some extra explanation.

- The semantics of a recursion $\mu h.\, H$ is the usual fixed point construction. For instance, the semantics of $\mu h.\, (\gamma + \alpha \cdot h \cdot \beta)$ consists of all the histories $\alpha^n \gamma \beta^n$, for $n \geq 0$ (i.e. γ, $\alpha\gamma\beta$, $\alpha\alpha\gamma\beta\beta, \ldots$).
- The construct $\ell : H$ localizes the behaviour H to the site ℓ. For instance, $\ell : \alpha \cdot (\ell' : \alpha') \cdot \beta$ denotes two histories: $\alpha\beta$ occurring at location ℓ, and α' occurring at location ℓ'.
- A planned selection abstracts from the behaviour of service requests. 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. For instance, the history expression $H = \{r[\ell_1] \triangleright H_1, r[\ell_2] \triangleright H_2\}$ is associated with a request $\mathtt{req}_r \tau$ that can be resolved into either ℓ_1 or 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). If π does not choose either ℓ_1 or ℓ_2, then H denotes no histories.

Typing judgments have the form $H \vdash B : \tau$. This means that the service with code B has type τ, and has execution histories included in the semantics of the effect H. Note that only the initiators of a computation may have $H \neq \varepsilon$; all the other services have typing judgments of the form $\varepsilon \vdash B : b \xrightarrow{H'} b'$. Typing judgments for our diagrams can be directly derived from those of the λ^{req} calculus (see Section 8.3).

In Section 8.4 we will state two fundamental results about our type and effect system. First, it correctly over-approximates the actual run-time histories. Second, it enjoys the following type safety property. We say that an effect H is

valid under a plan π when the histories denoted by H, under the plan π, never violate the security policies in H. Type safety ensures that, if (statically) a service B is well-typed and its effect is valid under a plan π, then (dynamically) the plan π is viable for B, i.e. it only drives safe computations.

In Section 9 we will present a model-checking technique to verify the validity of history expressions, and to extract the viable plans.

5 Service Selection

We now consider the problem of choosing the appropriate service for a block of requests. While one might defer service selection as long as possible, thus only performing it when executing a request, it is usually advantageous to decide how to resolve requests in advance, i.e. to build a plan. This is because "early planning" can provide better guarantees than late service selection. For instance, consider a block with two consecutive requests r_1 and r_2. It might be that, if we choose to resolve r_1 with a particular service ℓ_1, later on we will not be able to find safe choices for r_2. In this case we get stuck, and we must somehow escape from this dead-end, possibly with some expensive compensation (e.g. cancelling a reservation). Early planning, instead, can spot this kind of problem and try to find a better way, typically by considering also r_2 when taking a choice for r_1.

More in detail, when we build a complete viable plan π for a block B, we ensure that B can be securely executed, and we will never get stuck unless a service mentioned in π becomes unavailable. Furthermore, we will need no dynamic checks while executing B, and thus the execution monitor can be kept off, so improving the overall performance. This is a consequence of the type safety result for λ^{req} , formally proved in [6]. When we cannot find a complete viable plan, we could fall back to using an incomplete plan with unresolved requests $r[?]$. In this case, we get a weaker guarantee than the one above, namely that we will not get stuck until an unresolved request must actually be executed.

The rule Pln In in Fig. 5 defines the semantics for constructing plans. The actual values for the labels π', m' and Φ' are defined in Fig. 7. To provide graceful degradation in our model, the Fail rule considers the unfortunate case of executing a request r when either (i) r is still unresolved in the plan, or (ii) r is resolved with an unavailable service. Therefore, we will look for a way to continue the execution, possibly repairing the plan as shown in Fig. 6. The rules Down and Up say that a service may become unavailable, and then available again. We assume that transitions from available to unavailable state (and viceversa) can only happen when a service is not fulfilling a request.

Several strategies for constructing or repairing a plan are possible, and we discuss some of them below. Note that no strategy is always better than the others, since each of them has advantages and disadvantages, as we will point out. The choice of a given strategy depends on many factors, some of which lie outside of our formal model (e.g. availability of services, cost of dynamic checking, etc.).

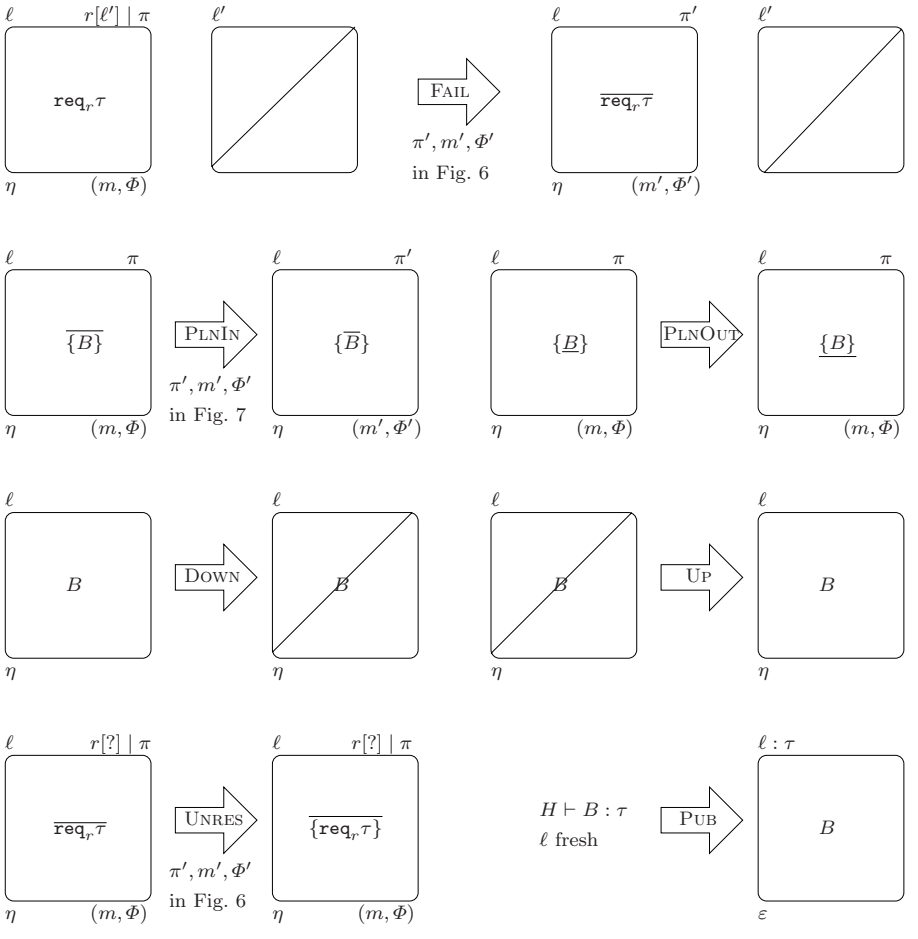


Fig. 5. Semantics of services: failing, planning and becoming up/down

We devise four main classes of strategies:

Greyfriars Bobby.[1] Follow loyally a former plan. If a service becomes unavailable, just wait until it comes back again. This strategy is always safe, although it might obviously block the execution for an arbitrarily long time — possibly forever.

[1] In 1858, a man named John Gray was buried in old Greyfriars Churchyard. For fourteen years the dead man's faithful dog kept constant watch and guard over the grave until his own death in 1872. The famous Skye Terrier, Greyfriars Bobby was so devoted to his master John Gray, even in death, that for fourteen years Bobby lay on the grave only leaving for food.

Patch. Try to reuse as much as possible the current plan. Replace the unavailable services with available ones, possibly newly discovered. The new services must be verified for compatibility with the rest of the plan.

Sandbox. Try to proceed with the execution monitor turned on. The new plan only respects a weak form of compatibility on types ignoring the effect H, but it does not guarantee that contracts and security policies are always respected. Turning on the execution monitor ensures that there will not be security violations, but execution might get stuck later on, because of attempted insecure actions.

Replan. Try to reconstruct the whole plan, possibly exploiting newly discovered services. If a viable plan is found, then you may proceed running with the execution monitor turned off. A complete plan guarantees that contracts and security policies will be always respected, provided than none of the services mentioned in the plan disappear.

In Fig. 6, we describe the effects of these strategies in the context of the Fail rule. There, we also make precise the recovered plan π' and the labels m' and Φ' appearing in the rule. For the "Greyfiars Bobby" strategy, we patiently wait for the service to reappear; on timeout, we will try another strategy. The Patch strategy mends the current plan with a local fix. Note that the Patch strategy is not always safe: in the general case, it is impossible to change just the way to resolve the failing request r and have a new safe plan. We shall return on this issue later on. However, as the figure shows, in some cases this is indeed possible, provided that we check the new choice for resolving r, to ensure the plan is valid again. The Replan strategy is safe when a suitable plan is found, but it could involve statically re-analysing a large portion of the system. When all else fails, it is possible to run a service under a Sandbox, hoping that we will not get stuck.

From now onwards, we use the following abbreviations for the various alternatives described in Section 2: stateless (1) / stateful (ω), local (L) / global (G), first order (F) / higher-order (H), dependent (D) / independent (I). For instance, the case IFL1 in the figure is the one about independent threads, first order requests, local histories, and stateless services.

In Fig. 7 we list the strategies for the rule Pln In, describing how to build a plan for a block B. Note that, when we construct a new plan π' we already have a plan $\pi \mid \pi_B$, where π_B only plans the requests inside B. We can then reuse the available information in π and π_B to build π'. The former plan $\pi \mid \pi_B$ can be non-empty when using nested planning blocks, so reusing parts from it is indeed possible. Since we can reuse the old plan, the strategies are exactly the same of those for the Fail case.

The "Greyfriars Bobby" strategy waits for *all* the services mentioned in the old plan to be available at planning time. This is because it might be wise not to start the block, if we know that we will likely get stuck later. Instead, if some services keep on being unavailable, we should rather consider the other strategies.

Strategy	State update	Case	Condition
Greyfriars Bobby	π Φ	all	The current plan π has a choice for r
Patch	$\pi \mid r[\ell_i]$ Φ	IFL1	$\varphi[H_i]$ is valid
		IFLω	$\varphi[H_i]$ is valid, and $\ell_i \notin \pi$
		IFG1	$\eta\varphi[H_i]$ is valid
		DFL1	$\varphi[H_i]$ is valid
Sandbox	$\pi \mid r[\ell_i]$ $(on, \Phi\varphi)$	all	The service ℓ_i has type $\tau \to \tau'$
Replan	π' $(off, \Phi\varphi)$	all	The new plan π' has a choice for r

Fig. 6. Failure handling strategies for a request $\mathtt{req}_r\tau \xrightarrow{\varphi} \tau'$

As for the Fail rule, the Patch strategy is not always safe, but we can still give some conditions that guarantee the safety of the plan update, which is local to the block B. The Replan strategy, instead, can change the whole plan, even for the requests outside B. If possible, we should always find a complete plan. When this is not the case, we might proceed with some unresolved requests $r[?]$, deferring them to the Fail rule. As a last resort, when no viable plan can be found, or when we deem Replan to be too expensive, we can adopt the Sandbox strategy that turns on the execution monitor.

We now show a situation where the Patch strategy is not safe. We consider the case IFLω case (independent threads, first order requests, local histories, stateful services). The initiator service, in the middle of Fig. 8, performs two requests r_1 and r_2 in sequence. The two requests have the same contract, and thus they can be resolved with the stateful services ℓ_1 and ℓ_2. The service at ℓ_2 performs an event α, within a security block φ. If φ allows only a single α, we

Strategy	State update	Case	Condition
Greyfriars Bobby	$\pi \mid \pi_B$ Φ	all	The plan π_B has a choice for all r_i
Patch	$\pi \mid r_i[\ell_i] \mid \cdots$ Φ	IFL1	$\varphi[H_i]$ is valid, for all i
		IFLω	$\varphi_i[H_i]$ are valid, ℓ_i are distinct, and all $\ell_i \notin \pi$
		IFG1	$\eta_i\varphi_i[H_i]$ are valid, for all i
		DFL1	$\varphi_i[H_i]$ are valid, for all i
Sandbox	$\pi \mid r_i[\ell_i] \mid \cdots$ $(on, \Phi\varphi)$	all	The services ℓ_i have type $\tau_i \to \tau_i'$
Replan	π' $(off, \Phi\varphi)$	all	ηH valid under π' , where η is the current history, and H approximates the future behaviour (may need to refine the analysis)

Fig. 7. Planning strategies for a block B involving requests $\mathtt{req}_{r_i}\tau_i \xrightarrow{\varphi_i} \tau_i'$

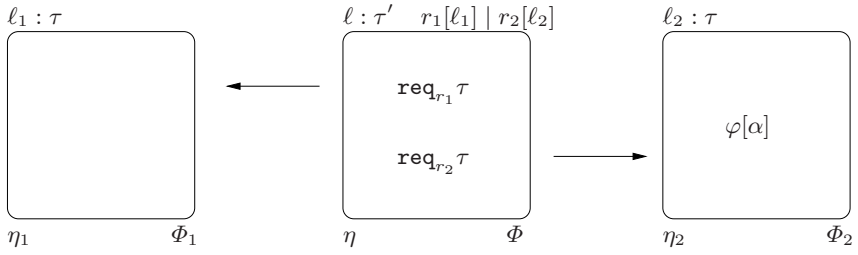


Fig. 8. An unsafe use of the Patch strategy

should be careful and invoke the (stateful) service ℓ_2 at most once. The current plan $\pi = r_1[\ell_1] \mid r_2[\ell_2]$ is safe, since it invokes ℓ_2 exactly once.

Now, consider what happens if the service ℓ_1 becomes unavailable. The FAIL rule is triggered: if we apply Patch and replace the current plan with $r_1[\ell_2] \mid r_2[\ell_2]$, then this patched plan is *not* viable. Indeed, the new plan invokes ℓ_2 twice, so violating φ. The safety condition in Fig. 6 is false, because $\ell_2 \in \pi$: therefore, this dangerous patch is correctly avoided.

