Semantics-based design for Secure Web Services

Massimo Bartoletti Pierpaolo Degano Gian Luigi Ferrari Roberto Zunino
bart@di.unipi.it degano@di.unipi.it giangi@di.unipi.it zunino@di.unipi.it
Dipartimento di Informatica
University of Pisa
L.go Bruno Pontecorvo, 3 - 56127 Pisa, Italy
Phone +390502212700 Fax +390502212726

Abstract

We outline a methodology for designing and composing services in a secure manner. In particular, we are concerned with safety properties of service behaviour. Services can enforce security policies locally and can invoke other services that respect given security contracts. This call-by-contract mechanism offers a significant set of opportunities, each driving secure ways to compose services. We discuss how to correctly plan services compositions in several relevant classes of services and security properties. To this aim, we propose a graphical modelling framework, based on a foundational calculus called λ^{req} [13]. Our formalism features dynamic and static semantics, so allowing for formal reasoning about systems. Static analysis and model checking techniques provide the designer with useful information to assess and fix possible vulnerabilities.

Index Terms

Web services, call-by-contract, language-based security, static analysis, system verification.

I. Introduction

Service-oriented computing (SOC) is an emerging paradigm to design distributed applications [42], [41], [25]. SOC applications are obtained by suitably composing and coordinating (i.e. *orchestrating*) available services. Services are stand-alone computational units distributed over a network, and made available through standard interaction mechanisms. Composition of services may require peculiar mechanisms to handle complex interaction patterns (e.g. to implement transactions), while enforcing non-functional requirements on the system behaviour, like e.g. security, availability, performance, transactionality, quality of service, etc. [46]. An important aspect is that services are *open*, in that they are built with little or no knowledge about their operating environment, their clients, and further services therein invoked. Web Services [2], [45], [48] built upon XML technologies are possibly the most illustrative and well developed example of the SOC paradigm. A variety of XML-based technologies have been devised for describing, discovering and invoking web services [17], [19], [16], [49]. There are also several standards for defining and enforcing non-functional requirements of services, e.g. WS-Security [4], WS-Trust [3] and WS-Policy [47] among others.

The success of the SOC paradigm is highly related to the development of network infrastructures supporting interoperable and secure messaging among services, as well as high-level

coordination standards. A further element is the definition of novel methodologies for modelling, analysing and certifying SOC systems, see e.g. [26]. A challenging issue for SOC research is how to compose existing services into more complex ones, also by properly selecting and configuring services so to guarantee that their composition enjoys some desirable properties. Non-functional aspects, e.g security, make service composition even harder.

From a methodological perspective, Software Engineering should facilitate the shift from traditional approaches to the emerging service-oriented solutions. Along these lines, one of the goals of this paper is to strenghten the adoption of formal techniques for modelling, designing and verifying SOC applications. In particular, we propose a SOC modelling framework supporting *history-based security* and *call-by-contract*.

The starting point of our work is λ^{req} [13], [8], a foundational calculus for describing, selecting and securely composing services. The execution of a program may involve accessing security-critical resources; these actions are logged into histories. The security mechanism may inspect these histories, and forbid those executions that would violate the prescribed policies. The call-by-contract selection mechanism implements a matchmaking algorithm based on service behaviour. This algorithm exploits static analysis techniques to detect the plans for resolving the call-by-contract involved in a service orchestration.

In our modelling framework, services are rendered as typed diagrams. Service interfaces extend the WSDL interfaces: besides the standard WSDL attributes, we add semantic information about service behaviour. In our model, the published interface of a service is an annotated functional type, of the form $\tau_1 \xrightarrow{H} \tau_2$. Intuitively, when supplied with an argument of type τ_1, the service evaluates to an object of type τ_2. The annotation H is a sort of context-free grammar that describes all the possible run-time histories of the services. Thus, H can be exploited to constrain the selection of a service that respects the desired security properties. Service interfaces are mechanically inferred by a type and effect system.

To select a service that matches a given contract, a client issues a request of the form $\mathtt{req}_r\tau$. A request type $\tau = \tau_1 \xrightarrow{\varphi} \tau_2$ matches those services with interface $\tau_1 \xrightarrow{H} \tau_2$ whose abstract behaviour H respects the policy φ. These services are guaranteed to respect φ in all the possible executions.

The selection algorithm searches a *service repository* for an interface matching the request type. 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.

The main contributions of the present paper are the following:

1) 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. Our diagrams have a formal operational semantics, that specifies the dynamic behaviour of services. Moreover, diagrams can be statically analysed, to infer the contracts satisfied by a service. Our approach allows for a fine-grained characterization of the design choices that affect security (Section II). We support our proposal with some case study scenarios.
2) the integration of a verification technique to study composition properties of service networks. A static analysis is used to infer an abstraction of the behaviour of a network. This abstraction is then model-checked to construct a correct orchestrator that coordinates the running services in a secure manner. Secure orchestration will also allow for improving the overall performance by avoiding unnecessary dynamic security checks while executing services. Studying the output of the model-checker may highlight possible design flaws, and suggest how to revise the calls-by-contract and the security policies. All the above machinery is completely mechanizable, and we are implementing a tool to support our methodology. The fact that the tool is based on strong theoretical grounds (i.e. λ^{req} type and effect inference and verifier) positively impacts the reliability to our approach.
3) a study of various planning and recovering strategies. We discuss several situations in which one needs to take a decision before proceeding with the execution. For instance, when a planned service disappears unexpectedly, one can choose to replan, i.e. to adapt the current plan 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.

The paper is organized as follows. In Section II we introduce a taxonomy of security aspects in service-oriented applications. Section III, IV and V present our formal model. In particular, Section III introduces our design notation and the operational semantics; Section IV presents service contracts, and outlines how they can be automatically inferred; Section V illustrates how to select services under the call-by-contract philosophy, and discusses some planning and recovering strategies. Two scenarios for secure service composition are presented in Section VI. We conclude the paper with some remarks (Section VII) about the expected impact of our methodology on Software Engineering, and we discuss some related works.

II. 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 actual access rights of a running piece of code depend on (a suitable abstraction of) the execution history of all the pieces of code run so far. This approach has been receiving major attention, both at the level of foundations [5], [30], [44] and of language design and implementation [1], [27].

The observations of security-relevant activities, e.g. opening socket connections, reading and writing files, accessing critical memory regions, are called *events*. *Histories* are sequences of events. 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 [43].

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 history across distinct invocations (yet it checks the history within each invocation). Instead, a stateful service keeps track of the histories of all the past invocations. Steteless services can enforce policies that inspect the history of the current invocation only, e.g. resource usage control. Stateful services allow for more expressive security policies: for instance, a stateful service can bound the number of invocations on a per-client basis, while a stateless service cannot. More in general, stateful services can exploit their histories to record security-relevant information about the state of client sessions. Consider for instance a service that requires password authentication, and that gives only three chances per hour to authenticate. This can be modelled as a stateful service. In this case the history keeps track of the number of failed authentication attempts. The security policy prevents the service from being used by a caller for which the history has recorded three failed authentication attempts in the last hour. Although stateless services admit security policies that are less expressive than those of stateful services, static analysis can usually infer enough information to ensure secure composition. For instance, consider a client that wants to buy some pharmaceuticals through an online vendor, while being assured that his shopping list is not leaked to other services (e.g. insurance companies). Statically analysing pharmacy services permits to match the client with a service conforming to its requirements. This is because the information recorded in stateless histories is enough to show possible leaks.

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. For instance, consider an alliance of services trusting each other, that wish to implement a distributed black-list. More in detail, when any of the services in the alliance black-lists (through a suitable event in the history) an external service, then all the alliance must abstain from invoking that external service. This can be implemented

by making the services in the alliance share a global history. Before invoking an external service, the caller inspects the global history to find whether it has been black-listed beforehand. Note that the trust relationship among the services in the alliance is crucial: even a single untrusted service could compromise the security of the alliance, by maliciously modifying the global history. To securely implement the global history, i.e. to protect its integrity, communication between services is done through suitable cryptographic protocols. These protocols must be designed to be coherent with the existing trust relationship, e.g. signed histories are considered reliable only if the signer is trusted [51].

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 enforces a given security policy. When there is no mobile code, more efficient mechanisms can be devised, e.g. local checks on security-critical operations. For instance, Java Applets are mobile code applications running on the client browser, and they can be invoked through higher order requests. A browser calls a service that returns an applet; the browser runs then the applet, while enforcing its own security policy, e.g. the standard Java sandboxing. Actually, history-based security is more general than Java, since it checks the *all* the past execution history, while Java only checks the frames in the call stack [34]. Higher order requests also allow a client to pass some mobile code as a parameter to a service. The execution of that code can be constrained both by the client and by the service, by suitable policies imposed on its execution.

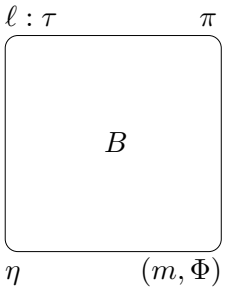
Dependent / independent threads

In a network of services, several threads may run concurrently and compete for services. A thread is a computation started from a particular *initiator* service in the network. Roughly, initiator services are those delegated by the active actors to attain a given goal. Independent

threads keep execution 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. Implementing independent threads requires that each service records the history of each thread that invoked that service, i.e. services actually maintain a mapping from thread initiators to their histories. For instance, consider a service that can be invoked only once. If threads are independent, this "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.