6 A Car Repair Scenario

To illustrate some of the features and design facilities made available by our framework, we consider a car repair scenario, where a car may break and then request assistance from a tow-truck and a garage.

In this scenario, we assume a car equipped with a diagnostic system that continuously reports on the status of the vehicle. When the car experiences some major failure (e.g. engine overheating, exhausted battery, flat tyres) the in-car emergency service is invoked to select the appropriate tow-truck and garage services. The selection may take into account some driver personalized policies, and other constraints, e.g. the tow-truck should be close enough to reach both the location where the car is stuck and the chosen garage.

The main focus here is not on the structure of the overall system architecture, rather on how to design the workflow of the service orchestration, taking into account the specific driver policies and the service contracts on demand.

The system is composed of three kinds of services: the CAR-EMERGENCY service, that tries to arrange for a car tow-trucking and repair, the TOW-TRUCK service, that picks the damaged car to a garage, and the GARAGE service, that repairs the car. We assume that all the involved services trust each other's history, and so we assume a shared global history, with independent threads. We also design all the services to be stateful, so that, e.g. the driver can personalize the choice of garages, according to past experiences.

We start by modelling the CAR-EMERGENCY service, i.e. the in-vehicle service that handles the car fault. This service is invoked by the embedded diagnosis system, each time a fault is reported. The actual kind of fault, and the geographic

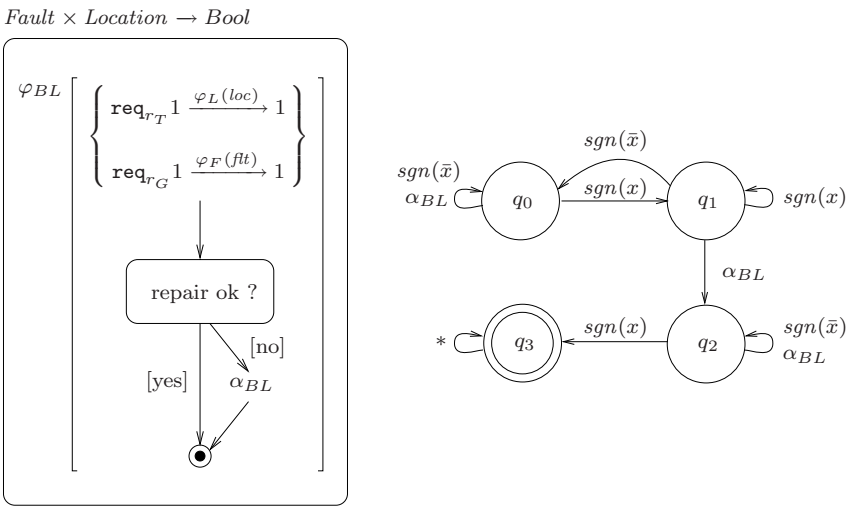


Fig. 9. The CAR-EMERGENCY service and the black-listing policy φ_{BL}

location where the car is stuck, are passed as parameters — named *flt* and *loc*. The diagram of the CAR-EMERGENCY service is displayed on the left-hand side of Fig. 9.

The outer policy φ_{BL} (black-list) has the role of enforcing a sort of "quality of service" constraint. The CAR-EMERGENCY service records in its history the list of all the garages used in past repair requests. When the selected garage ℓ_G completes repairing a car, it appends to the history its own signature $sgn(\ell_G)$. When the user is not satisfied with the quality (or the bill!) of the garage, the garage is black-listed (event α_{BL}). The policy φ_{BL} ensures that a black-listed garage (marked by a signature $sgn(\ell_G)$ followed by a black-listing tag α_{BL}) cannot be selected for future emergencies. The black-listing policy φ_{BL} is formally defined by the *template usage automaton* [7] in Fig. 9, right-hand side. Note that some labels in φ_{BL} are parametric: $sgn(x)$ and $sgn(\bar{x})$ stands respectively for "the signature of garage x" and "a signature of any garage different from x", where x can be replaced by an arbitrary garage identifier. If, starting from the state q_0, a garage signature $sgn(x)$ is immediately followed by a black-listing tag α_{BL}, then you reach the state q_2. From q_2, an attempt to generate again $sgn(x)$ will result in a transition to the non-accepting sink state q_3. For instance, the history $sgn(\ell_1)sgn(\ell_2)\alpha_{BL} \cdots sgn(\ell_2)$ violates the policy φ_{BL}.

The crucial part of the design is the planning block. It contains two requests: r_T (for the tow-truck) and r_G (for the garage), to be planned together. The contract $\varphi_L(loc)$ requires that the tow-truck is able to serve the location *loc* where the car is broken down. The contract $\varphi_F(flt)$ selects the garages that can repair the kind of faults *flt*.

The planning block has the role of determining the orchestration plan for both the requests. In this case, it makes little sense to continue executing with

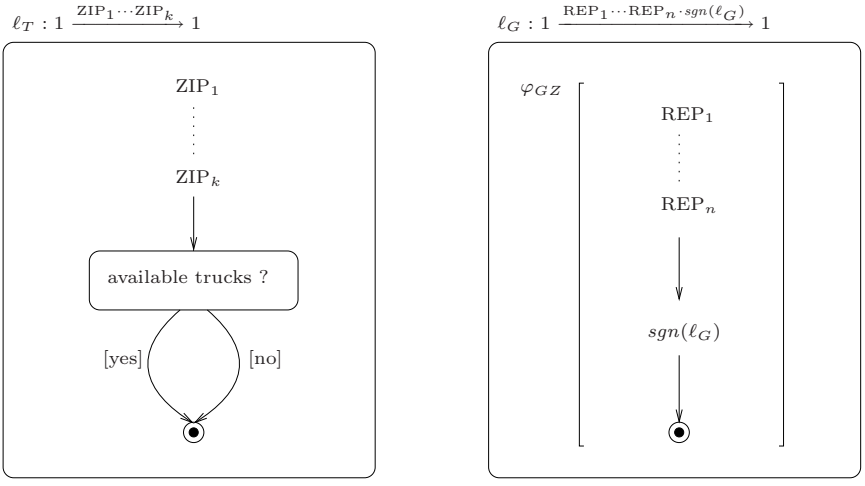


Fig. 10. The TOW-TRUCK (left) and GARAGE (right) services

an incomplete plan or with sandboxing: you should perhaps look for a car rental service, if either the tow-truck or the garage are unavailable. Therefore, a meaningful planning strategy is trying to find a couple of services matching both r_T and r_G, and wait until both the services are available.

The diagram of the TOW-TRUCK service is displayed in Fig. 10, on the left. The service will first expose the list of geographic locations $\text{ZIP}_1, \ldots, \text{ZIP}_k$ it can reach. Each zip code ZIP_i is modelled as an event. The contract $\varphi_T(loc)$ imposed by the CAR-EMERGENCY service ensures that the location *loc* is covered by the truck service. Formally, $\varphi_T(loc)$ checks if the zip code *loc* is contained in the interface of the tow-truck service (we omit the automaton for $\varphi_T(loc)$ here). Then, the TOW-TRUCK may perform some internal activities (irrelevant in our model), possibly invoking other internal services. The exposed interface is of the form $1 \xrightarrow{\text{ZIP}_1 \cdots \text{ZIP}_k} 1$, where 1 is the void type.

The GARAGE service (Fig. 10, right) exposes the kinds of faults $\text{REP}_1, \ldots, \text{REP}_n$ the garage can repair, e.g. tyres, engine, etc. The request contract $\varphi_G(\mathit{flt})$ ensures that the garage can repair the kind of fault *flt* experienced by the car. The GARAGE service may perform some internal bookkeeping activities to handle the request (not shown in the figure), possibly using internal services from its local repository. After the car repair has been completed, the garage ℓ_G signs a receipt, through the event $sgn(\ell_G)$. This signature can be used by the CAR-EMERGENCY service to implement its black-listing policy.

The GARAGE service exploits the policy φ_{GZ} (for Garage-Zip) to ensure that the tow-truck can reach the garage address. If the garage is located in the area identified by ZIP_G, the policy φ_{GZ} checks that the tow-truck has exposed the event ZIP_G among the locations it can reach. When both the contract $\varphi_T(loc)$ and the policy φ_{GZ} are satisfied, we have the guarantee that the tow-truck can pick the car and deposit it at the garage.

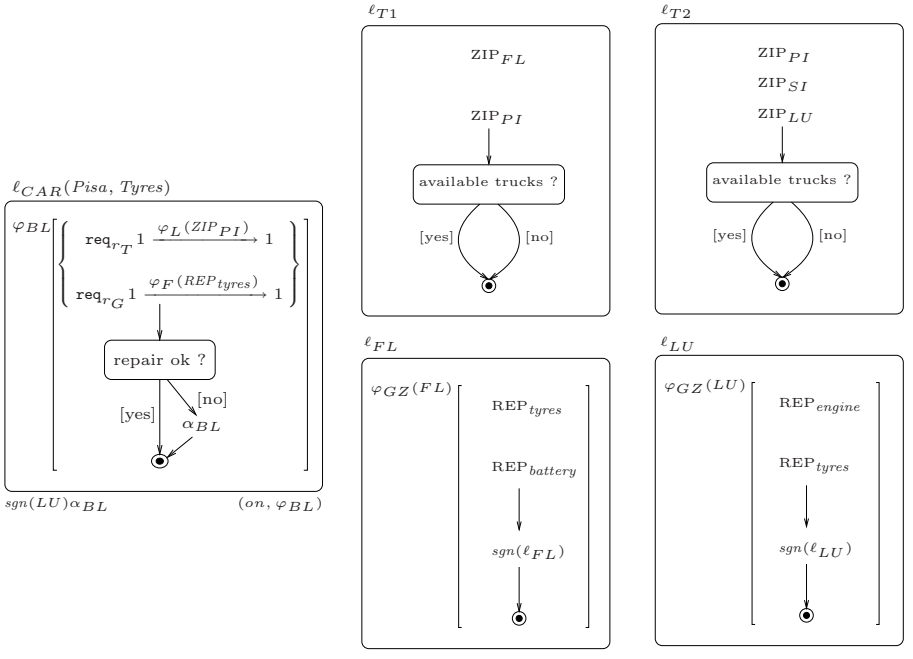


Fig. 11. The CAR-EMERGENCY client (ℓ_{CAR}), two tow-truck services (ℓ_{T1}, ℓ_{T2}), and two garages (ℓ_{FL}, ℓ_{LU})

In Fig. 11, we show a system composed by one car ℓ_{CAR}, two TOW-TRUCK services ℓ_{T1} and ℓ_{T2}, and two GARAGE services ℓ_{FL} and ℓ_{LU}. The car has experienced a flat tyres accident in Pisa (ZIP$_{PI}$), and it has black-listed the garage in Lucca, as recorded in the history $sgn(LU)\,\alpha_{BL}$. The tow-truck service ℓ_{T1} can reach Florence and Pisa, while ℓ_{T2} covers three zones: Pisa, Siena and Lucca. The garage ℓ_{FL} is located in Florence, and it can repair tyres and batteries; the garage ℓ_{LU} is in Lucca, and repairs engines and tyres.

We now discuss all the possible orchestrations:

- the plan $r_T[\ell_{T1}] \mid r_G[\ell_{LU}]$ is not viable, because it violates the policy $\varphi_{GZ}(LU)$. Indeed, the tow-truck can serve Florence and Pisa, but the garage is located in Lucca.
- similarly, the plan $r_T[\ell_{T2}] \mid r_G[\ell_{FL}]$ violates $\varphi_{GZ}(FL)$.
- the plan $r_T[\ell_{T2}] \mid r_G[\ell_{LU}]$ is not viable, because it violates the black-listing policy φ_{BL}. Indeed, it would give rise to a history $sgn(LU)\,\alpha_{BL} \cdots sgn(LU)$, not accepted by the automaton in Fig. 9.
- finally, the plan $r_T[\ell_{T1}] \mid r_G[\ell_{FL}]$ is viable. The tow-truck can reach both the car, located in Pisa, and the garage in Florence, which is not black-listed.

7 A Core Calculus for Services

In this Section we describe λ^{req} , a core calculus for secure service orchestration. The version of λ^{req} we present here has stateless services, local histories, higher-order requests, and independent threads. We first define the syntax of services and the *stand-alone* operational semantics, i.e. the behaviour of a service in isolation. We then define the syntax and operational semantics of networks.

7.1 Services

A service is modelled as an expression in a λ-calculus enriched with primitives for security and service requests. Security-relevant operations (i.e. the events) are rendered as side-effects in the calculus. Roughly speaking, λ^{req} services e implement the specification of blocks B in the graphical notation (Section 3). Note that λ^{req} augments the features of the design language with recursion (instead of loops), parameter passing and higher-order functions.

The abstract syntax of services follows. To enhance readability, our calculus comprises conditional expressions and named abstractions (the variable z in $e' = \lambda_z x.\, e$ stands for e' itself within e, so allowing for explicit recursion). We assume as given the language for guards in conditionals, and we omit its definition here.

Definition 3. Syntax of services

e, e'	::=	x	variable
		α	access event
		$\texttt{if}\, b\, \texttt{then}\, e\, \texttt{else}\, e$	conditional
		$\lambda_z x.\, e$	abstraction
		$e\, e'$	application
		$\varphi[e]$	safety framing
		$\{e\}$	planning
		$\texttt{req}_r \tau$	service request
		$\texttt{wait}\, \ell$	wait reply
		$\mathsf{N/A}$	unavailable

The values v of our calculus are the variables, the abstractions, and the requests. We write $*$ for a distinguished value, and $\lambda.\, e$ for $\lambda x.\, e$, for x not free in e. The following abbreviation is standard: $e; e' = (\lambda.\, e')\, e$. Without loss of generality, we assume that framings include at least one event, possibly dummy.

The stand-alone evaluation of a service is much alike the call-by-value semantics of the λ-calculus; additionally, it enforces all the policies within their framings. Since here services are considered in isolation, the semantics of requests is deferred to Section 7.2. The configurations are triples η, m, e. A transition $\eta, m, e \to \eta', m', e'$ means that, starting from a history η and a monitor flag m, the service e evolves to e', extends η to η', and sets the flag to m'. We assume as given a total function $\mathcal{B}$ that evaluates the guards in conditionals.

Definition 4. Service semantics (stand-alone)

$$\eta, m, (\lambda_z x.\, e)v \to \eta, m, e\{v/x, \lambda_z x.\, e/z\}$$
$$\eta, m, \alpha \to \eta\alpha, m, *$$
$$\eta, m, \mathtt{if}\ b\ \mathtt{then}\ e_{tt}\ \mathtt{else}\ e_{ff} \to \eta, m, e_{\mathcal{B}(b)}$$
$$\eta, m, \mathcal{C}(e) \to \eta', m', \mathcal{C}(e') \quad \text{if } \eta, m, e \to \eta', m', e' \text{ and } m' = \mathit{off} \vee \eta' \models \Phi(\mathcal{C})$$
$$\eta, m, \mathcal{C}(\varphi[v]) \to \eta, m, \mathcal{C}(v) \quad \text{if } m = \mathit{off} \vee \eta \models \varphi$$

where $\mathcal{C}$ is an *evaluation context*, of the following form:

$$\mathcal{C} \ ::= \ \bullet \mid \mathcal{C}\, e \mid v\, \mathcal{C} \mid \varphi[\mathcal{C}]$$

and $\Phi(\mathcal{C})$ is the set of *active policies* of $\mathcal{C}$, defined as follows:

$$\Phi(\mathcal{C}\, e) = \Phi(v\, \mathcal{C}) = \Phi(\mathcal{C}) \qquad \Phi(\varphi[\mathcal{C}]) = \{\varphi\} \cup \Phi(\mathcal{C})$$

The first 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). The form of contexts implies call-by-value evaluation; as usual, functions are not reduced within their bodies. To evaluate a redex enclosed in a set of active policies $\Phi(\mathcal{C})$, the extended history η' must obey each $\varphi \in \Phi(\mathcal{C})$, when the execution monitor is on. A value can leave the scope of a framing φ if the current history satisfies φ. When the monitor is on and the history is found not to respect an active policy φ, the evaluation gets stuck.

7.2 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 in Sect. 8.3. Recall that services cannot invoke each other circularly. A *client* is a special published service $\ell\langle e : \mathit{unit}\rangle$. As we will see, this special form prevents anyone from invoking a client. A *network* is a set of clients and published services.

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

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

where π is the plan used by the current instantiation of the service, η is the history generated so far, m is the monitor flag, 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 $\|$ 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:

$$\ell_1 : \pi_1 \rhd \eta_1, m_1, e'_1 \parallel \cdots \parallel \ell_k : \pi_k \rhd \eta_k, m_k, e'_k$$

abbreviated as $\{\ell_i : \pi_i \rhd \eta_i, m_i, e'_i\}_{i\in 1..k}$. To trigger a computation of the network, we need to 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 $\eta_i = \varepsilon$, $m_i = \mathit{off}$, and $e'_i = *$ if ℓ_i is a service, while $e'_i = e_i$ if ℓ_i is an initiator.

We now discuss the semantic rules of networks in Definition 5. 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. Rule Pub inserts a new service in the network, by publishing its interface τ, certified by the type and effect system. The rules Down/Up make an idle service unavailable/available. 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). Rule Unres triggers the construction of a new plan, in case of an unresolved choice. The rules Pln and Fail exploit the planning/failing strategies to obtain a plan in case of a planned expression $\{e\}$ and of a chosen service which has become unavailable. The actual implementations of the auxiliary functions *plan* and *fail* may vary; in Section 8.5, we shall show a simple case that mixes replan and sandboxing.

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. It is straightforward to derive a truly concurrent semantics from the above one, e.g. using C/E Petri nets.

8 Static Semantics

In this Section we define a static analysis for our core calculus. The analysis takes the form of a type and effect system [29,39,44] where the effects, called

Definition 5. Network semantics

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

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

$$[\textsc{Pub}]\quad N \to N \parallel \ell\langle e : \tau\rangle : 0 \rhd \varepsilon, \mathit{ff}, * \quad \text{if } \ell \text{ fresh and } \vdash_\ell e : \tau$$

$$[\textsc{Down}]\quad \ell\langle e : \tau\rangle : 0 \rhd \varepsilon, m, * \to \ell\langle e : \tau\rangle : 0 \rhd \varepsilon, m, \mathsf{N/A}$$

$$[\textsc{Up}]\quad \ell\langle e : \tau\rangle : 0 \rhd \varepsilon, m, \mathsf{N/A} \to \ell\langle e : \tau\rangle : 0 \rhd \varepsilon, m, *$$

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

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

$$[\textsc{Unres}]\quad \ell : (r[?] \mid \pi) \rhd \eta, m, \mathcal{C}(\mathtt{req}_r \rho\, v) \to \ell : (r[?] \mid \pi) \rhd \eta, m, \mathcal{C}(\{\mathtt{req}_r \rho\, v\})$$

$$[\textsc{Pln}]\quad \ell : \pi \rhd \eta, m, \{e\} \to \ell : \pi' \rhd \eta, m', e \quad \text{if } (\pi', m') = \mathit{plan}(\pi, m, e)$$

$$[\textsc{Fail}]\quad \begin{array}{c}\ell : (r[\ell'] \mid \pi) \rhd \eta, m, \mathcal{C}(\mathtt{req}_r \rho\, v) \parallel \ell'\langle e : \tau\rangle : 0 \rhd \varepsilon, m'', \mathsf{N/A} \to \\ \ell : \pi' \rhd \eta, m', \mathcal{C}(\mathtt{req}_r \rho\, v) \parallel \ell'\langle e : \tau\rangle : 0 \rhd \varepsilon, m'', \mathsf{N/A} \\ \text{if } (\pi', m') = \mathit{fail}(r[\ell'] \mid \pi, m, \mathtt{req}_r \rho)\end{array}$$

history expressions, represent all the possible behaviour of services, while the types extend those of the λ-calculus.

In Section 8.1 we formally give semantics to history expressions, introduced in Section 4. In Section 8.2 we define *validity* of history expressions: roughly, a history expression is valid when the histories it represents do not violate any security constraint. In Section 8.3 we introduce our type and effect system, and in Section 8.4 we establish type safety.

8.1 History Expressions

The *denotational semantics* of a history expression is a set, written $(\ell_i : \mathcal{H}_i)_{i \in I}$. The intended meaning is that the behaviour of the service at location ℓ_i is approximated by the set of histories $\mathcal{H}_i$ (I is a finite set of indexes). Technically, $\mathcal{H}$ belongs to the lifted cpo of sets of histories [47], ordered by (lifted) set inclusion $\subseteq_\perp$ (where $\perp \subseteq_\perp \mathcal{H}$ for all $\mathcal{H}$, and $\mathcal{H} \subseteq_\perp \mathcal{H}'$ whenever $\mathcal{H} \subseteq \mathcal{H}'$). The least upper bound between two elements of the cpo is standard set union $\cup$, assuming that $\perp \cup \mathcal{H} = \mathcal{H}$. The set of events is enriched with *framing events* of the form $[_\varphi$, $]_\varphi$,

Definition 6. Semantics of history expressions

$$\langle\!\langle H \rangle\!\rangle^\pi = \{\, \ell : \{\, \langle\!\langle \eta \rangle\!\rangle \mid \eta \in \mathcal{H} \,\} \mid \ell : \mathcal{H} \in [\![H]\!]^\pi_\emptyset \,\}$$

$$\text{where } \langle\!\langle \eta \rangle\!\rangle = \begin{cases} \eta & \text{if } \sigma \notin \eta \\ \langle\!\langle \eta_0 \rangle\!\rangle \cup \langle\!\langle \eta_1 \rangle\!\rangle & \text{if } \eta = \eta_0\, \sigma\, \eta_1 \end{cases}$$

$$[\![\varepsilon]\!]^\pi_\theta = (? : \{\varepsilon\}) \quad [\![\alpha]\!]^\pi_\theta = (? : \{\alpha\}) \quad [\![\ell : H]\!]^\pi_\theta = [\![H]\!]^\pi_\theta\{\ell/?\}$$

$$[\![\varphi[H]]\!]^\pi_\theta = \varphi[[\![H]\!]^\pi_\theta] \qquad [\![H \cdot H']\!]^\pi_\theta = [\![H]\!]^\pi_\theta \odot [\![H']\!]^\pi_\theta$$

$$[\![h]\!]^\pi_\theta = \theta(h) \qquad [\![H + H']\!]^\pi_\theta = [\![H]\!]^\pi_\theta \oplus [\![H']\!]^\pi_\theta$$

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

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

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

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

that denote the opening and closing of a framing $\varphi[\cdots]$. For example, the history $\eta = \alpha[_\varphi \alpha']_\varphi$ represents a computation that (i) generates an event α, (ii) enters the scope of φ, (iii) generates α' within the scope of φ, and (iv) leaves the scope of φ. Also, a history may end with the *truncation marker* ! (bang). The history $\eta!$ represents a prefix of a possibly non-terminating computation that generates the sequence of events η. We assume that histories are indistinguishable after truncation, i.e. $\eta!$ followed by η' equals to $\eta!$. For notational convenience, we feel free to omit curly braces when writing singleton sets, and we write $\varphi[\mathcal{H}]$ for $\{\, [_\varphi \eta\,]_\varphi \mid \eta \in \mathcal{H} \,\}$.

The *stateless* semantics $\langle\!\langle H \rangle\!\rangle^\pi$ of a closed history expression H depends on the given evaluation plan π, and is defined in two steps. In the first, we define the *stateful* semantics $[\![H]\!]^\pi_\theta$ (in an environment θ binding variables), i.e. a semantics in which services keep track of the histories generated by all the past invocations. A simple transformation then yields $\langle\!\langle H \rangle\!\rangle^\pi$, in which each invocation is instead independent of the previous ones, i.e. it always starts with the empty history.

We first comment on the rules for $[\![H]\!]^\pi_\theta$. The meaning of an event α is the pair $(? : \{\alpha\})$, where ? is dummy and will be bound to the relevant location. The rule for localizing H at ℓ records the actual binding: the current location ℓ replaces "?". The semantics of a sequence $H \cdot H'$ is a suitable concatenation of

Definition 7. Auxiliary operators $\odot$ and $\oplus$

$$\{\ell_i : \mathcal{H}_i\}_I \odot (? : \bot) = (? : \bot) = (? : \bot) \odot \{\ell_i : \mathcal{H}_i\}_I$$

$$\{\ell_i : \mathcal{H}_i\}_I \oplus (? : \bot) = (? : \bot) = (? : \bot) \oplus \{\ell_i : \mathcal{H}_i\}_I$$

$$\{\ell_i : \mathcal{H}_i\}_I \odot \{\ell_j : \mathcal{H}'_j\}_J = \{\ell_i : \mathcal{H}_i\mathcal{H}'_i\}_{I\cap J} \cup \{\ell_i : \mathcal{H}_i\}_{I\setminus J} \cup \{\ell_j : \mathcal{H}'_j\}_{J\setminus I}$$

$$\{\ell_i : \mathcal{H}_i\}_I \oplus \{\ell_j : \mathcal{H}'_j\}_J = \{\ell_i : \mathcal{H}_i \cup \mathcal{H}'_i\}_{I\cap J} \cup \{\ell_i : \mathcal{H}_i \cup \{\varepsilon\}\}_{I\setminus J} \cup \{\ell_j : \mathcal{H}'_j \cup \{\varepsilon\}\}_{J\setminus I}$$

the histories denoted by H and H' site by site (the operator $\odot$ is defined below). Similarly for the semantics of choices $H + H'$, that joins the histories site by site through the operator $\oplus$. The semantics of $\mu h.\, H$ is the least fixed point of the operator f above, computed in the cpo obtained by coalesced sum of the cpos of sets of histories $\mathcal{H}$. The semantics of a planned selection $\{\pi_i \rhd H_i\}_{i\in I}$ under a plan π is the sum of the semantics of those H_i such that π resolves π_i.

The sequentialization $\odot$ of $(\ell_i : \mathcal{H}_i)_{i\in I}$ and $(\ell_j : \mathcal{H}'_j)_{j\in J}$ comprises $\ell_i : \mathcal{H}_i\mathcal{H}'_j$ for all $i = j$ (i.e. $\ell_i : \{\, \eta\eta' \mid \eta \in \mathcal{H}_i, \eta' \in \mathcal{H}'_i \,\}$), and it also comprises $\ell_i : \mathcal{H}_i$ and $\ell_j : \mathcal{H}'_j$ for all $i \notin J$ and $j \notin I$. As an example, $(\ell_0 : \{\alpha_0\}, \ell_1 : \{\alpha_1, \beta_1\}) \odot (\ell_1 : \{\gamma_1\}, \ell_2 : \{\alpha_2\}) = (\ell_0 : \{\alpha_0\}, \ell_1 : \{\alpha_1\gamma_1, \beta_1\gamma_1\}, \ell_2 : \{\alpha_2\})$. The choice operator $\oplus$ is pretty the same, except that union replaces language concatenation. For example, $(\ell_0 : \{\alpha_0\}) \oplus (\ell_0 : \{\beta_0\}, \ell_1 : \{\beta_1\}) = (\ell_0 : \{\alpha_0, \beta_0\}, \ell_1 : \{\beta_1\})$. Note that both $\odot$ and $\oplus$ are strict.