III. SERVICES

The basic entity in our graphical formalism is that of *service*. We shall first describe the syntax of services, and then study how they behave when plugged into a network. 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 IV). 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 η is an abstraction of the service execution *history*. In particular, we are concerned with the sequence of security-relevant events 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*. The monitor must check that services adhere to each policy φ_i, for $i \in 1..k$, when the flag is on. Security policies are modelled as regular properties

of event histories, i.e. properties that are recognizable by a finite state automaton. Although the soundness of our design and analysis techniques does not depend on the logic chosen for expressing regular properties of histories, we find it convenient to express policies through our *template security automata* (see below).

The label π is the *plan* used for resolving service choices. A plan formalises how a call-by-contract $\texttt{req}_r\tau$ is transformed into a call-by-name. Plans may come in several different shapes [11]. Here we focus on a very simple form of plans, i.e. mappings from request labels r to service locations ℓ. We represent plans with the syntax defined in Fig. 2.

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

Fig. 2. Syntax of plans.

The empty plan 0 has no choices; the plan $r[\ell]$ associates the service published at site ℓ with the request labelled r. A plan is *complete* when it has no unresolved choices. Composition $\mid$ on plans is associative, commutative and idempotent, and its identity is the empty plan 0. Since plans are functions, they have a single choice for each request, i.e. $r[\ell] \mid r[\ell']$ implies $\ell = \ell'$.

The label B inside the box is a *block* that describes the workflow behaviour of the service, in particular of the security-relevant activities. Formally, B it is a control flow graph [40] with nodes modelling activities, and arrows modelling intra-procedural flow. The syntax of activities and flow graphs are defined in Fig. 3, while Fig. 4 summarizes the relevant syntactic categories.

Activities comprise basic activities, events, requests, security blocks, and planning blocks.

- A *basic activity* a is an internal computation that does not affect security-critical objects. For instance, evaluating a boolean or arithmetic expression are basic activities.
- An *event* $\alpha(o)$ abstracts from a security-critical action α performed on an object o. For instance, $write(foo)$ for writing the file foo, $sgn(\ell)$ for a certificate signed by ℓ, etc. We simply write α when the target object is immaterial.
- A *service request* takes the form $\texttt{req}_r\tau$. The label r uniquely identifies the request in a

A	$::=$	activities
	a	basic activity
	$\alpha(o)$	event
	$\mathtt{req}_r\tau$	request
	$\varphi[B]$	security block
	$\{B\}$	planning block
B	$= (N, \Lambda, \rightarrow)$	service flow graph
where	N	finite set of nodes
	$\Lambda : N \rightarrow A$	labelling function
	$\rightarrow \subseteq N \times N$	set of arrows

Fig. 3. Syntax of services.

network of services, and the request type τ is defined by:

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

where b is a base type ($Int, Bool, void, \ldots$). 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 $\tau \rightarrow \tau'$, and whose behaviour respects the policy φ.

- A *security block* $\varphi[B]$ annotates B with the policy φ, with the intention that φ must be obeyed while running B. This can be accomplished by enabling the execution monitor, that checks the history against φ at each step of the execution of B. We shall see later on a static analysis of services that will allow to turn off the execution monitor, while guaranteeing that the policy φ is obeyed.
- A *planning block* $\{B\}$ constructs a plan for the execution of B (see Section V for more details on how this is accomplished).

Both kinds of blocks can be nested, and they determine the scope of policies (hence called *local policies* [7]) and of planning.

Service flow graphs $B = (N, \Lambda, \rightarrow)$ are directed graphs, where N is the (finite) set of nodes, Λ is a function that maps nodes to activities, and $\rightarrow$ is the set of arrows. Note that loops are

$o, o', \ldots$	$\in \mathsf{Obj}$	Objects (a finite set)
$\alpha, \alpha', \ldots$	$\in \mathsf{Act}$	Actions (a finite set)
$\alpha(o), \ldots$	$\in \mathsf{Ev} = \mathsf{Act} \times \mathsf{Obj}$	Events
$\eta, \eta', \ldots$	$\in \mathsf{Ev}^*$	Histories (finite sequences of events)
$\varphi, \varphi', \ldots$	$\in \mathsf{Pol}$	Policies (regular properties of histories)
$r, r', \ldots$	$\in \mathsf{Req}$	Request labels
$\ell, \ell', \ldots$	$\in \mathsf{Loc}$	Service Locations
$\pi, \pi', \ldots$	$\in \mathsf{Req} \to \mathsf{Loc}$	Plans (functions from Req to Loc)

Fig. 4. Syntactic categories.

permitted in flow graphs, to model iterative computations. We assume that each graph has a single entry node, and a single exit node.

This graphical formalism is based on λ^{req}, a call-by-value λ-calculus enriched with local security policies and call-by-contract service requests. Since our main focus is on secure composition, in the graphical model we do not render all the features of λ^{req}. In particular, we neglect variables, higher-order functions, and parameter passing. However, we feel free to use these features in our examples, because their treatment can be directly inherited from λ^{req}.

Semantics of services

We formally define the behaviour of services through a graph rewriting semantics [6]. In this section, we assume that the services which initiate a computation are furnished with an arbitrary plan. In the next section, we shall discuss a static machinery that will enable us to construct these plans so to guarantee that computations will never go wrong, i.e. they satisfy all the contracts and the security policies on demand. We shall also show some strategies to adopt when services disappear unexpectedly (Section V).

The graph rewriting semantics for the case of dependent threads is split in two parts: basic activities, events, security blocks, requests and returns are shown in Fig. 5. Instead, Fig. 10 details the rules that involve planning and recovering strategies, i.e. planning blocks, requests to unavailable services, unresolved requests, services going down and up, and publication of new services. We shall briefly discuss the case of independent threads below in this section.

All the remaining axes in the taxonomy are covered by our semantics When irrelevant, we omit the label τ in services. Note that the actual values for some labels in rules REQ and RET are defined later on in Fig. 6, since they depend on the choice made on the security aspects discussed in Section II. This gives rise to different behaviours of requests and returns according to the possible choices in the taxonomy.

The configurations of our semantics are sets (i.e. networks) of services. We mark the next activity to be performed by a running service with an overline, e.g. $\overline{\alpha(o)}$ means that the event $\alpha(o)$ is about to be fired. An activity just executed is marked with an underline, e.g. $\underline{\alpha(o)}$. We extend this notation to blocks B, i.e. $\overline{B}$ means that the entry node of B is the next activity, while $\underline{B}$ means that the exit node of B was the last executed one. Also, configurations comprise dashed arrows that connect a running request with the invoked service.

We now briefly discuss the graph rewritings in Fig. 5. Note that we only depict rewriting within a context. For instance, the rewriting $\overline{a} \Rightarrow \underline{a}$ in the SKIP rule can be applied in any context, i.e. within any block B containing $\overline{a}$.

- A basic activity is just passed over (rule SKIP).
- The evaluation of an event $\alpha(o)$ requires checking compliance of the new history $\eta\alpha(o)$ with each policy φ in Φ (denoted $\eta\alpha(o) \models \Phi$), if the execution monitor is on. If all the policies are respected (rule EV), then $\alpha(o)$ is appended to the current history, and the execution proceeds to the next block. Otherwise, if some policy is violated (rule FAIL), then the execution goes into a stuck state `fail`. This state models a security exception (for simplicity, we do not model here exception handling; extending our formalism in this direction would require to define, among the other things, how to compensate from aborted computations, e.g. like in Sagas [32], [20])
- The rule SEQ says that, after a block B has been evaluated, the next activity is chosen among the blocks with incoming arrows from 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, and the execution monitor is on, the evaluation gets stuck.
- A request $\texttt{req}_r\tau$ 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),