Example 1. Consider the history expression:

$$H = \ell_0 : \alpha_0 \cdot \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1, r[\ell_2] \rhd \ell_2 : \sigma \cdot \alpha_2\} \cdot \beta_0$$

The stateful semantics of H under plan $\pi = r[\ell_1]$ is:

$$\begin{aligned}
&[\![\alpha_0 \cdot \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1, r[\ell_2] \rhd \ell_2 : \sigma \cdot \alpha_2\} \cdot \beta_0]\!]^\pi \{\ell_0/?\} \\
&= \big((? : \{\alpha_0\}) \odot [\![\{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1, r[\ell_2] \rhd \ell_2 : \sigma \cdot \alpha_2\}]\!]^\pi \\
&\qquad\qquad \odot (? : \{\beta_0\})\big)\{\ell_0/?\} \\
&= \big((? : \{\alpha_0\}) \odot [\![\ell_1 : \sigma \cdot \alpha_1]\!]^\pi \odot (? : \{\beta_0\})\big)\{\ell_0/?\} \\
&= \big((? : \{\alpha_0\}) \odot (\ell_1 : \{\sigma\alpha_1\}) \odot (? : \{\beta_0\})\big)\{\ell_0/?\} \\
&= (? : \{\alpha_0\beta_0\},\ \ell_1 : \{\sigma\alpha_1\})\{\ell_0/?\} \\
&= (\ell_0 : \{\alpha_0\beta_0\},\ \ell_1 : \{\sigma\alpha_1\})
\end{aligned}$$

In this case, the stateless semantics just removes the event σ, i.e.:

$$\langle\!\langle H \rangle\!\rangle^\pi \;=\; (\ell_0 : \{\alpha_0\beta_0\},\ \ell_1 : \{\alpha_1\}) \qquad \square$$

Example 2. Consider the history expression:

$$H = \ell_0 : (\mu h.\, \beta_0 + \alpha_0 \cdot \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1\} \cdot h)$$

This represents a service ℓ_0 that recursively generates α_0 and raises a request r (which can be served by ℓ_1 only). Let $\pi = r[\ell_1]$. Then:

$$\begin{aligned}\llbracket H \rrbracket^\pi &= \llbracket \ell_0 : \mu h.\, \beta_0 + \alpha_0 \cdot \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1\} \cdot h \rrbracket^\pi \\ &= \llbracket \mu h.\, \beta_0 + \alpha_0 \cdot \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1\} \cdot h \rrbracket^\pi \{\ell_0/?\} \\ &= \big(\textstyle\bigcup_{n>0} \llbracket f^n(\ell_0 : \{!\}, \ell_1 : \{!\}) \rrbracket^\pi\big)\{\ell_0/?\}\end{aligned}$$

where $f(X) = \llbracket \beta_0 + \alpha_0 \cdot \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1\} \cdot h \rrbracket^\pi_{\{X/h\}}$. The first approximation $f^1(\ell_0 : \{!\}, \ell_1 : \{!\})$ is:

$$\begin{aligned}&\llbracket \beta_0 + \alpha_0 \cdot \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1\} \cdot h \rrbracket^\pi_{\{(\ell_0:\{!\},\ell_1:\{!\})/h\}} \\ &= \llbracket \beta_0 \rrbracket^\pi \oplus (\llbracket \alpha_0 \rrbracket^\pi \odot \llbracket \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1\} \rrbracket^\pi \odot \llbracket h \rrbracket^\pi_{\{(\ell_0:\{!\},\ell_1:\{!\})/h\}}) \\ &= (? : \{\beta_0\}) \oplus \big((? : \alpha_0) \odot (\ell_1 : \{\sigma\alpha_1\}) \odot (\ell_0 : \{!\}, \ell_1 : \{!\})\big) \\ &= (? : \{\beta_0\}) \oplus (? : \{\alpha_0!\}, \ell_1 : \{\sigma\alpha_1!\}) \\ &= (? : \{\beta_0, \alpha_0!\}, \ell_1 : \{\varepsilon, \sigma\alpha_1!\})\end{aligned}$$

The fixed point of f, after the substitution $\{\ell_0/?\}$, is:

$$\begin{aligned}&(\ell_0 : \{\beta_0, \alpha_0!, \alpha_0\beta_0, \alpha_0\alpha_0!, \alpha_0\alpha_0\beta_0, \alpha_0\alpha_0\alpha_0! \ldots\}, \\ &\ell_1 : \{\varepsilon, \sigma\alpha_1, \sigma\alpha_1!, \sigma\alpha_1\sigma\alpha_1, \sigma\alpha_1\sigma\alpha_1!, \ldots\})\end{aligned}$$

The stateless semantics $\langle\!\langle H \rangle\!\rangle^\pi$ is the set:

$$(\ell_0 : \{\beta_0, \alpha_0!, \alpha_0\beta_0, \alpha_0\alpha_0!, \ldots\}, \ell_1 : \{\varepsilon, \alpha_1, \alpha_1!\}) \qquad \square$$

Example 3. Consider the history expression:

$$H = \{r[\ell_0] \rhd \ell_0 : \alpha\} \cdot \{r'[\ell_1] \rhd \beta\}$$

The stateful semantics of H under $\pi = r[\ell_0] \mid r'[\ell_2]$ is:

$$\begin{aligned}\llbracket H \rrbracket^\pi &= \llbracket \{r[\ell_0] \rhd \ell_0 : \alpha\} \rrbracket^\pi \odot \llbracket \{r'[\ell_1] \rhd \beta\} \rrbracket^\pi \\ &= (\ell_0 : \{\alpha\}) \odot (? : \bot) \\ &= (? : \bot)\end{aligned}$$

In this case there are no σ, so the stateless and the stateful semantics coincide. $\square$

8.2 Validity

We now define when histories are valid, i.e. they arise from viable computations that do not violate any security constraint. Consider for instance $\eta_0 = \alpha_w\alpha_r\varphi[\alpha_w]$, where φ requires that no write α_w occurs after a read α_r. Then, η_0 is *not* valid according to our intended meaning, because the rightmost α_w occurs within a framing enforcing φ, and $\alpha_w\alpha_r\alpha_w$ does not obey φ. To be valid, a history η must obey all the policies within their scopes, determined by the framing events in η.

Definition 8. Safe sets and validity

The *safe sets* $S(\eta)$ of a history η are defined as:

$$S(\varepsilon) = \emptyset \qquad S(\eta\,\alpha) = S(\eta) \qquad S(\eta_0\,\varphi[\eta_1]) = S(\eta_0\,\eta_1) \cup \varphi[\eta_0^\flat\,(\eta_1^\flat)^\partial]$$

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

$$\varphi[\mathcal{H}] \in S(\eta) \implies \forall \eta' \in \mathcal{H} \,:\, \eta' \models \varphi$$

A history expression H is π-*valid* when:

$$\langle\!\langle H \rangle\!\rangle^\pi \neq (? : \bot) \;\text{ and }\; \forall \ell \,:\, \forall \eta \in \langle\!\langle H \rangle\!\rangle^\pi @\, \ell \,:\, \models \eta$$

where $(\ell_i : \mathcal{H}_i)_{i\in I} @\, \ell_j \;=\; \mathcal{H}_j$.

We formally define validity through the notion of *safe set*. For example, the safe set of η_0 is $\varphi[\{\alpha_w\alpha_r, \alpha_w\alpha_r\alpha_w\}]$. Intuitively, this means that the scope of the framing $\varphi[\cdots]$ spans over the histories $\alpha_w\alpha_r$ and $\alpha_w\alpha_r\alpha_w$. For each safe set $\varphi[\mathcal{H}]$, validity requires that *all* the histories in $\mathcal{H}$ obey φ.

Some notation is now needed. Let $\eta^\flat$ be the history obtained from η by erasing all the framing events, and let η^∂ be the set of all the prefixes of η, including the empty history ε. For example, if $\eta_0 = \alpha_w\alpha_r\varphi[\alpha_w]$, then $(\eta_0^\flat)^\partial = ((\alpha_w\alpha_r[_\varphi\alpha_w]_\varphi)^\flat)^\partial = (\alpha_w\alpha_r\alpha_w)^\partial = \{\varepsilon, \alpha_w, \alpha_w\alpha_r, \alpha_w\alpha_r\alpha_w\}$. Then, the safe set $S(\eta)$ and validity of histories and of history expressions are defined as in Def. 8.

Note that validity of a history expression is parametric with the given evaluation plan π, and it is defined location-wise on its semantics. If the plan contains unresolved choices for requests mentioned in H, then H is not π-valid, because the operators $\odot$ and $\oplus$ are strict on $\bot$.

Example 4. The safe sets of the history expression $H = \varphi[\alpha_0 \cdot \{r[\ell_1] \triangleright \alpha_1, r[\ell_2] \triangleright \varphi'[\alpha_2]\}] \cdot \alpha_3$, with respect to plans $r[\ell_1]$ and $r[\ell_2]$, are:

$$\begin{aligned} S(\langle\!\langle H \rangle\!\rangle^{r[\ell_1]}) &= S([_\varphi\alpha_0\alpha_1]_\varphi\alpha_3) = \{\; \varphi[\{\varepsilon, \alpha_0, \alpha_0\alpha_1\}] \;\} \\ S(\langle\!\langle H \rangle\!\rangle^{r[\ell_2]}) &= S([_\varphi\alpha_0[_{\varphi'}\alpha_2]_{\varphi'}]_\varphi\alpha_3) \\ &= \{\; \varphi[\{\varepsilon, \alpha_0, \alpha_0\alpha_2\}], \varphi'[\{\alpha_0, \alpha_0\alpha_2\}] \;\} \end{aligned}$$

Let φ require "never α_3", and let φ' require "never α_2". Then, H is $r[\ell_1]$-valid, because the histories ε, α_0, and $\alpha_0\alpha_1$ obey φ. Instead, H is not $r[\ell_2]$-valid, because the history $\alpha_0\alpha_2$ in the safe set $\varphi'[\{\alpha_0, \alpha_0\alpha_2\}]$ does not obey φ'. □

8.3 Type and Effect System

We now introduce a type and effect system for our calculus, building upon [8]. 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.

Definition 9. Types and Type Environments

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

For notational convenience, we assume that the request type ρ in $\texttt{req}_r\rho$ is a special type. E.g. we use $1 \xrightarrow{\varphi[\varepsilon]} (1 \xrightarrow{\varphi'[\varepsilon]} 1)$ for the request type of a service obeying φ and returning a function subject to the policy φ'. Additionally, we put some restrictions on request types. First, only functional types are allowed: this models services being considered as remote procedures (instead, initiators have type 1, so they cannot be invoked). Second, no constraints should be imposed over ρ_0 in a request type $\rho_0 \xrightarrow{\varphi} \rho_1$, i.e. in ρ_0 there are no annotations. This is because the constraints on the selected service should not affect its argument.

A typing judgment $\Gamma, H \vdash e : \tau$ means that the service e evaluates to a value of type τ, and produces a history denoted by the effect H. The auxiliary typing judgment $\Gamma, H \vdash_\ell e : \tau$ is defined as the least relation closed under the rules below, and we write $\Gamma, (\ell : H) \vdash e : \tau$ when the service e at ℓ is typed by $\Gamma, H \vdash_\ell e : \tau$. The effects in the rule for application are concatenated according to the evaluation order of the call-by-value semantics (function, argument, latent effect). The actual effect of an abstraction is the empty history expression, while the latent effect is equal to the actual effect of the function body. The rule for abstraction constraints the premise to equate the actual and latent effects, up to associativity, commutativity, idempotency and zero of $+$, associativity and zero of $\cdot$, α-conversion, and elimination of vacuous μ-binders. The next-to-last rule allows for weakening of effects. Note that our type system does not assign any type to `wait` expressions: indeed, waits are only needed in configurations, and not in service code.

We stipulated that the services provided by the network have certified types. Consequently, the typing relation is parametrized by the set W of services $\ell\langle e : \tau\rangle$ such that $\emptyset, \varepsilon \vdash_\ell e : \tau$. We assume W to be fixed, and we write $\vdash_\ell$ instead of $\vdash_{\ell,W}$. To enforce non-circular service composition, we require W to be partially ordered by $\prec$, where $\ell \prec \ell'$ if ℓ can invoke ℓ'; initiators are obviously the least elements of $\prec$, and they are not related to each other. Note that the up-wards cone of $\prec$ of an initiator represents the (partial) knowledge it has of the network.

Example 5. Consider the following λ^{req} expression:

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

Definition 10. Typing services

$$\frac{\Gamma, H \vdash_\ell e : \tau}{\Gamma, \ell : H \vdash e : \tau} \quad \text{if } e \text{ is published at } \ell$$

$$\Gamma, \varepsilon \vdash_\ell * : 1 \qquad \Gamma, \alpha \vdash_\ell \alpha : 1 \qquad \Gamma, \varepsilon \vdash_\ell x : \Gamma(x)$$

$$\frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash_\ell e : \tau'}{\Gamma, \varepsilon \vdash_\ell \lambda_z x.\, e : \tau \xrightarrow{H} \tau'} \qquad \frac{\Gamma, H \vdash_\ell e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash_\ell e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash_\ell e\, e' : \tau'}$$

$$\frac{\Gamma, H \vdash_\ell e : \tau}{\Gamma, \varphi[H] \vdash_\ell \varphi[e] : \tau} \qquad \frac{\Gamma, H \vdash_\ell e : \tau \quad \Gamma, H \vdash_\ell e' : \tau}{\Gamma, H \vdash_\ell \mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' : \tau} \qquad \frac{\Gamma, H \vdash_\ell e : \tau}{\Gamma, H + H' \vdash_\ell e : \tau}$$

$$\frac{\tau = \uplus\{\, \rho \boxplus_{r[\ell']} \tau' \mid \emptyset, \varepsilon \vdash_{\ell'} e : \tau' \quad \ell \prec \ell' \langle e : \tau' \rangle \quad \rho \approx \tau' \,\}}{\Gamma, \varepsilon \vdash_\ell \mathtt{req}_r \rho : \tau} \qquad \frac{\Gamma, H \vdash_\ell e : \tau}{\Gamma, H \vdash_\ell \{e\} : \tau}$$

Let $\tau = 1$, and $\Gamma = \{z : \tau \xrightarrow{\alpha+\alpha'} \tau; x : \tau\}$. Then, the following typing derivation is possible:

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

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

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

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

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

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

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

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

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

Note in passing that a simple extension of the type inference algorithm of [43] suffices for solving constraints as the one above. □

A service invocation $\mathtt{req}_r\rho$ has an empty actual effect, and a functional type τ, whose latent effect is a planned selection that picks from the network those services known by ℓ and matching the request type ρ.