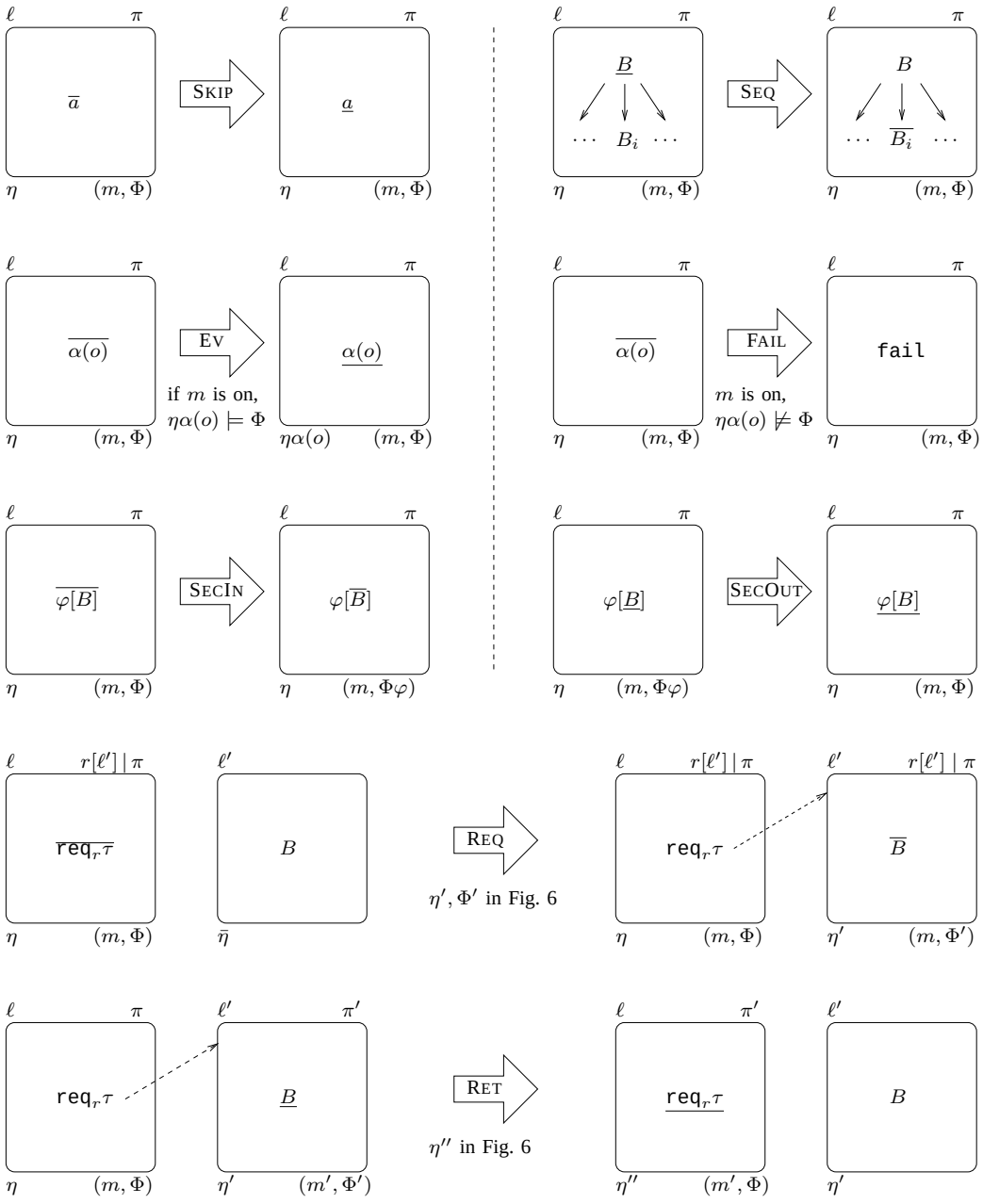


Fig. 5. Semantics of services in the case of dependent threads (Part I).

and waits until it returns. Note that the meaning of the labels η' and Φ' is left undefined in Fig. 5, since it depends on the choice made on the security aspects discussed in Section II. The actual values for the undefined labels are shown in Fig. 6. In particular, the initial history of the invoked service is: (i) empty, if the service is stateless with local history; (ii)

	Stateless service ℓ'		Stateful service ℓ'	
	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. 6. Histories and policies in four cases of the taxonomy.

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 the history η'' are defined in Fig. 6.

The cases N/A, PLG IN, PLG OUT and UNRES are defined in Fig. 10, 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 V.

A plan is *viable* when it drives no stuck computations (unless some service mentioned in the plan becomes unavailable). Under a viable plan, a service can always proceed its execution without attempting to violate some security policy (therefore the execution monitor is unneeded), and it will always manage to resolve each request. The static machinery described in Section IV discovers viable plans.

An example

Consider a network composed by two services at locations ℓ and ℓ', both stateful and sharing a global history. The service at ℓ starts with an action α (the omitted target object is immaterial in this example), and then issues a request r, resolved to ℓ' by the provided plan. The service at ℓ' performs α within a security block $\varphi[\cdots]$, where the policy φ prevents α from being fired twice. A computation of the network is depicted in Fig. 7, where we assume the execution monitor is on. Note that the plan $r[\ell']$ is not viable, because it drives a computation that fails because of an attempted security violation, right before the second α is fired at ℓ'.

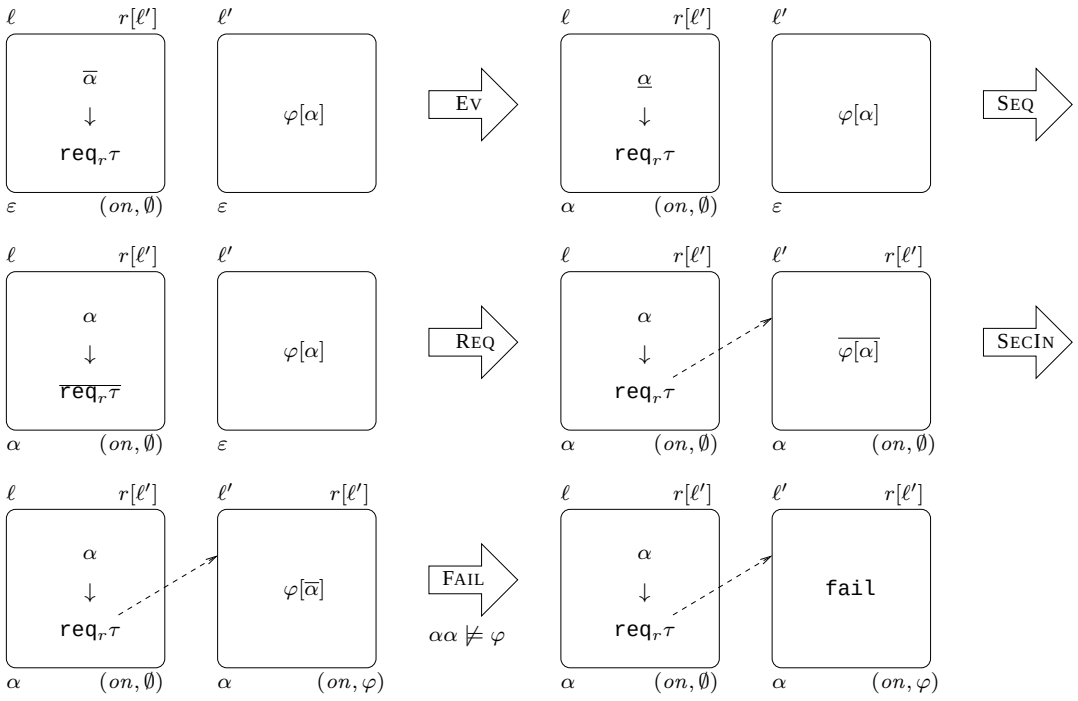


Fig. 7. A computation of two stateful services with global history.

Independent threads

To model independent threads, each service must keep a separate history for each thread. Equivalently, we keep a history for each thread initiator. To this aim, instead of a single history η, services now carry a function L mapping the label ℓ_I of each initiator to its corresponding history. Moreover, we keep track of the initiator name: each service invoked on behalf of the initiator ℓ_I is tagged as such. The semantics of services can now be easily adapted, making $L(\ell_I)$ play the role of η in the semantics for dependent threads. We depict in Fig. 8 the rule REQ for independent threads. The relation among η, η', Φ and Φ' is still the one defined by Fig. 6. Note that η' is used to update $\bar{L}(\ell_I)$, only: all other histories in $\bar{L}$ are left unchanged. The rule RET is dealt with similarly, as for the rules FAIL, PLG IN and PLG OUT.

Modelling security policies

Security policies φ are regular properties of histories, and they are defined through *template security automata* $A_{\varphi(x)}$. A template security automaton gives rise to a finite state automaton

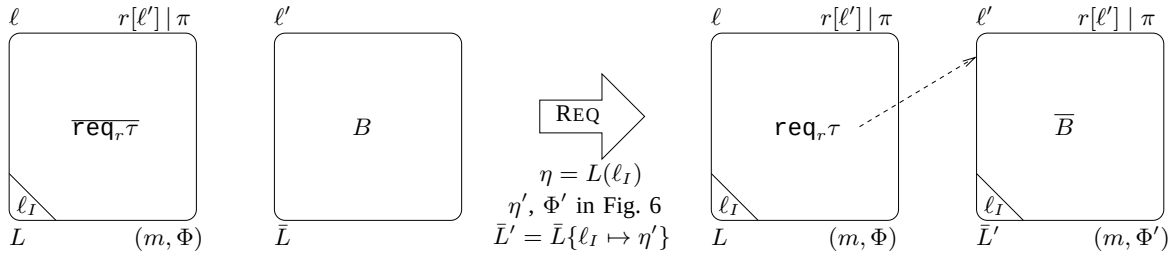


Fig. 8. Maintaining separate histories in the case of independent threads.

when the parameter x is instantiated to an actual object o. These automata will be exploited to recognize those histories obeying φ. Formally, a *template security automaton* $A_{\varphi(x)}$ is a 5-tuple (S, Q, q_0, F, E), where x is a parameter, and:

- $S \subseteq \mathsf{Act} \times (\mathsf{Obj} \cup \{x, \bar{x}\})$ is the input alphabet,
- Q is a *finite* set of states,
- $q_0 \in Q \setminus F$ is the start state,
- $F \subset Q$ is the set of final "offending" states,
- $E \subseteq Q \times S \times Q$ is a finite set of *template* edges, written $q \overset{\vartheta}{\multimap} q'$.

The edges in a template security automaton can be of three kinds: either $q \overset{\alpha(o)}{\multimap} q'$ where o is an object, or $q \overset{\alpha(x)}{\multimap} q'$, or $q \overset{\alpha(\bar{x})}{\multimap} q'$ (where $\bar{x}$ means "different from x"). Given a object $o \in \mathsf{Obj}$, a template security automaton $A_{\varphi(x)}$ is instantiated into a finite state automaton $A_{\varphi(o)}$ by binding the variable x to o. The intuition is that $q \overset{\alpha(x)}{\multimap} q'$ will result in an actual transition $q \overset{\alpha(o)}{\longrightarrow} q'$, while $q \overset{\alpha(\bar{x})}{\multimap} q'$ will give rise to a finite set of transitions $q \overset{\alpha(o')}{\longrightarrow} q'$, for all $o' \in \mathsf{Obj} \setminus \{o\}$.