To give a type to requests, we need some auxiliary technical notation. First we introduce $\approx$, $\boxplus$ and $\Cup$, with the help of a running example. We write $\rho \approx \tau$, and say ρ, τ *compatible*, whenever, omitting the annotations on the arrows, ρ and τ are equal. Formally:

$$\begin{aligned} &1 \approx 1 \\ &(\rho_0 \xrightarrow{\varphi} \rho_1) \approx (\tau_0 \xrightarrow{H} \tau_1) \quad \text{iff } \rho_0 \approx \tau_0 \text{ and } \rho_1 \approx \tau_1 \end{aligned}$$

Example 6. Let $\rho = (\tau \to \tau) \xrightarrow{\varphi} (\tau \to \tau)$, with $\tau = 1$, be the request type in $\mathtt{req}_r\rho$, and consider two services $\ell_i\langle e_i : \tau_i\rangle$ with $\tau_i = (\tau \xrightarrow{h_i} \tau) \xrightarrow{\alpha_i \cdot h_i} (\tau \xrightarrow{\beta_i} \tau)$, for $i \in 1..2$. We have that $\tau_1 \approx \rho \approx \tau_2$, i.e. both the services are compatible with the request r. □

The operator $\boxplus_{r[\ell]}$ combines a request type ρ and a type τ, when they are compatible. Given a request type $\rho = \rho_0 \xrightarrow{\varphi} \rho_1$ and a type $\tau = \tau_0 \xrightarrow{H} \tau_1$, the result of $\rho \boxplus_{r[\ell]} \tau$ is $\tau_0 \xrightarrow{\{r[\ell] \rhd \ell : \varphi[\sigma \cdot H]\}} (\rho_1 \underline{\boxplus}_{r[\ell]} \tau_1)$, where:

$$\begin{aligned} &1 \underline{\boxplus}_{r[\ell]} 1 = 1 \\ &(\rho_0 \xrightarrow{\varphi} \rho_1) \underline{\boxplus}_{r[\ell]} (\tau_0 \xrightarrow{H} \tau_1) = \\ &\qquad (\rho_0 \underline{\boxplus}_{r[\ell]} \tau_0) \xrightarrow{\{r[\ell] \rhd \varphi[H]\}} (\rho_1 \underline{\boxplus}_{r[\ell]} \tau_1) \end{aligned}$$

Example 6 (cont.). The request type ρ is composed with the service types τ_1 and τ_2 as follows:

$$\begin{aligned} \hat{\tau}_1 &= (\tau \xrightarrow{h_1} \tau) \xrightarrow{\{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot h_1]\}} (\tau \xrightarrow{\{r[\ell_1] \rhd \beta_1\}} \tau) \\ \hat{\tau}_2 &= (\tau \xrightarrow{h_2} \tau) \xrightarrow{\{r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot h_2]\}} (\tau \xrightarrow{\{r[\ell_2] \rhd \beta_2\}} \tau) \end{aligned}$$

where $\hat{\tau}_1 = \rho \boxplus_{r[\ell_1]} \tau_1$ and $\hat{\tau}_2 = \rho \boxplus_{r[\ell_2]} \tau_2$. □

The top-level arrow carries a planned selection $\{r[\ell] \rhd \ell : \varphi[\sigma \cdot H]\}$, meaning that, if the service at ℓ is chosen for r, then it generates (at location ℓ, and prefixed by σ) the behaviour H, subject to the policy φ. This top-level choice induces a dependency on the further choices for r recorded in $\rho_1 \underline{\boxplus}_{r[\ell]} \tau_1$. In the example

$$\dfrac{\dfrac{\dfrac{\dfrac{\emptyset, \varepsilon \vdash_{\ell_0} \mathtt{req}_r\rho : \tau' \qquad \emptyset, \varepsilon \vdash_{\ell_0} (\lambda.\gamma) : \tau \xrightarrow{\gamma} \tau}{\emptyset, \{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot \gamma],\ r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot \gamma]\} \vdash_{\ell_0} (\mathtt{req}_r\rho)(\lambda.\gamma) : \tau \xrightarrow{\{r[\ell_1] \rhd \beta_1,\ r[\ell_2] \rhd \beta_2\}} \tau}}{\emptyset, \{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot \gamma],\ r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot \gamma]\} \cdot \{r[\ell_1] \rhd \beta_1,\ r[\ell_2] \rhd \beta_2\} \vdash_{\ell_0} (\mathtt{req}_r\rho)(\lambda.\gamma)* : \tau}}{\emptyset, \ell_0 : \{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot \gamma],\ r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot \gamma]\} \cdot \{r[\ell_1] \rhd \beta_1,\ r[\ell_2] \rhd \beta_2\} \vdash (\mathtt{req}_r\rho)(\lambda.\gamma)* : \tau}$$

Fig. 12. Typing derivation for Example 7

above, the service at ℓ_1 returns a function whose (latent) effect $\{r[\ell] \rhd \beta_1\}$ means that β_1 occurs in the location where the function will be actually applied.

Note that combining functional types never affects the type of the argument. This reflects the intuition that the type of the argument to be passed to the selected service cannot be constrained by the request.

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

$$1 \underline{\Cup} 1 = 1$$
$$(\tau_0 \xrightarrow{H} \tau_1) \underline{\Cup} (\tau'_0 \xrightarrow{H'} \tau'_1) = (\tau_0 \underline{\Cup} \tau'_0) \xrightarrow{H \cup H'} (\tau_1 \underline{\Cup} \tau'_1)$$

Example 6 (cont.). We now unify the combination of the request type ρ with the service types, obtaining:

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

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

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

Example 7. Consider the request and the services of Example 6, and consider the initiator $(\mathtt{req}_r\rho)(\lambda.\gamma)*$ at site ℓ_0. Note that applying any service resulting from the request r to the function $\lambda.\gamma$ yields a new function, which we eventually apply to the value $*$. We have the typing derivation in Fig. 7. The stateful semantics $[\![H]\!]^\pi$ under $\pi = r[\ell_1]$ is:

$$\begin{aligned} &[\![\{r[\ell_1] \rhd \ell_1 : \varphi[\sigma \cdot \alpha_1 \cdot \gamma], r[\ell_2] \rhd \ell_2 : \varphi[\sigma \cdot \alpha_2 \cdot \gamma]\}]\!]^\pi \\ &\quad \odot\ [\![\{r[\ell_1] \rhd \beta_1,\ r[\ell_2] \rhd \beta_2\}]\!]^\pi\{\ell_0/?\} \\ &= (\ell_1 : \{\varphi[\sigma\alpha_1\gamma]\}) \odot (? : \{\beta_1\})\{\ell_0/?\} \\ &= (\ell_1 : \{\varphi[\sigma\alpha_1\gamma]\},\ \ell_0 : \{\beta_1\}) \end{aligned}$$

□

8.4 Type Safety

We now state two central results about our type and effect system. In this section we shall restrict our attention to the case where (i) services never become

unavailable, and (ii) planning is only performed at start-up of execution, i.e. there is no dynamic replanning. Under these assumptions, rules Pln, Fail and Unres are never used. In Section 8.5 we shall come back on this issue.

A plan π is *well-typed* for a service at ℓ, $wt_{@\ell}(\pi)$, when, for each request $\mathtt{req}_r\rho$, the chosen service is compatible with ρ, while respecting the partial order $\prec$:

$$\begin{array}{ll} wt_{@\ell}(0) & \\ wt_{@\ell}(\pi \mid \pi') & \text{if } wt_{@\ell}(\pi) \text{ and } wt_{@\ell}(\pi') \\ wt_{@\ell}(r[\ell']) & \text{if } \ell \prec \ell',\ \ell'\langle e : \tau\rangle \text{ and } \rho \approx \tau \end{array}$$

The next theorem states that our type and effect system correctly over-approximates the actual run-time histories. Consider first a network with a single initiator e at location ℓ_1, and let its computed effect be H, with $\langle\!\langle H \rangle\!\rangle^\pi = (\ell_1 : \mathcal{H}_1, \ldots, \ell_k : \mathcal{H}_k)$ for a given plan π. For each site ℓ_i, the run-time histories occurring therein are prefixes of the histories in $\mathcal{H}_i$ (without framing events). Now, consider a network with $n < k$ initiators at the first n sites, each with its own plan π_j and effect H_j. Since initiators cannot invoke each other, we have $\langle\!\langle H_j \rangle\!\rangle^{\pi_j} = (\ell_1 : \emptyset, ..., \ell_j : \mathcal{H}_j, ..., \ell_n : \emptyset, \ell_{n+1} : \mathcal{H}_{n+1,j}, ..., \ell_k : \mathcal{H}_{k,j})$. For each service ℓ_i, the run-time histories at ℓ_i belong to (the prefixes of) one of the $\mathcal{H}_{i,j}$, with $1 \le j \le n$ (see Ex. 8). As usual, precision is lost when reducing the conditional construct to non-determinism, and when dealing with recursive functions.

Theorem 1. *Let $\{\ell_i\langle e_i : \tau_i\rangle\}_{i\in I}$ be a network, let N_0 be its initial configuration with all π_i well-typed, and let $\emptyset, H_i \vdash e_i : \tau_i$. If $N_0 \to^* \{\ell_i : \pi'_i \rhd \eta_i, e'_i\}_{i\in I}$, then:*

$$\eta_i \in \begin{cases} (\langle\!\langle H_i \rangle\!\rangle^{\pi_i} @\, \ell_i)^\flat\, \partial & \text{if } \ell_i \text{ is an initiator} \\ (\sigma\langle\!\langle H_j \rangle\!\rangle^{\pi_j} @\, \ell_i)^\flat\, \partial & \text{if } \ell_i \text{ is a service,} \\ & \text{for some initiator } \ell_j \end{cases}$$

Example 8. Consider an initiator $e_0 = \alpha_0; (\mathtt{req}_r\rho)*$ at site ℓ_0, with $\rho = 1 \to 1$, and a single service $e_1 = \lambda.\, \alpha_1; \varphi[\mathtt{if}\ b\ \mathtt{then}\ \alpha_2\ \mathtt{else}\ \alpha_3]$ at site ℓ_1, with φ requiring "never α_3". Assume that the guard b always evaluates to true, and that the execution monitor is off (we therefore omit it from configurations). Then, under the plan $\pi_0 = r[\ell_1]$, we have the following computation:

$$\begin{array}{rl} & \ell_0 : \pi_0 \rhd \varepsilon, e_0 \parallel \ell_1 : 0 \rhd \varepsilon, * \\ \to & \ell_0 : \pi_0 \rhd \alpha_0, \mathtt{req}_r\rho * \parallel \ell_1 : 0 \rhd \varepsilon, * \\ \to & \ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\,\ell_1 \parallel \ell_1 : 0 \rhd \sigma, e_1 * \\ \to & \ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\,\ell_1 \parallel \ell_1 : 0 \rhd \sigma\alpha_1, \varphi[\mathtt{if} \cdots] \\ \to & \ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\,\ell_1 \parallel \ell_1 : 0 \rhd \sigma\alpha_1, \varphi[\alpha_2] \\ \to & \ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\,\ell_1 \parallel \ell_1 : 0 \rhd \sigma\alpha_1\alpha_2, \varphi[*] \\ \to & \ell_0 : \pi_0 \rhd \alpha_0, \mathtt{wait}\,\ell_1 \parallel \ell_1 : 0 \rhd \sigma\alpha_1\alpha_2, * \\ \to & \ell_0 : \pi_0 \rhd \alpha_0, * \parallel \ell_1 : 0 \rhd \varepsilon, * \end{array}$$

The history expression H_0 extracted from e_0 is:

$$\ell_0 : \alpha_0 \cdot \{r[\ell_1] \rhd \ell_1 : \sigma \cdot \alpha_1 \cdot \varphi[\alpha_2 + \alpha_3]\}$$

Then, $\langle\!\langle H_0 \rangle\!\rangle^{\pi_0} = (\ell_0 : \{\alpha_0\}, \ell_1 : \{\alpha_1[_\varphi\alpha_2]_\varphi, \alpha_1[_\varphi\alpha_3]_\varphi\})$, and the run-time histories generated at site ℓ_1 are strictly contained in the set $(\sigma\langle\!\langle H_0 \rangle\!\rangle^{\pi_0}@\ell_1)^{\flat\partial} = \{\sigma\alpha_1[_\varphi\alpha_2]_\varphi, \sigma\alpha_1[_\varphi\alpha_3]_\varphi\}^{\flat\partial} = \{\sigma\alpha_1\alpha_2, \sigma\alpha_1\alpha_3\}^{\partial} = \{\varepsilon, \sigma, \sigma\alpha_1, \sigma\alpha_1\alpha_2, \sigma\alpha_1\alpha_3\}$. □

We can now state the type safety property. We say that a plan π is *viable* for e at ℓ when the evolution of e within a network, under plan π, does not go wrong at ℓ. A computation *goes wrong* at ℓ when it reaches a configuration whose state at ℓ is stuck. A state $\ell : \pi \rhd \eta, e$ is *not stuck* if either $e = v$, or $e = (\mathtt{req}_r\rho)v$, or $e = \mathtt{wait}\,\ell'$, or $\ell : \pi \rhd \eta, e \to \ell : \pi \rhd \eta', e'$. Note that we do not consider requests and waits to be stuck. To see why, consider e.g. the network configuration $\ell_1 : r[\ell_2] \rhd \eta_1, (\mathtt{req}_r\rho)v \parallel \ell_2 : \pi \rhd \eta_2, e \parallel \ell_3 : r[\ell_2] \rhd \eta_3, \mathtt{wait}\,\ell_2$. The initiator at ℓ_1 is not stuck, because a fair scheduler will allow it to access the service at ℓ_2, as soon as the initiator at ℓ_3 has obtained a reply.