More formally, let $A_{\varphi(x)} = (S, Q, q_0, F, E)$ be a template security automaton, and let $o \in \mathsf{Obj}$. The finite state automaton $A_{\varphi(o)}$ is defined by the 5-tuple $(\mathsf{Ev}, Q, q_0, F, \delta)$, where the transition relation δ is defined as follows:

$$\delta = \bar{\delta} \cup \{\, q \overset{\alpha(o')}{\longrightarrow} q \,|\, o' \in \mathsf{Obj}, \nexists q' : q \overset{\alpha(o')}{\longrightarrow} q' \in \bar{\delta} \,\}$$

where the auxiliary relation $\bar{\delta}$ is defined as:

$$\begin{aligned}\bar{\delta} \;=\; & \{\, q \xrightarrow{\alpha(o)} q' \mid q \overset{\alpha(x)}{\multimap} q' \in E \,\} \\ & \cup \bigcup_{o' \in \mathsf{Obj}\setminus\{o\}} \{\, q \xrightarrow{\alpha(o')} q' \mid q \overset{\alpha(\bar{x})}{\multimap} q' \in E \,\} \\ & \cup \{\, q \xrightarrow{\alpha(o')} q' \mid q \overset{\alpha(o')}{\multimap} q' \in E \,\}\end{aligned}$$

Note that the definition for δ adds self-loops for all the events not explicitly mentioned in the template automaton $A_{\varphi(x)}$.

We say that a history η *respects* φ, written $\eta \models \varphi$, when η is *not* in the language of $A_{\varphi(o)}$, for all $o \in \mathsf{Obj}$. When η is in the language of $A_{\varphi(o)}$ for some object o, we say that η *violates* φ, written $\eta \not\models \varphi$. Note that instantiated template security automata are non-deterministic. Given a history η and a policy φ, we want that *all* the traces of the instances of $A_{\varphi(x)}$ comply with φ. This is a form of diabolic (or internal) non-determinism. To account for that, we make the "offending" states as final — thus going into a final state represents a violation of the policy, while the other states mean compliance to the policy.

IV. Call-by-contract

A service B 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 similar to that in [13]. Also, we assume that services cannot invoke each other circularly, since this is quite unusual in the SOC scenario.

Contracts

The types τ are annotated with *history expressions* H that over-approximate the possible run-time histories. Fig. 9 displays the syntax of types and history expressions. 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 $\tau \xrightarrow{\varphi} \tau'$.

History expressions, defined in Fig. 9, 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 branching, security blocks $\varphi[H]$, recursion $\mu h.H$ (μ binds the occurrences of the variable h in H), localization $\ell : H$, and planned selection $\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$.

τ, τ'	$::=$	types
	b	base type
	$\tau \xrightarrow{H} \tau'$	annotated functional type
H, H'	$::=$	history expressions
	ε	empty
	h	variable
	$\alpha(o)$	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

Fig. 9. Service interfaces: annotated types and history expressions

The *semantics* of a history expression is a set of histories η, possibly carrying security annotations in the form $\varphi[\eta]$. We denote by $[\![H]\!]$ the semantics of H. We now briefly describe how this semantics is constructed: see [8] for the formal treatment.

- The semantics of the history expression α is the set of histories $\{\alpha\}$.
- The semantics of $H \cdot H'$ is the set of histories $\eta\eta'$ such that $\eta \in [\![H]\!]$ and $\eta' \in [\![H']\!]$.
- The semantics of $H + H'$ comprises the histories η such that $\eta \in [\![H]\!] \cup [\![H']\!]$.
- The semantics of $\varphi[H]$ is the set of histories $\varphi[\eta]$ such that $\eta \in [\![H]\!]$.
- The semantics of $\mu h.\, H$ is the least fixed point of the operator $f(\mathcal{H}) = [\![H]\!]_{\{\mathcal{H}/h\}}$, where we denote with $[\![H]\!]_\rho$ the semantics of a history expression H in an environment ρ, mapping variables h to sets of histories. 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. E.g., $\ell : \alpha \cdot (\ell' : \alpha') \cdot \beta$ denotes two histories: $\alpha\beta$ occurring at location ℓ, and α' occurring at ℓ'.
- A planned selection abstracts 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. Intuitively, the history expression $H = \{r[\ell_1] \triangleright H_1, r[\ell_2] \triangleright H_2\}$ is associated with a request r that can be resolved into either 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.

Certifying contracts and planning

Our planning and verification technique for services is inherited from that of the λ^{req} calculus. The interested reader can find the formal foundations of our work in [10], [8]. Here, we only summarize the relevant results for our present modelling framework.

A static analysis over our diagrams infers judgements of the form $H \vdash B : \tau$. Roughly, this means that the service with code B has type τ, and its execution histories are represented by the history expression H. This static analysys enjoys two fundamental results.

Correctness. *Effects correctly over-approximate service run-time histories*

More formally, consider a service with code B such that $H \vdash B : \tau$. If running the service (plugged into a network) generates a history η, then $\eta \in [\![B]\!]$ ([8], Th. 1).

The second property of our static analysis is *type safety*. We say that an effect H is *valid* under a plan π when the histories denoted by H never violate the security policies in H. Type safety ensures that, if (statically) the effect of a service B is valid under a plan π, then (dynamically) the plan π is viable for B.

Type safety. *Valid effects drive computations that never attempt security violations.*

More formally, consider a service with code B such that $H \vdash B : \tau$. If H is valid under π, then the execution is *secure* (i.e. $\eta \models \Phi$ whenever a running service carries labels η and (m, Φ)), and it never reaches a `fail` configuration. Moreover, if π is complete (i.e. it has no unresolved choices $r[?]$) and the chosen services do not disappear, then the execution monitor can be safely kept off ([8], Th. 2).

A further result is that the validity of history expressions can be mechanically verified.

Model-checking. *Validity for history expressions is model-checkable.*

The actual model-checking algorithm for the validity of H returns the set of plans under which H is valid. Determining such viable plans is not a trivial task: indeed, resolving requests independently might not lead to a viable plan, as discussed in the introduction. Our model-checking technique requires several preliminary steps. Technically, we first *linearize* the history expression H to collect all the plans and the associated effects, while preserving the semantics of H ([8], Ths. 4 and 5). The resulting history expression has the form of a single, top-level planned selection $\{\pi_1 \triangleright H_1 \cdots \pi_k \triangleright H_k\}$. We can then check each H_i independently. First, H_i is *regularized* to remove the redundant framings $\varphi[\cdots\varphi[\cdots]\cdots]$ ([8], Th. 6). This makes possible to construct a finite state automaton that recognizes the validity of histories ([8], Lemma 8). This finite state automaton is then used in the actual model-checking of H_i ([8], Th. 8). If H_i model-checks, then the associated plan π_i is viable. Summing up:

Planning = Correctness + Type Safety + Model Checking

V. Planning and recovering strategies

We now consider the problem of choosing the appropriate service for a block of requests. While one might defer service selection as much 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 problems and try to find a better way, typically by considering also r_2 when taking a choice for r_1.

As seen in the previous section, a complete viable plan π for a block B guarantees that B can be securely executed without execution monitoring, and that we will never get stuck unless a service mentioned in π becomes unavailable. 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.

To provide graceful degradation in our model, we also consider the unfortunate case of executing a request r when either r is still unresolved in the plan (Rule UNRES), or r is resolved with an unavailable service (Rule N/A, for Not Available). Notationally, unavailable services are represented as slashed boxes. Therefore, we will look for a way to continue the execution, possibly repairing the plan.

Figure 10 formalizes the behaviour of services related to planning and recovering.

- Rule DOWN models an idle service becoming unavailable. For simplicity, we assume that services cannot become unavailable while serving a request. Unavailable services are modelled as slashed boxes.
- Rule UP models an unavailable service going back to the available state.
- Rule PUB is for service publishing. The behavioural interface τ of a new service (with code B) must be statically certified, written $H \vdash B : \tau$. Note that $H = \varepsilon$ for non-initiator services. The effect of an invoked service with type $\tau = \tau_0 \xrightarrow{H'} \tau_1$ is the latent effect H'.
- Rule N/A models a request to an unavailable service. Recovering from this situation demands for updating the current plan, and possibly activating the execution monitor. Below in this section we shall examine some possible strategies for doing that.
- Rule UNRES handles the case of unresolved requests. These are dealt with similarly to the rule N/A.
- Rule PLN IN describes which actions have to be taken when entering a planning block $\{B\}$. Before start the execution of B, we need to devise a plan for resolving the service requests in B. Again, several strategies are applicable, as discussed below.
- Rule PLN OUT simply exits from a planning block. This operation affects neither the plan nor the execution monitor.

We now discuss some strategies for constructing or repairing a plan. As a matter of fact, 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.).

We devise four main classes of strategies:

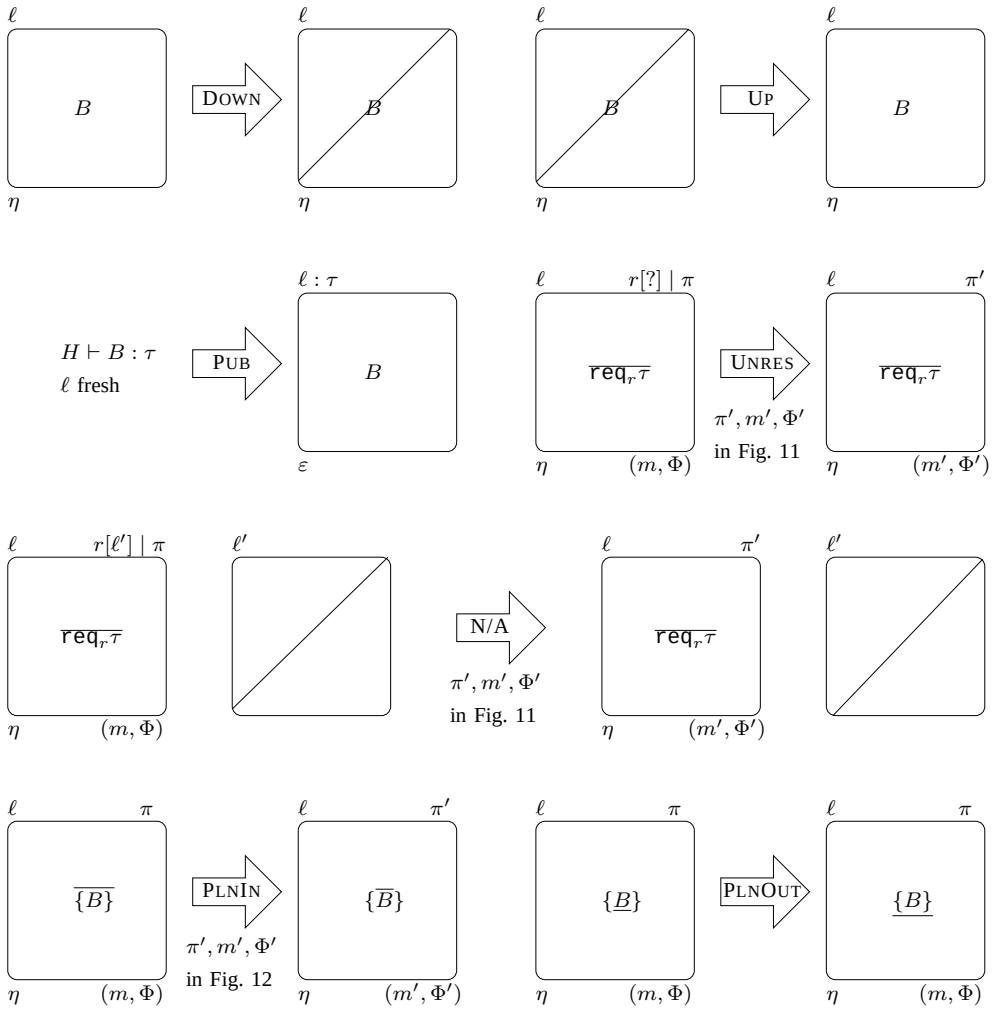


Fig. 10. Semantics of services in the case of dependent threads (Part II).

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.

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.

[1] In 1858, a man named John Gray was buried in old Greyfriars Churchyard in Edinburgh. The famous Skye Terrier, Greyfriars Bobby was so faithful to his master that for fourteen years, until his own death, Bobby lay on the grave only leaving for food.

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. 11, we describe the effects of these strategies in the context of the N/A and UNRES rules. There, we also make precise the recovered plan π' and the (m', 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 the plan with the new choice for r is checked for validity. 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 II: 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. 12 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 N/A 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

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' $(\mathit{off}, \Phi\varphi)$	all	The new plan π' has a choice for r

Fig. 11. Failure handling strategies for a request $\mathtt{req}_r\tau \xrightarrow{\varphi} \tau'$.

that we will likely get stuck later. Instead, if some services keep on being unavailable, we should rather consider the other strategies.

As for the N/A 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 N/A 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. 13, 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 a single occurrence of α, we 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.

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

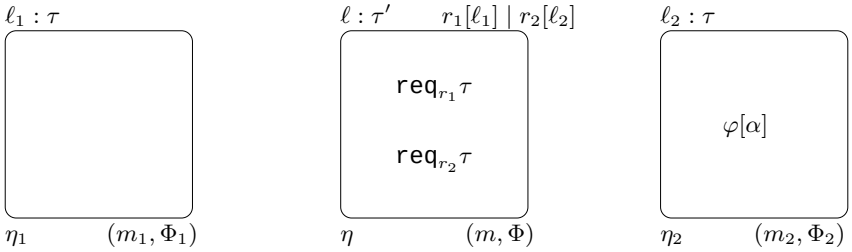


Fig. 13. An unsafe use of the Patch strategy.

Now, consider what happens if the service ℓ_1 becomes unavailable. The N/A 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. 11 is false, because $\ell_2 \in \pi$: therefore, this dangerous patch is correctly avoided.

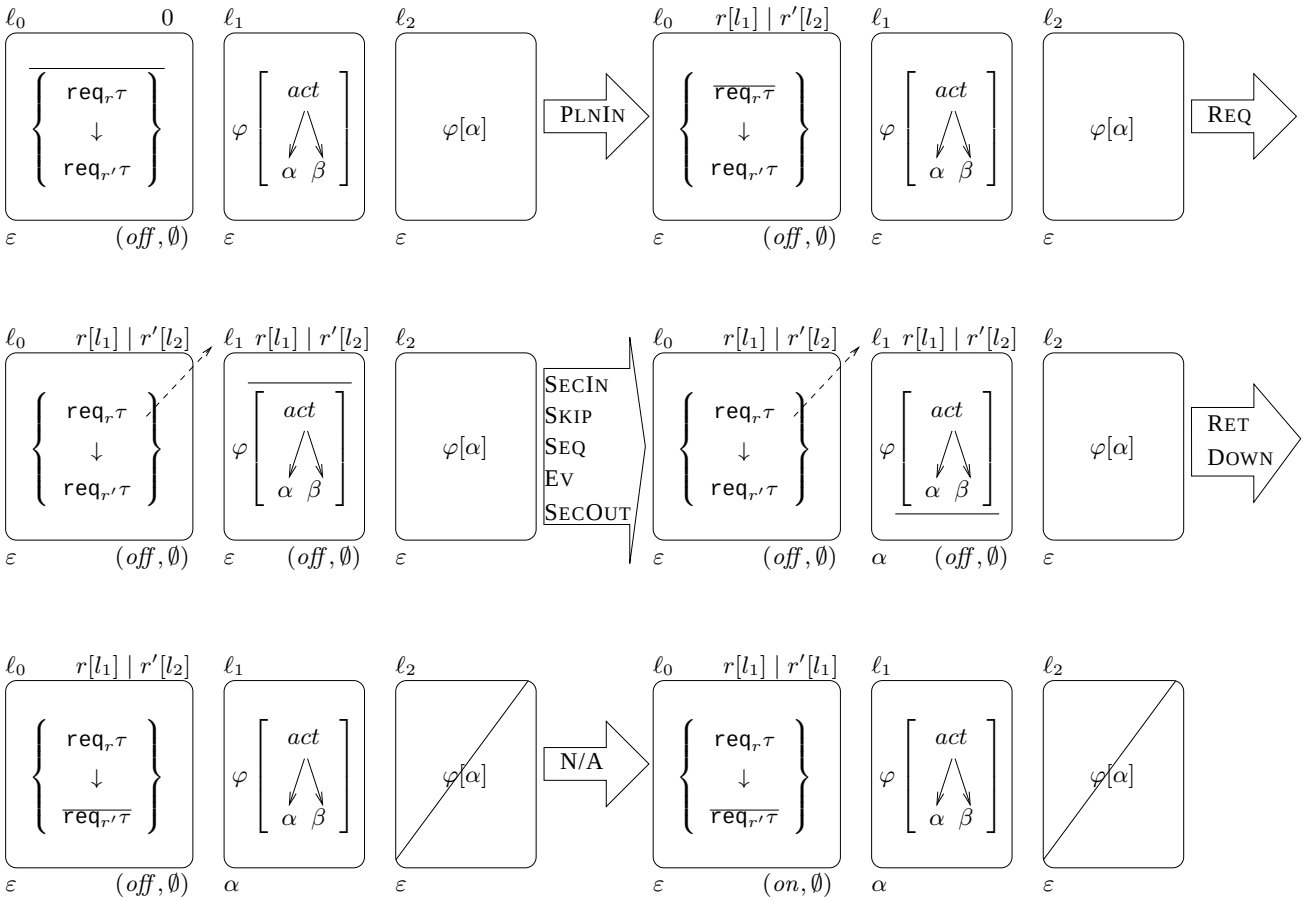


Fig. 14. Semantics Example

An example

We show in Fig. 14 a short example illustrating our service semantics, in the case of stateful local histories ($ILF\omega$). The network is composed of three services: an initiator (labelled ℓ_0) and two other services (ℓ_1 and ℓ_2). The initiator performs two requests using the same contract τ; in this example we simply assume that both services ℓ_1 and ℓ_2 are compatible with τ. The service ℓ_1, when invoked, runs some activity act and then may perform either the event α or the event β. Instead, ℓ_2 may perform α, only. Both services run their code under a local policy φ, stating that the event α can be performed at most once, in the whole life of each service. In the initial state, all the histories are empty (ε), the initiator has not yet computed a plan (0), and the execution monitor is not active (*off*).

We now comment on the transitions. For brevity, in Fig. 14 we sometimes compacted more steps in a single one, as we shall point out shortly. First, rule PLNIN is used to form a plan, resolving both requests r and r'. There are four possible plans, since each of the requests can be resolved with either ℓ_1 or ℓ_2. Invoking ℓ_2 twice will surely violate the policy φ. Service ℓ_1 could also invalidate the policy φ if the α branch is taken in both invocations: our static machinery, to err on the safe side, assumes this worst-case situation and consider invoking ℓ_1 twice as unsafe. So, the only viable plans are $r[\ell_1] \mid r'[\ell_2]$ and $r[\ell_2] \mid r'[\ell_1]$. In the figure we choose the first plan. In the second transition we simply use rule REQ and invoke ℓ_1. Then many transition rules are applied: we enter the security block with rule SECIN; we run act with rule SKIP; we choose the α branch with rule SEQ; we run α with rule EV; we finally exit the security block with rule SECOUT. The event α is therefore recorded in the history of ℓ_1. Finally, we can return to ℓ_0 using rule RET. Here, we show what happens if service ℓ_2 becomes unavailable through rule DOWN. To run request r' we can not use rule REQ, but instead we can apply rule N/A and try to recovery from the failure of ℓ_2. We then apply the Sandbox strategy. We turn on the execution monitor, and fix the plan so that r' resolves to some available service compatible with τ: in the example, ℓ_1. Doing this, we shall run again service ℓ_1. If the service will attempt to perform another α, this will be prevented by the execution monitor. If instead ℓ_1 will choose the β branch, it will complete successfully.