Theorem 2 (Type Safety). *Let $\{\ell_i\langle e_i : \tau_i\rangle\}_{i\in I}$ be a network of services, and let $\emptyset, H_i \vdash e_i : \tau_i$ for all $i \in I$. If H_i is π_i-valid for π_i well-typed, then π_i is viable for e_i at ℓ_i.*

Example 9. Consider the network in Ex. 6, where we fix $e_i = \lambda x.\,(\alpha_i; (x*); (\lambda.\beta_i))$ for $i \in 1..2$. Assume the constraint φ on the request type ρ is true. Consider now the initiator $e_0 = \varphi_0[(\mathtt{req}_r\rho(\lambda.\gamma))*]$ at ℓ_0, where φ_0 requires "never β_2". Let $\pi = r[\ell_1]$. The history expression H_0 of e_0 (inferred as in Ex. 7) is π-valid. Indeed, $\langle\!\langle H_0 \rangle\!\rangle^{\pi} = (\ell_0 : \{\varphi_0[\beta_1]\}, \ell_1 : \{\varphi[\alpha_1\gamma]\})$, and both $\varphi_0[\beta_1]$ and $\varphi[\alpha_1\gamma]$ are valid. As predicted by Theorem 2, the plan π is viable for e_0 at ℓ_0:

$$\begin{aligned}
&\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{req}_r\rho(\lambda.\gamma))*] \parallel \ell_1 : 0 \rhd \varepsilon, *\\
\to\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{wait}\,\ell_1)*] \parallel \ell_1 : 0 \rhd \sigma, e_1(\lambda.\gamma)\\
\to\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{wait}\,\ell_1)*] \parallel \ell_1 : 0 \rhd \sigma\alpha_1, \gamma; (\lambda.\beta_1)\\
\to\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{wait}\,\ell_1)*] \parallel \ell_1 : 0 \rhd \sigma\alpha_1\gamma, (\lambda.\beta_1)\\
\to\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\lambda.\beta_1)*] \parallel \ell_1 : 0 \rhd \varepsilon, *\\
\to\ &\ell_0 : \pi \rhd \beta_1, \varphi_0[*] \parallel \ell_1 : 0 \rhd \varepsilon, *
\end{aligned}$$

Note that we have not displayed the state of the execution monitor (always off), nor the configurations at site ℓ_2, because irrelevant here. Consider now the plan $\pi' = r[\ell_2]$. Then H_0 is not π'-valid, because $\langle\!\langle H_0 \rangle\!\rangle^{\pi'} = (\ell_0 : \{\varphi_0[\beta_2]\}, \ell_2 : \{\varphi[\alpha_2\gamma]\})$, and the event β_2 violates φ_0. In this case the computation:

$$\begin{aligned}
&\ell_0 : \pi \rhd \varepsilon, \varphi_0[(\mathtt{req}_r\rho(\lambda.\gamma))*] \parallel \ell_2 : 0 \rhd \varepsilon, *\\
\to^*\ &\ell_0 : \pi \rhd \varepsilon, \varphi_0[\beta_2] \parallel \ell_2 : 0 \rhd \varepsilon, *
\end{aligned}$$

is correctly aborted, because $\beta_2 \not\models \varphi_0$. □

8.5 A Planning Strategy

We now focus on the case where services may become unavailable, and planning may be performed at run-time. To do that, we shall complete the definition of the PLN and FAIL rules of the operational semantics of networks, by implementing the functions *plan* and *fail*. To do that, we adopt a planning strategy that mixes the replan and sandboxing strategies introduced in Section 5. To keep our presentation simple, we resort to the Greyfriars Bobby strategy to cope with disappearing services, and so we just wait that the disappeared services become available again. Definition 11 summarizes our planning and recovery strategies.

Consider you want to replan an expression e, when the current plan is π and the current state of the monitor flag is m. Our strategy constructs a new plan π' which is coherent with π on the choices already taken in the past (indeed, modifying past choices could invalidate the viability of the new plan, as shown in Ex. 10). To do that, we first update the global history expression (i.e. that used to compute the starting plan) with all the information about the newly discovered services, possibly discarding the services become unavailable. The result of this step is then analysed with the model-checker of Section 9 in search of viable plans. If a viable plan is found, then it is substituted for the old plan, and the execution proceeds with the execution monitor turned off. If no viable plan is found, the service repository is searched for services that fulfill the "syntactical" requirements of requests, i.e. for each $\mathtt{req}_r\rho$ to replan, the contract type ρ is compatible with the type of the chosen service. The execution then continues with the so-constructed plan, but the monitor is now turned on, because there is no guarantee that the selected services will obey the imposed constraints. If there are no services in the repository that obey this weaker condition, we still try to proceed with a plan with unresolved choices, keeping the execution monitor on and planning "on demand" each future request.

Example 10. Consider the following initiator service at location ℓ_0:

$$\begin{aligned} e_0 \;=\; \varphi[&(\mathtt{let}\ f = \mathtt{req}_r 1 \to (1 \to 1)\ \mathtt{in}\ f\,*);\\ &\{\mathtt{let}\ g = \mathtt{req}_{r'} 1 \to (1 \to 1)\ \mathtt{in}\ g\,*\}] \end{aligned}$$

The service obtains a function f through the first request r, applies it, then it asks for a plan to get a second function g through r' and apply it. The policy φ requires that neither $\alpha\alpha$ nor $\beta\beta$ are performed at ℓ_0. Suppose the network repository consists just of the following two services, located at ℓ_1 and ℓ_2:

$$\ell_1\langle \lambda x.\lambda y.\,\alpha : 1 \to (1 \xrightarrow{\alpha} 1)\rangle \qquad \ell_2\langle \lambda x.\lambda y.\,\beta : 1 \to (1 \xrightarrow{\beta} 1)\rangle$$

The history expression of the initiator service is:

$$H = \ell_0 : \varphi[\{r[\ell_1] \rhd \alpha, r[\ell_2] \rhd \beta\} \cdot \{r'[\ell_1] \rhd \alpha, r'[\ell_2] \rhd \beta\}]$$

Definition 11. Planning and recovering strategies

Let $\bar{\pi}$ be the sub-plan of π containing all the already resolved choices.
Let H be the history expression of the initiator of the computation.
Let L be the set of newly-discovered available services.
Let H_L be the update of H with the information about the services in L.
Let $\pi' = \bar{\pi} \mid \pi''$ be a plan coherent with π on the already resolved choices.

Then:

$$plan(\pi, m, e) = \begin{cases} (\pi', \mathit{off}) & \text{if } H_L \text{ is } \pi'\text{-valid} \\ (\pi', \mathit{on}) & \text{if } \forall \mathtt{req}_r \rho \in e \ : \ r[\ell'] \in \pi' \wedge \ell' : \tau \implies \rho \approx \tau \\ (\bar{\pi} \mid \pi_?, \mathit{on}) & \text{otherwise, where } \pi_? \text{ maps each } r \notin \bar{\pi} \text{ to } ? \end{cases}$$

$$fail(\pi, m, e) = (\pi, m) \qquad \text{(Greyfriars Bobby)}$$

Assume that the execution starts with the viable plan $\pi = r[\ell_1] \mid r'[\ell_2]$, which would generate the history $\alpha\beta$ at ℓ_0, so obeying the policy φ.

$$\begin{aligned} & \ell_0 : \pi \rhd \varepsilon, \mathit{off}, e_0 \parallel \ell_1 \rhd \varepsilon, \lambda x.\lambda y.\, \alpha \parallel \ell_2 \rhd \varepsilon, \lambda x.\lambda y.\, \beta \\ \to^* \ & \ell_0 : \pi \rhd \alpha, \mathit{off}, \{\mathtt{let}\ g = \mathtt{req}_{r'} 1 \to (1 \to 1)\ \mathtt{in}\ g\, *\} \\ & \qquad \parallel \ell_1 \rhd \varepsilon, \lambda x.\lambda y.\, \alpha \parallel \ell_2 \rhd \varepsilon, \lambda x.\lambda y.\, \beta \end{aligned}$$

Just after the function f has been applied, the service at ℓ_2 becomes unavailable:

$$\begin{aligned} \to^* \ & \ell_0 : \pi \rhd \alpha, \mathit{off}, \{\mathtt{let}\ g = \mathtt{req}_{r'} 1 \to (1 \to 1)\ \mathtt{in}\ g\, *\} \\ & \qquad \parallel \ell_1 \rhd \varepsilon, \lambda x.\lambda y.\, \alpha \parallel \ell_2 \rhd \varepsilon, \mathsf{N/A} \end{aligned}$$

Assume now that a new service is discovered at ℓ_3, with type $1 \to (1 \xrightarrow{\beta} 1)$:

$$\begin{aligned} \to^* \ & \ell_0 : \pi \rhd \alpha, \mathit{off}, \{\mathtt{let}\ g = \mathtt{req}_{r'} 1 \to (1 \to 1)\ \mathtt{in}\ g\, *\} \\ & \qquad \parallel \ell_1 \rhd \varepsilon, \lambda x.\lambda y.\, \alpha \parallel \ell_2 \rhd \varepsilon, \mathsf{N/A} \parallel \ell_3 \rhd \varepsilon, \lambda x.\lambda y.\, \beta \end{aligned}$$

The planning strategy in Def. 11 determines that the plan $\pi' = r[\ell_1] \mid r'[\ell_3]$ is viable, and so the execution can safely proceed with π' and with the monitor turned off. Observe that the plan $\pi'' = r[\ell_3] \mid r'[\ell_1]$ is also viable, but it changes the choice already made for the request r. Using π'' instead of π' would lead to a violation of the policy φ, because of the history $\alpha\alpha$ generated at ℓ_0. □

It is possible to extend the Type Safety result of Theorem 2 in the more general case that also the rules Pln, Fail and Unres can be applied. As before, as long as none of the selected services disappear and the initial plan is complete, we have the same static guarantees ensured by Theorem 2: starting from a

viable plan will drive secure computations that never go wrong, so making the execution monitor unneeded. The same property also holds when the dynamic plan obtained through the rule Pln is a complete one, and the monitor is off.

Instead, when the new plan is not complete, we get a weaker property. The execution monitor guarantees that security will never be violated, but now there is no liveness guarantee: the execution may get stuck because of an attempted unsecure action, or because we are unable to find a suitable service for an unresolved request.

In the following section, we shall present a verification technique that extracts from a history expression the plans that make it valid.

9 Planning Secure Service Composition

Once extracted a history expression H from an expression e, we have to analyse H to find if there is any viable plan for the execution of e. This issue is not trivial, because the effect of selecting a given service for a request is not confined to the execution of that service. For instance, the history generated while running a service may later on violate a policy that will become active after the service has returned (see Example 11 below). Since each service selection affects the *whole* execution of a program, we cannot simply devise a viable plan by selecting services that satisfy the constraints imposed by the requests, only.

The first step of our planning technique (Section 9.1) consists then in lifting all the service choices $r[\ell]$ to the top-level of H. This semantic-preserving transformation, called *linearization*, results in effects of the form $\{\pi_1 \rhd H_1 \cdots \pi_n \rhd H_n\}$, where each H_i is free of further planned selection. Its intuitive meaning is that, under the plan π_i, the effect of the overall service composition e is H_i.

The other steps in our technical development allows for mechanically verifying the validity of history expressions that, like the H_i produced by linearization, have no planned selections. Our technique is based on model checking Basic Process Algebras (BPAs) with Finite State Automata (FSA). The standard decision procedure for verifying that a BPA process p satisfies a ω-regular property φ amounts to constructing the pushdown automaton for p and the Büchi automaton for the negation of φ. Then, the property holds if it is empty the (context-free) language accepted by the conjunction of the above, which is still a pushdown automaton. This problem is known to be decidable, and several algorithms and tools show this approach feasible [25].

Recall however that, as it is, our notion of validity is non-regular, because of the arbitrary nesting of framings. As an example, consider again the history expression $H = \mu h.\, \alpha + h \cdot h + \varphi[h]$. The language $[\![H]\!]^\pi$ is context-free and non-regular, because it contains unbounded pairs of balanced $[_\varphi$ and $]_\varphi$. Since context-free languages are not closed under intersection, the emptiness problem is undecidable.

To apply the procedure sketched above, we will first manipulate history expressions in order to make validity a regular property. This transformation, called *regularization*, is defined in Section 9.2, and it preserves validity of history expressions. In Section 9.3 we make history expression amenable to model-checking, by

transforming them into BPAs. In Section 9.4 we construct the FSAs $A_{\varphi_{[]}}$ used for model-checking validity, by suitably transforming the automata A_φ defining security policies.

Summing up, we extract from an expression e a history expression H, we linearize it into $\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$, and if some H_i is valid, then we can deduce that H is π_i-valid. By Theorem 2, the plan π_i safely drives the execution of e, without resorting to any run-time monitor. To verify the validity of an history expressions that, like the H_i above, has no planned selections, we regularize H_i to remove redundant framings, we transform H_i into a BPA $BPA(H_i)$, and we model-check $BPA(H_i)$ with the FSAs $A_{\varphi_{[]}}$.