VI. SCENARIOS FOR SECURE SERVICE COMPOSITIONS

To illustrate some of the features and design facilities made available by our framework, we consider two small case studies. First, we consider a car repair scenario, where a car may break and then request assistance from a tow-truck and a garage. The second scenario is about an embedded computational device that wants to delegate execution of mobile code.

A. Car repair

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 custom 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 customize the choice of garages, according to past experiences.

We start by modelling the CAR-EMERGENCY, 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 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. 15.

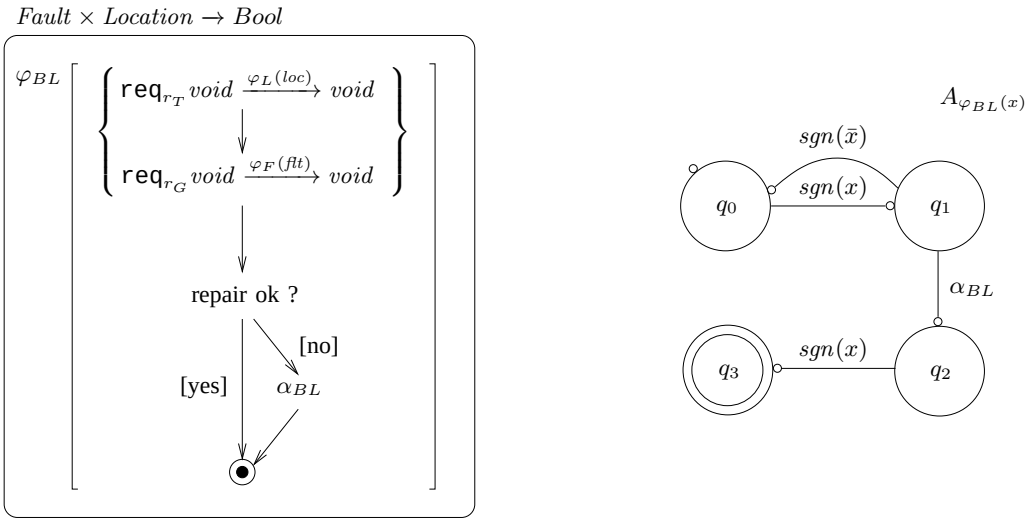


Fig. 15. The CAR-EMERGENCY service and the black-listing policy φ_{BL}.

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 security automaton in Fig. 15, 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 "offending" sink state q_3. For instance, the history $sgn(\ell_1)sgn(\ell_2)\,\alpha_{BL}\cdots sgn(\ell_2)$ drives the automaton $A_{\varphi_{BL}(\ell_1)}$ to the state q_3, thus violating 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. 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 $\varphi_F(\mathit{flt})$ 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 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. 16, on the left. The service will first fire the event $init_T$, to signal starting of execution, and then it will expose the list of geographic locations $\mathrm{ZIP}_1, \ldots, \mathrm{ZIP}_k$ it can reach. Each zip code ZIP_i is modelled as an event. The computation then branches, according to whether there are any available trucks. This is rendered as a basic activity with two outgoing edges. The contract $\varphi_L(loc)$ imposed by the CAR-EMERGENCY service ensures that the location loc is covered by the truck service. Formally, $\varphi_L(loc)$ checks if the zip code loc is contained in the interface of the tow-truck service (we omit the automaton for $\varphi_L(loc)$ 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 $void \xrightarrow{init_T \cdot \mathrm{ZIP}_1 \cdots \mathrm{ZIP}_k} void$.

The GARAGE service (Fig. 16, center) exposes the kinds of faults $\mathrm{REP}_1, \ldots, \mathrm{REP}_n$ the garage

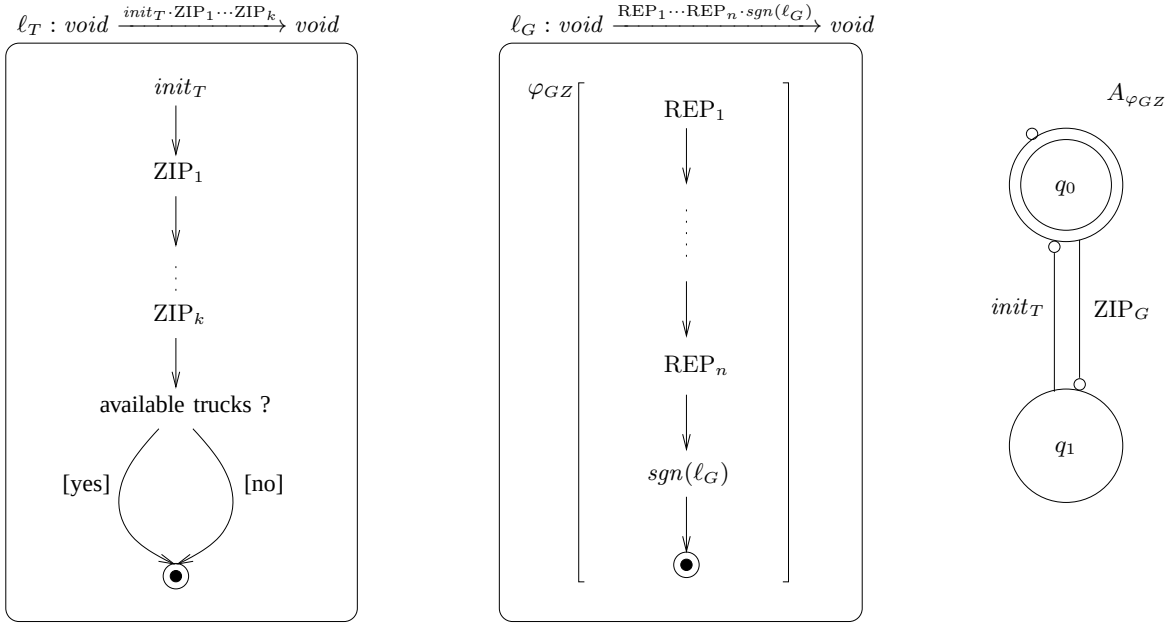


Fig. 16. The TOW-TRUCK (left) and GARAGE (right) services, and the Garage-Zip policy φ_{GZ}

can repair, e.g. tyres, engine, etc. The request contract $\varphi_F(\mathit{flt})$ 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, see Fig. 16, right) to ensure that the tow-truck can reach the garage address. Assume the garage is located in the area identified by ZIP_G. Then, the policy φ_{GZ} checks that the tow-truck has exposed the event ZIP_G among the locations it can reach. The event $init_T$ ensures that only the last invocation of the TOW-TRUCK service is considered. For instance, the history $init_T\,\mathrm{ZIP}_{G_1}\,init_T\,\mathrm{ZIP}_{G_1}\,\mathrm{ZIP}_G\,\mathrm{ZIP}_{G_2}$ obeys φ_{GZ} (recall that the instantiation of template security automata adds the self-loops for ZIP_{G_1} and ZIP_{G_2}, and that the final states are actually the offending ones). When both the contract $\varphi_L(loc)$ 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.

In Fig. 17, we show a system composed by one car ℓ_{CAR}, two TOW-TRUCK services ℓ_{T1}