9.1 Linearization of History Expressions

Example 11. Let $e = (\lambda x.\,(\mathtt{req}_{r_2}\rho_2)x)\,((\mathtt{req}_{r_1}\rho_1)*)$, be an initiator, $\rho_1 = \tau \to (\tau \to \tau)$ and $\rho_2 = (\tau \to \tau) \xrightarrow{\varphi} \tau$, where $\tau = 1$ and φ requires "never γ after β". Intuitively, the service selected upon the request r_1 returns a function, which is then passed as an argument to the service selected upon r_2. Assume the network comprises exactly the following four services:

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

where φ' requires "never β'". Since the request type ρ_1 matches the types of e_{ℓ_1} and $e_{\ell'_1}$, both these services can be selected for the request r_1. Similarly, both e_{ℓ_2} and $e_{\ell'_2}$ can be chosen for r_2. Therefore, we have to consider four possible plans when evaluating the history expression H of e:

$$\begin{aligned} H = \{&r_1[\ell_1] \triangleright \ell_1 : \sigma\cdot\alpha, r_1[\ell'_1] \triangleright \ell'_1 : \sigma\cdot\alpha'\}\cdot \\ &\{r_2[\ell_2] \triangleright \ell_2 : \varphi[\sigma\cdot\{\underline{r_1}[\ell_1] \triangleright \beta, \underline{r_1}[\ell'_1] \triangleright \beta'\}\cdot\gamma], \\ &r_2[\ell'_2] \triangleright \ell'_2 : \varphi[\sigma\cdot\varphi'[\{\underline{r_1}[\ell_1] \triangleright \beta, \underline{r_1}[\ell'_1] \triangleright \beta'\}]]\} \end{aligned}$$

Consider first H under the plan $\pi_1 = r_1[\ell_1] \mid r_2[\ell_2]$, yielding $\langle\!\langle H\rangle\!\rangle^{\pi_1} = (\ell_0 : \emptyset,\ \ell_1 : \{\alpha\},\ \ell_2 : \{\varphi[\beta\gamma]\})$. Then, H is not π_1-valid, because the policy φ is violated at ℓ_2. Consider now $\pi_2 = r_1[\ell'_1] \mid r_2[\ell'_2]$, yielding $\langle\!\langle H\rangle\!\rangle^{\pi_2} = (\ell_0 : \emptyset,\ \ell'_1 : \{\alpha'\},\ \ell_2 : \{\varphi[\varphi'[\beta']]\})$. Then, H is not π_2-valid, because the policy φ' is violated. Instead, the remaining two plans, $r_1[\ell_1] \mid r_2[\ell'_2]$ and $r_1[\ell'_1] \mid r_2[\ell_2]$ are viable for e. □

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

$$\begin{aligned} &\{r_1[\ell_1] \mid r_2[\ell_2] \triangleright \ell_1 : \sigma\cdot\alpha\cdot(\ell_2 : \varphi[\sigma\cdot\beta\cdot\gamma]), \\ &\ r_1[\ell_1] \mid r_2[\ell'_2] \triangleright \ell_1 : \sigma\cdot\alpha\cdot(\ell'_2 : \varphi[\sigma\cdot\varphi'[\beta]]), \\ &\ r_1[\ell'_1] \mid r_2[\ell_2] \triangleright \ell'_1 : \sigma\cdot\alpha'\cdot(\ell_2 : \varphi[\sigma\cdot\beta'\cdot\gamma]), \\ &\ r_1[\ell'_1] \mid r_2[\ell'_2] \triangleright \ell'_1 : \sigma\cdot\alpha'\cdot(\ell'_2 : \varphi[\sigma\cdot\varphi'[\beta']])\} \end{aligned}$$

Formally, we say that H is *equivalent* to H' ($H \equiv H'$ in symbols) when $\langle\!\langle H \rangle\!\rangle^\pi = \langle\!\langle H' \rangle\!\rangle^\pi$, for each plan π. The following properties of $\equiv$ hold.

Theorem 3. *The relation $\equiv$ is a congruence, and it satisfies the following equations between planned selections:*

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

Example 12. Let $H = \mu h.\, \{r[\ell_1] \triangleright \alpha_1,\ r[\ell_2] \triangleright \alpha_2\} \cdot h$. Then, using equations (1), (2) and (6) of Theorem 3, and the identity of the plan 0, we obtain:

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

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

Example 13. Let $H = \{r[\ell_1] \triangleright \alpha_1 \cdot \{r'[\ell'_1] \triangleright \beta_1, r'[\ell'_2] \triangleright \beta_2\},\ r[\ell_2] \triangleright \alpha_2\}$. Applying equations (1), (2) and (7) of Theorem 3, we obtain:

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

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

Given a history expression H, we obtain its linearization in three steps. First, we apply the first equation of Theorem 3 to each event, variable and ε in H. Then, we orient the equations of Theorem 3 from left to right, obtaining a rewriting system that is easily proved finitely terminating and confluent – up to the equational laws of the algebra of plans. The resulting planned selection $H' = \{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$ has no further selections in H_i, but there may be non-independent plans (recall that we discard $\pi_i \triangleright H_i$ when π_i is ill-formed).

In the third linearization step, for each such pairs, we update H' by inserting $\pi_i \rhd H_i + H_j$, and removing $\pi_j \rhd H_j$.

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

Theorem 4. *If $H = \{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$ is linear, and H_i is valid for some $i \in 1..k$, then H is π_i-valid.*

9.2 Regularization of Redundant Framings

History expressions can generate histories with *redundant framings*, i.e. nesting of the same framing. For example, $\eta = \alpha\varphi[\alpha'\varphi'[\varphi[\alpha'']]]$ has an inner redundant framing φ around α''. Since α'' is already under the scope of the outermost φ-framing, it happens that η is valid if and only if $\alpha\varphi[\alpha'\varphi'[\alpha'']]$ is valid. Formally, the S-sets of η comprise $\varphi[\{\alpha, \alpha\alpha', \alpha\alpha'\alpha''\}]$ for the outer framing, and $\varphi[\{\alpha\alpha', \alpha\alpha'\alpha''\}]$ for the inner one. Validity requires that all the histories in $\{\alpha, \alpha\alpha', \alpha\alpha'\alpha''\}$ and $\{\alpha\alpha', \alpha\alpha'\alpha''\}$ obey φ. Since the second set is strictly included in the first one, then the *inner* framing is redundant.

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

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

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

Below, we define a transformation that, given a history expression H, yields a H' that does not generate redundant framings, and H' is valid if and only if H is such. Recall that there is no need for regularizing planned selections, because, by Theorem 4, we will always verify the validity of history expressions with no selections.

Example 14. Let $H = \varphi[\alpha \cdot h_0 \cdot \varphi'[\alpha' + h_1]] \cdot h_2$, and let $\tilde{H} = \varphi[\alpha \cdot h_0 \cdot \varphi'[\alpha' + \bullet]] \cdot h_2$. Then, $H = \tilde{H}\{h_1/\bullet\}$, and so h_1 is guarded by $guard(\tilde{H}) = \{\varphi, \varphi'\}$. Similarly, h_0 is guarded by $\{\varphi\}$, and h_2 is unguarded (i.e. guarded by $\emptyset$). □

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

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

Definition 12. Guards

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

$$\begin{aligned}
\mathit{guard}(H_0 \cdot H_1) &= \mathit{guard}(H_i) \quad \text{if } \bullet \in H_i \\
\mathit{guard}(H_0 + H_1) &= \mathit{guard}(H_i) \quad \text{if } \bullet \in H_i \\
\mathit{guard}(\varphi[H]) &= \{\varphi\} \cup \mathit{guard}(H) \\
\mathit{guard}(\mu h.\, H) &= \mathit{guard}(H)
\end{aligned}$$

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

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

Example 15. Consider the history expression $H_0 = \mu h.\, H$, where $H = \alpha + h \cdot h + \varphi[h]$. Then, H can be written as $H'\{h/h_i\}_{i\in 0..2}$, where $H' = \alpha + h_0 \cdot h_1 + \varphi[h_2]$. Since $\mathit{guard}(H'\{\bullet/h_2\}) = \mathit{guard}(\alpha + h_0 \cdot h_1 + \varphi[\bullet]) = \{\varphi\} \not\subseteq \emptyset$, then:

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

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

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

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

Example 16. Let $H_0 = \mu h.\, H_1$, where $H_1 = \mu h'.\, H_2$, and $H_2 = \alpha + h \cdot \varphi[h']$. Since $\mathit{guard}(H_1\{\bullet/h\}) = \emptyset$, we have that:

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

Definition 13. Regularization of framings

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

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

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

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

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

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

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

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

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

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

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

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

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

Summing up, we have that:

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

□

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

Theorem 5. *For all history expressions H:*

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

9.3 From History Expressions to Basic Process Algebras

Basic Process Algebras [11] (BPAs) provide a natural characterization of history expressions. BPA processes contain the terminated process 0, events α, that may stand for an access or framing event, the operators $\cdot$ and $+$ that denote sequential composition and (non-deterministic) choice, and variables X. To allow for recursion, a BPA is then defined as a process p and a set of definitions Δ for the variables X that occur therein.

Definition 14. Syntax of BPA processes

$$p, p' \quad ::= \quad 0 \mid \alpha \mid p \cdot p' \mid p + p' \mid X$$

Definition 15. LTS for Basic Process Algebras

The semantics $[\![P]\!]$ of a BPA $P = (p_0, \Delta)$ is the set of the histories labelling finite computations:

$$\{\, \eta = a_1 \cdots a_i \mid p_0 \xrightarrow{a_1} \cdots \xrightarrow{a_i} p_i \,\}$$

where $a \in \mathsf{Ev} \cup \{\varepsilon\}$, and the relation $\xrightarrow{a}$ is inductively defined as:

$$0 \cdot p \xrightarrow{\varepsilon} p \qquad \alpha \xrightarrow{\alpha} 0 \qquad p + q \xrightarrow{\varepsilon} p \qquad p + q \xrightarrow{\varepsilon} q$$

$$\frac{p \xrightarrow{a} p'}{p \cdot q \xrightarrow{a} p' \cdot q} \qquad X \xrightarrow{\varepsilon} p \quad \text{if } X \triangleq p \in \Delta$$

We assume a finite set $\Delta = \{X \triangleq p\}$ of definitions, such that, for each variable X, $X \triangleq p \in \Delta$ and $X \triangleq p' \in \Delta$ imply $p = p'$. The operational semantics of BPAs is in Def. 15.

We now introduce a mapping from history expressions to BPAs, in the line of [5,43]. Again, note that there is no need for transforming planned selections into BPAs, because we are only interested in the validity of history expressions with no selections.

Definition 16. Mapping history expressions to BPAs

$$
\begin{aligned}
BPA(\varepsilon, \Theta) &= (0, \emptyset) \\
BPA(\alpha, \Theta) &= (\alpha, \emptyset) \\
BPA(h, \Theta) &= (\Theta(h), \emptyset) \\
BPA(H_0 \cdot H_1, \Theta) &= (p_0 \cdot p_1, \Delta_0 \cup \Delta_1), \text{where } BPA(H_i, \Theta) = (p_i, \Delta_i) \\
BPA(H_0 + H_1, \Theta) &= (p_0 + p_1, \Delta_0 \cup \Delta_1), \text{where } BPA(H_i, \Theta) = (p_i, \Delta_i) \\
BPA(\varphi[H], \Theta) &= ([_\varphi \cdot p \cdot]_\varphi, \Delta), \text{where } BPA(H, \Theta) = (p, \Delta) \\
BPA(\mu h.H, \Theta) &= (X, \Delta \cup \{X \triangleq p\}), \text{where } BPA(H, \Theta\{X/h\}) = (p, \Delta)
\end{aligned}
$$

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

The rules that transform history expressions into BPAs are rather natural. Events, variables, concatenation and choice are mapped into the corresponding BPA counterparts. A history expression $\varphi[H]$ is mapped to the BPA for H, surrounded by the opening and closing of the φ-framing. A history expression $\mu h.H$ is mapped to a fresh BPA variable X, bound to the translation of H in the set of definitions Δ.

We now state the correspondence between history expressions and BPAs. The semantics of $BPA(H)$ comprises all and only the prefixes of the histories generated by H (i.e. $[\![H]\!]^{0\,\partial}$).

Theorem 6. *For all history expressions H with no planned selection:*

$$\left([\![H]\!]^0\right)^\partial \;=\; [\![BPA(H)]\!]$$

9.4 Model-Checking Validity

Given a policy φ, we are interested in defining a FSA $A_{\varphi_{[\,]}}$ to be used in verifying the validity of a history η with respect to security policies within their framings. Since our histories are always finite and our properties are regular, FSA suffice.

The automaton $A_{\varphi_{[\,]}}$ is partitioned into two layers. The first layer is a copy of A_φ, where all the states are final. This models the fact that we are outside the scope of φ, i.e. the history leading to any state in this layer has balanced framings of φ (or none). The second layer is reachable from the first one when opening a framing for φ, while closing the framing gets back. The transitions in the second layer are a copy of those connecting accepting states in A_φ. Consequently, the states in the second layer are exactly the final states in A_φ. Since $A_{\varphi_{[\,]}}$ is only concerned with the verification of φ, the transitions for opening and closing framings $\varphi' \neq \varphi$ are rendered as self-loops.

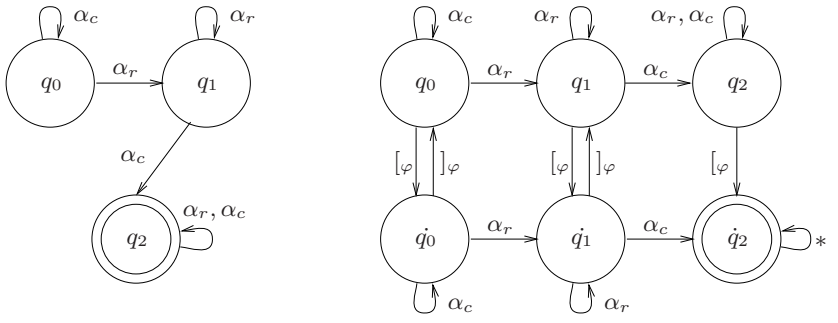


Fig. 13. Finite state automata A_φ (left) and $A_{\varphi_{[]}}$ (right)

Definition 17. Finite state automaton for $\varphi_{[]}$

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

For all histories η, we write $\eta \models \varphi_{[]}$ when η is accepted by $A_{\varphi_{[]}}$.