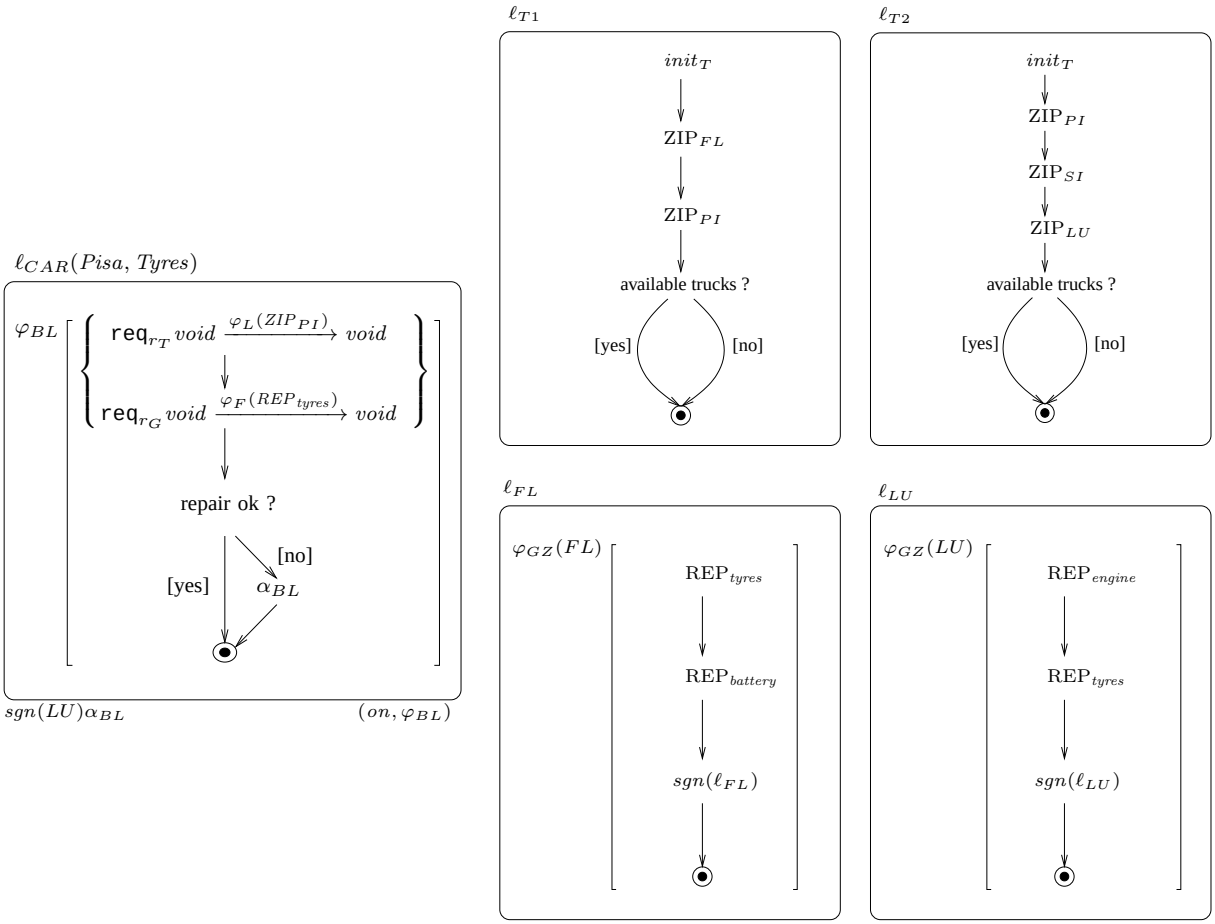


Fig. 17. The CAR-EMERGENCY client (ℓ_{CAR}), two tow-truck services (ℓ_{T1}, ℓ_{T2}), and two garages (ℓ_{FL}, ℓ_{LU}).

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 it 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. 15.

- 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.

B. Remote code execution

Consider the scenario depicted in Fig. 18. Assume that the client at site ℓ_0 is a device with limited computational capabilities, wanting to execute some code downloaded from the network. To do that, the client issues two (higher order) requests in sequence, the first one to obtain a piece of mobile code (e.g. an applet), and the second one to dispatch its execution to another service. The sites ℓ_1 and ℓ_2 are the available code providers, while ℓ_3 and ℓ_4 are the code executers. Modelling this scenario requires enriching our graphical notation with some extra features, e.g. parameter passing and higher-order services. In Fig. 18 we shall briefly introduce the needed notation. A more formal treatment can be obtained by using the calculus λ^{req} [12].

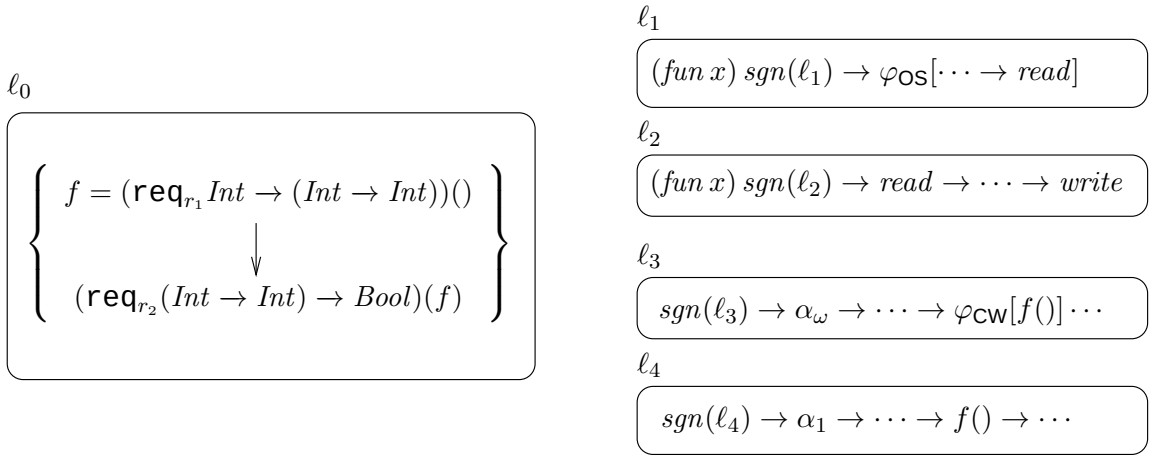


Fig. 18. One client (ℓ_0), two code providers (ℓ_1, ℓ_2), and two code executors (ℓ_3, ℓ_4). To deal with higher order, we introduce some extra notation. Passing a parameter f to a service invoked through a request r is denoted $\mathtt{req}_r\tau(f)$. A service returning a function that takes as input a parameter x and then evaluates the block B is denoted $(fun\, x)\, B$.

The request labelled r_1 asks for some code, and it can be served by two code providers at ℓ_1 and ℓ_2, both stateless and with local histories. The request type $Int \to (Int \to Int)$ means that, upon receiving a value of type Int, the invoked service replies with a function from Int to Int, with no security constraints.

The service at ℓ_1 returns a "one-shot" function that can be used only once. Within the function body, the only security-relevant operations are writing the service signature ($sgn(\ell_1)$) and reading ($read$) on the file system where the delivered code is run. The policy φ_{OS} ensures that the code is one-shot. To do that, φ_{OS} permits using the function in stateful sites only, and then prevents the event $sgn(\ell_1)$ from being executed twice (see the template security automaton $A_{\varphi_{\mathsf{OS}}(x)}$ in Fig. 19, right). We assume that a service declares that it supports stateful execution by emitting the event α_ω, while the event α_1 is for stateless services. The code provided by ℓ_2 first reads ($read$) some local data, and eventually writes them back ($write$) to ℓ_2.

Since ℓ_0 is assumed to have a limited computational power, the code f obtained by the request r_1 is passed as a parameter to the service invoked by the request r_2. This request can be served by either ℓ_3 or ℓ_4, both with local histories. The service at ℓ_3 is stateful (α_ω), and it runs the provided code f under a "Chinese Wall" security policy φ_{CW}, requiring that no data can be written after reading them (see $A_{\varphi_{\mathsf{CW}}}$ in Fig. 19, left). The service at ℓ_4 is stateless (α_1), and it simply runs the code f, with no security constraints.

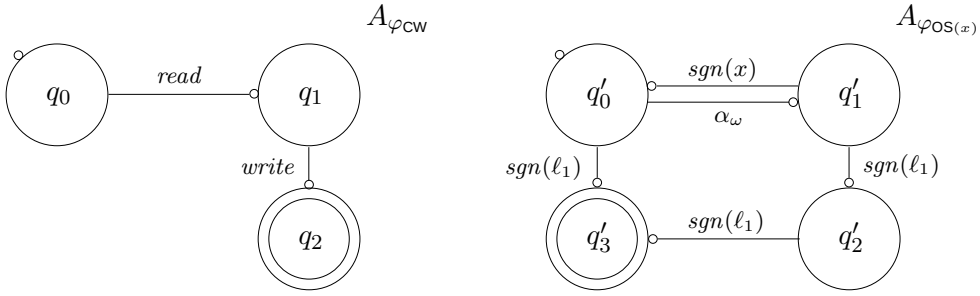


Fig. 19. The Chinese Wall policy φ_{CW} (left) and the one-shot policy $\varphi_{\mathsf{OS}}(x)$ (right).

The types inferred for the services are shown in Fig. 20. For instance, the type of ℓ_3 is a polymorphic function that, when applied to a function with a latent effect h (where h is an effect variable, to be bound to a history expression), will produce a value of type $Bool$, and a history in the semantics of $sgn(\ell_3) \cdot \alpha_\omega \cdot \varphi_{\mathsf{CW}}[h]$.

The abstract behaviour of the whole network of services is therefore rendered by the following

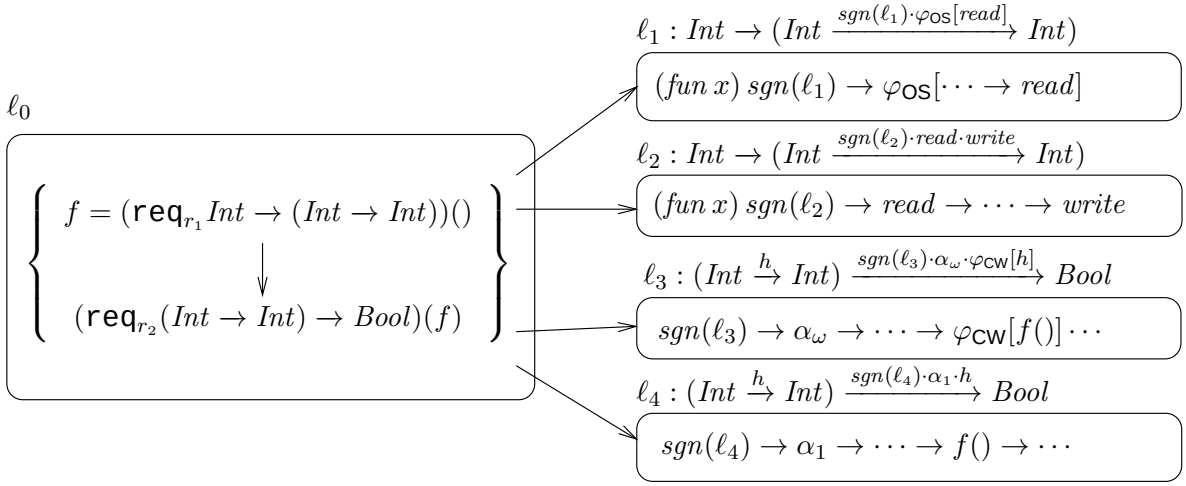


Fig. 20. One client, four services, and their certified published interfaces.