Example 17. Consider the policy φ saying that no event α_c can occur after an α_r. The FSA A_φ and $A_{\varphi_{[]}}$ are shown in Fig. 13, where the doubly-circled states are the offending ones (i.e. those modelling violation of the policy). It is immediate checking that the history $[_\varphi \alpha_r]_\varphi \alpha_c$ obeys the policy represented by $A_{\varphi_{[]}}$, while $\alpha_r [_\varphi \alpha_c]_\varphi$ does not. □

We require that the history η to be verified against $A_{\varphi_{[]}}$ has no redundant framings, i.e. η has been regularized. Hereafter, let the formula φ be defined by the FSA $A_\varphi = (\Sigma, Q, q_0, \rho, F)$, which we assume to have a distinguished non-final sink state. The FSA $A_{\varphi_{[]}}$ is constructed as in Def. 17.

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

Theorem 7. *Let η be a history with no redundant framings. Then, η is valid if and only if $\eta \models \varphi_{[]}$ for all φ occurring in η.*

Since finite state automata are closed under intersection, a valid history η is accepted by the intersection of the automata $A_{\varphi[]}$ for all φ in η. Validity of a closed history expression H with no planned selections can be decided by showing that the BPA generated by the regularization of H satisfies the given regular formula. Together with Theorem 2, the execution of an expression in our calculus never violates security if its effect is verified valid. Thus we are dispensed from using an execution monitor to enforce the security properties.

Theorem 8. *Let H be a history expression H with no planned selections. Then, H is valid if and only if:*

$$[\![BPA(H\downarrow)]\!] \models \bigwedge_{\varphi \in H} \varphi_{[]}$$

10 Related Work

Process calculi techniques have been used to study the foundation of services. The main goal of some of these proposals, e.g. [26,18,32,35] is to formalise various aspects of standards for the description, execution and orchestration of services (WSDL, SOAP and WS-BPEL). The Global Calculus [21] addresses the problem of relating orchestration and choreography. As a matter of fact, our λ^{req} builds over the standard service infrastructure the above calculi formalise. Indeed, our call-by-contract supersedes standard invocation mechanisms and allows for verified planning.

The secure composition of components has been the main concern underlying the design of Sewell and Vitek's box-π [42], an extension of the π-calculus that allows for expressing safety policies in the form of *security wrappers*. These are programs that encapsulate a component to control the interactions with other (possibly untrusted) components. The calculus is equipped with a type system that statically captures the allowed causal information flows between components. Our safety framings are closely related to wrappers, but in [42] there is no analog of our liveness framings.

Gorla, Hennessy and Sassone [31] consider a calculus for mobile agents which may migrate between sites in a controlled manner. Each site has a *membrane*, representing both a security policy and a classification of external sites with respect to their levels of trust. A membrane guards the incoming agents before allowing them to execute. Three classes of membranes are studied, the most complex being the class of policies enforceable by finite state automata. When an agent comes from an untrusted site, *all* its code must be checked. Instead, an agent coming from a trusted site must only provide the destination site with a *digest* of its behaviour, so allowing for more efficient checks.

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

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

From a technical point of view, the work of Skalka and Smith [43] is the closest to this paper. We share with them the use of a type and effect system and that of model checking validity of effects. In [43], a static approach to history-based access control is proposed. The λ-calculus is enriched with access events and local checks on the past event history. Local checks make validity a regular property, so regularization is unneeded. The programming model and the type system of [43] allow for access events parametrized by constants, and for let-polymorphism. We have omitted these features for simplicity, but they can be easily recovered by using similar techniques.

A related line of research addresses the issue of modelling and analysing resource usage. Igarashi and Kobayashi [34] introduce a type systems to check whether a program accesses resources according to a user-defined usage policy. Our model is less general than the framework of [34], but we provide a static verification technique, while [34] does not. Colcombet and Fradet [22] and Marriot, Stuckey and Sulzmann [37] mix static and dynamic techniques to transform programs in order to make them obey a given safety property. Besson, de Grenier de Latour and Jensen [12] tackle the problem of characterizing when a program can call a stack-inspecting method while respecting a global security policy. Compared to [22,37,12], our programming model allows for local policies, while the other only considers global ones.

Recently, increasing attention has been devoted to express service contracts as behavioural (or session) types. These synthetise the essential aspects of the interaction behaviour of services, while allowing for efficient static verification of properties of composed systems. Session types [33] have been exploited to formalize compatibility of components [46] and to describe adaptation of web services [19]. Security issues have been recently considered in terms of session types, e.g. in [17], which proves the decidability of type-checking in an extension of the π-calculus with session types and correspondence assertions [48]. Our λ^{req} has no explicit primitive for sessions. However, they can be suitably encoded, via higher-order functions.

Other papers have proposed type-based methodologies to check security properties of distributed systems. For instance, Gordon and Jeffrey [30] use a type and effect system to prove authenticity properties of security protocols. Web service authentication has been recently modelled and analysed in [13,14] through a process calculus enriched with cryptographic primitives. In particular, [15] builds security libraries using the WS-Security policies of [2]. These libraries are then mechanically analysed with ProVerif [16].

11 Conclusions

We have described a formal framework for designing and implementing secure service-oriented applications. The main features of our framework are its

security-awareness, a call-by-contract service invocation, a formal semantics, and a system verification machinery, as well as a graphical modelling language. All the above items contribute to achieving static guarantees about planning, and graceful degradation when services disappear.

The formal foundation of our work is λ^{req}, a core calculus for services with primitives to express non-functional constraints on service composition. We focussed here on security properties, in particular on those expressible as safety properties of service execution histories. In other papers [8,6], we have also explored liveness properties.

We have then defined a type and effect system to safely approximate the runtime behaviour of services. These approximations are called *history expressions*. They are a sort of context-free grammars with special constructs to describe the (localized) histories produced by executing services, and the selection of services upon requests.

Analysing these approximations allowed us to single out the *viable plans* that drive secure service composition, i.e. those that achieve the task assigned while respecting all the security constraints on demand. To do that, we exploited model checking over Basic Process Algebras and Finite State Automata. This verification step required some pre-processing on history expressions: technically, we linearized and regularized them to expose the possible plans and to make model checking feasible.

As a further contribution, we proposed a graphical modelling language for services, supporting most of the features of λ^{req}. This calculus has a formal operational semantics in the form of a graph rewriting system. Services described in the graphical model can be naturally refined into more concrete λ^{req} programs. This can be done with the help of simple model transformation tools. One can then reuse all the λ^{req} tools, including its static machinery, and therefore rapidly build a working prototype of a service-based system.

As usual, a prototype can help in the design phase, because one can perform early tests on the system, e.g. by providing as input selected data, one can observe whether the outputs are indeed the intended ones. The call-by-contract mechanism makes this standard testing practice even more effective, e.g. one can perform a request with a given policy φ and observe the resulting plans. The system must then consider *all* the services that satisfy φ, and the observed effect is similar to running a *class* of tests. For instance, a designer of an online bookshop can specify a policy such as "order a book without paying" and then inspect the generated plans: the presence of viable plans could point out an unwanted behaviour, e.g. due to an unpredicted interaction between different special offers. As a matter of facts, standard testing techniques are yet not sophisticated enough to spot such kind of bugs. Thus, a designer may find the λ^{req} prototype useful to check the system, since unintended plans provide him with a clear description of the unwanted interactions between services.

Acknowledgments

This research has been partially supported by EU-FETPI Global Computing Project IST-2005-16004 Sensoria (Software Engineering for Service-Oriented Overlay Computers).

References

1. Abadi, M., Fournet, C.: Access control based on execution history. In: Proc. 10th Annual Network and Distributed System Security Symposium (2003)
2. Atkinson, B., et al.: Web Services Security (WS-Security) (2002), http://www.oasis-open.org
3. Banerjee, A., Naumann, D.A.: History-based access control and secure information flow. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, Springer, Heidelberg (2005)
4. Barendregt, H.P., et al.: Term graph rewriting. In: Parallel Languages on PARLE: Parallel Architectures and Languages Europe (1987)
5. Bartoletti, M., Degano, P., Ferrari, G.L.: History based access control with local policies. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, Springer, Heidelberg (2005)
6. Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and verifying service composition. Technical Report TR-07-02, Dip. Informatica, Univ. of Pisa. (to appear in Journal of Computer Security, 2007), http://compass2.di.unipi.it/TR/Files/TR-07-02.pdf.gz
7. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Types and effects for resource usage analysis. In: Proc. Foundations of Software Science and Computation Structures (Fossacs) (to appear, 2007)
8. Bartoletti, M., Degano, P., Ferrari, G.L.: Enforcing secure service composition. In: Proc. 18th Computer Security Foundations Workshop (CSFW) (2005)
9. Bartoletti, M., Degano, P., Ferrari, G.L.: Plans for service composition. In: Workshop on Issues in the Theory of Security (WITS) (2006)
10. Bartoletti, M., Degano, P., Ferrari, G.L.: Types and effects for secure service orchestration. In: Proc. 19th Computer Security Foundations Workshop (CSFW) (2006)
11. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theoretical Computer Science, 37 (1985)
12. Besson, F., de Grenier de Latour, T., Jensen, T.: Interfaces for stack inspection. Journal of Functional Programming 15(2) (2005)
13. Bhargavan, K., Corin, R., Fournet, C., Gordon, A.D.: Secure sessions for web services. In: Proc. ACM Workshop on Secure Web Services (2004)
14. Bhargavan, K., Fournet, C., Gordon, A.D.: A semantics for web services authentication. In: Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2004)
15. Bhargavan, K., Fournet, C., Gordon, A.D.: Verified reference implementations of WS-security protocols. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, Springer, Heidelberg (2006)
16. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Computer Security Foundations Workshop (CSFW) (2001)

17. Bonelli, E., Compagnoni, A., Gunter, E.: Typechecking safe process synchronization. In: Proc. Foundations of Global Ubiquitous Computing. ENTCS, vol. 138(1) (2005)
18. Boreale, M., et al.: SCC: a service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, Springer, Heidelberg (2006)
19. Brogi, A., Canal, C., Pimentel, E.: Behavioural types and component adaptation. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, Springer, Heidelberg (2004)
20. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Proc. of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL) (2005)
21. Carbone, M., Honda, K., Yoshida, N.: Structured global programming for communicating behaviour. In: European Symposium in Programming Languages (ESOP) (to appear, 2007)
22. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2000)
23. Van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases 14(1) (2003)
24. Edjlali, G., Acharya, A., Chaudhary, V.: History-based access control for mobile code. In: Vitek, J. (ed.) Secure Internet Programming. LNCS, vol. 1603, Springer, Heidelberg (1999)
25. Esparza, J.: On the decidability of model checking for several μ-calculi and Petri nets. In: Tison, S. (ed.) CAAP 1994. LNCS, vol. 787, Springer, Heidelberg (1994)
26. Ferrari, G.L., Guanciale, R., Strollo, D.: JSCL: A middleware for service coordination. In: Najm, E., Pradat-Peyre, J.F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, Springer, Heidelberg (2006)
27. Fong, P.W.: Access control by tracking shallow execution history. In: IEEE Symposium on Security and Privacy (2004)
28. Garcia-Molina, H., Salem, K.: Sagas. In: Proc. ACM SIGMOD, ACM Press, New York (1987)
29. Gifford, D.K., Lucassen, J.M.: Integrating functional and imperative programming. In: ACM Conference on LISP and Functional Programming (1986)
30. Gordon, A., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. In: Proc. IEEE Computer Security Foundations Workshop (CSFW) (2002)
31. Gorla, D., Hennessy, M., Sassone, V.: Security policies as membranes in systems for global computing. Logical Methods in Computer Science 1(3) (2005)
32. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: A calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, Springer, Heidelberg (2006)
33. Honda, K., Vansconcelos, V., Kubo, M.: Language primitives and type discipline for structures communication-based programming. In: Hankin, C. (ed.) ESOP 1998 and ETAPS 1998. LNCS, vol. 1381, Springer, Heidelberg (1998)
34. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2002)
35. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: European Symposium in Programming Languages (ESOP) (to appear, 2007)
36. Lazovik, A., Aiello, M., Gennari, R.: Encoding requests to web service compositions as constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, Springer, Heidelberg (2005)

37. Marriott, K., Stuckey, P.J., Sulzmann, M.: Resource usage verification. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, Springer, Heidelberg (2003)
38. Misra, J.: A programming model for the orchestration of web services. In: 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004) (2004)
39. Nielson, F., Nielson, H.R.: Type and effect systems. In: Correct System Design (1999)
40. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
41. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security (TISSEC) 3(1) (2000)
42. Sewell, P., Vitek, J.: Secure composition of untrusted code: box-π, wrappers and causality types. Journal of Computer Security 11(2) (2003)
43. Skalka, C., Smith, S.: History effects and verification. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, Springer, Heidelberg (2004)
44. Talpin, J.P., Jouvelot, P.: The type and effect discipline. Information and Computation 2(111) (1994)
45. Toma, I., Foxvog, D.: Non-functional properties in Web Services. WSMO Deliverable (2006)
46. Vallecillo, A., Vansconcelos, V., Ravara, A.: Typing the behaviours of objects and components using session types. In: Proc. of FOCLASA (2002)
47. Winskel, G.: The Formal Semantics of Programming Languages. The MIT Press, Cambridge (1993)
48. Woo, T.Y.C., Lam, S.S.: A semantic model for authentication protocols. In: IEEE Symposium on Security and Privacy (1993)