history expression H:

$$\{r_2[\ell_3] \rhd \ell_3 : sgn(\ell_3)\cdot\alpha_\omega\cdot\varphi_{\mathsf{CW}}[\{r_1[\ell_1] \rhd sgn(\ell_1)\cdot\varphi_{\mathsf{OS}}[read], r_1[\ell_2] \rhd read\cdot write\}]$$
$$r_2[\ell_4] \rhd \ell_4 : sgn(\ell_4)\cdot\alpha_1\cdot\{r_1[\ell_1] \rhd sgn(\ell_1)\cdot\varphi_{\mathsf{OS}}[read], r_1[\ell_2] \rhd read\cdot write\}\}$$

The intuitive meaning of H is that, under the plan $r_2[\ell_3]$, i.e. if r_2 is served by ℓ_3, the events $sgn(\ell_3)$ and α_ω are generated at site ℓ_3, followed by a security block φ_{CW}. This block wraps $sgn(\ell_1)\cdot\varphi_{\mathsf{OS}}[read]$ if ℓ_1 is chosen for r_1, or $read\cdot write$ if ℓ_2 is chosen instead. Otherwise, if r_2 is served by ℓ_4, then the behaviour (on site ℓ_4) depends on the former choice for r_1. If ℓ_1 was selected, then $sgn(\ell_1)\cdot\varphi_{\mathsf{OS}}[read]$, otherwise $read\cdot write$. Note also that no event is generated by the client at site ℓ_0.

The presence of higher order requests makes non-trivial analysing H to find if there is any viable plan. The problem is that the effect of selecting a given service for a request is not confined to the execution of that service. The history generated while running a service may later on violate a policy that will become active after the service has returned. Since each service selection affects the *whole* execution of a program, we cannot simply devise a viable plan by looking at local requests constraints, only.

In our example, we find that H is equivalent to the following H':

$$\begin{aligned} H' = \{ & r_1[\ell_1] \mid r_2[\ell_3] \rhd \ell_3 : sgn(\ell_3) \cdot \alpha_\omega \cdot \varphi_{\mathsf{CW}}[sgn(\ell_1) \cdot \varphi_{\mathsf{OS}}[read]], \\ & r_1[\ell_2] \mid r_2[\ell_4] \rhd \ell_4 : sgn(\ell_4) \cdot \alpha_1 \cdot sgn(\ell_2) \cdot read \cdot write, \\ & r_1[\ell_1] \mid r_2[\ell_4] \rhd \ell_4 : sgn(\ell_4) \cdot \alpha_1 \cdot sgn(\ell_1) \cdot \varphi_{\mathsf{OS}}[read], \\ & r_1[\ell_2] \mid r_2[\ell_3] \rhd \ell_3 : sgn(\ell_3) \cdot \alpha_\omega \cdot \varphi_{\mathsf{CW}}[sgn(\ell_2) \cdot read \cdot write] \} \end{aligned}$$

Every element of H' clearly separates the plan from the associated abstract behaviour. This piece of behaviour has no further plans within, and so it has all the information needed to model-check its validity. E.g., under the plan $r_1[\ell_1] \mid r_2[\ell_3]$, the abstract behaviour at site ℓ_3 is:

$$sgn(\ell_3) \cdot \alpha_\omega \cdot \varphi_{\mathsf{CW}}[sgn(\ell_1) \cdot \varphi_{\mathsf{OS}}[read]]$$

There are then four possible plans for the execution: $r_1[\ell_1] \mid r_2[\ell_3]$, $r_1[\ell_1] \mid r_2[\ell_4]$, $r_1[\ell_2] \mid r_2[\ell_3]$, and $r_1[\ell_2] \mid r_2[\ell_4]$. The plan $r_1[\ell_2] \mid r_2[\ell_3]$ is not viable, because it would drive a computation aborted by the execution monitor at site ℓ_3. The monitor aborts the execution just before generating the event $write$, because the history $sgn(\ell_3)\, \alpha_\omega\, sgn(\ell_2)\, read\, write$ (local at ℓ_3) would violate the Chinese-Wall policy φ_{CW}. The plan $r_1[\ell_1] \mid r_2[\ell_4]$ is not viable, too. Indeed, the history $sgn(\ell_4)\, \alpha_1\, sgn(\ell_1)$ at ℓ_4 violates the policy $\varphi_{\mathsf{OS}(\ell_4)}$ (recall that φ_{OS} prevents the code provided by ℓ_1 from being executed by stateless services). There are two further plans to consider, i.e. $r_1[\ell_1] \mid r_2[\ell_3]$ and $r_1[\ell_2] \mid r_2[\ell_4]$. These plans are judged viable by our static analysis, and indeed they drive executions that never fail.

Summing up, we have inferred the overall effect H, we have transformed it into a simple planned selection $\{\pi_1 \rhd H_1 \cdots \pi_k \rhd H_k\}$, and we have model-checked the validity of the H_i. The plans π_i associated with the valid H_i safely drive the execution, without resorting to any run-time monitor.

VII. Conclusions and related work

We have introduced a UML-like graphical language for designing and verifying security policies of service-oriented applications. The distinguished feature of our modelling framework is that it provides high-level constructs (e.g. call-by-contract) abstracting from the underlying middleware for service programming and deploying. We tackled the problem of modelling and verifying properties of service orchestration in the presence of security constraints. Our main

result is a semantic-based methodology for synthesizing the skeleton structure of the orchestration engine. The orchestration plan details which services the orchestrator engine has to choose in order to complete the original task, while obeying the security policies on demand. Here, we dealt with security policies, but our methodology can be applied to handle a variety of non-functional *safety* constraints.

We envisage the impact of our approach on the service protocol stack as follows. First, our proposal requires extending services description languages: besides the standard WSDL attributes, service description includes information about service behaviour. Moreover, the call-by-contract invocation mechanism adds a further *planning* layer to the standard service protocol stack. This layer provides the orchestrator with the plans guaranteeing that the relevant services always respect the required properties. The trustfulness of the planning layer and of the orchestrator follows from our formal approach, in particular from the soundness of the type and effect system, and the correctness of planning and of verification [8].

Another feature offered by our framework is that of mapping high-level service descriptions into more concrete λ^{req} programs. This can be done with the help of simple model transformation tools. Such model-driven transformation would require very little user intervention. Typically, the user just needs to (i) make the argument passing and return values explicit, e.g. decorating the diagram with variables, and (ii) annotate conditional branches with the proper guards. Transforming the diagram into a λ^{req} program can instead be performed in a completely automated way, e.g. by implementing diagram loops through recursive functions, available in λ^{req} . This model transformation would allow one to reuse all the λ^{req} tools, including its static machinery, and therefore to rapidly build a working prototype of a service-based application. As usual, a prototype can help in the design phase, because one can perform tests on the system, e.g. by providing as input selected data, one can observe whether the outputs are indeed the intended ones. In λ^{req}, this standard testing practice can be more effective by exploiting the call-by-contract mechanism. For instance, one can perform a request with a given policy φ and observe the resulting plans. This makes the system consider *all* the services that satisfy φ, and the observed effect is similar to running a *class* of tests. To make a concrete example, 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. Standard testing

techniques are not sophisticated enough to spot such kind of bugs. Thus, a designer may find the λ^{req} prototype useful to check the system correctness, since unintended plans provide him with a clear description of the unwanted interactions between services.

Related work

Several approaches have been developed to support verification of service-oriented systems. For example, dynamic bisimulation-based techniques have been adopted to analyse the consistency between orchestration and choreography of services [21], [22], while state-space analysis has been exploited to check correctness of service orchestration [31]. Our approach allows for synthesizing *and* checking the correctness of the orchestration *statically*.

Process calculi techniques have been used to formalize Web Services standards (see e.g. [28], [18], [35], [37], [23]). A different approach is Cook and Misra's Orc [38], an abstract programming model for structured orchestration of services. Web service authentication has been recently modelled and analysed in [14], [15] through a process calculus enriched with cryptographic primitives. The main difference of these approaches with ours stands on the level of abstraction.

Technically, the work of Skalka and Smith [44] is the closest to our framework. We share with them the use of a type and effect system and that of model checking validity of effects (they handle history-based access control). A related line of research addresses the issue of modelling and analysing resource usage. Igarashi and Kobayashi [36] 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 [36], but we provide a static verification technique [9], while [36] does not.

From a software engineering perspective, the advent of service-oriented applications has led to the development of higher level modelling languages which only focus on the service interfaces and orchestration (business) logic and abstract from the underlying programming middleware. A well known example is provided by the Service Component Architecture (SCA) [24]. This framework aims at simplifying implementations, by allowing designers to focus on the business logic only while complying with existing standards. Our approach complements the SCA view providing a full-fledged mathematical framework for designing and verifying properties of service assemblies. It would be interesting develop a (model-transformation) mapping from our formal framework to SCA.

Also related to our approach is the work on SRML [29], a high-level core language, independent from the underlying programming middleware. SRML has a mathematical semantics providing a basis for verification and offers both syntactic and behavioural service interfaces. The logic for the specification of behavioural properties of services is still under development. The interconnections between services are specified in a declarative style, but they are not driven by the properties of a contract as it is our proposal.

Finally, there are several UML extensions, called *UML profiles*, dealing with services. Here, we mention the UML profile for BPEL [33], the UML profile for long-running transactions [50], and the UML profile for λ^{req} [39]. Our graphical design notation basically provides the notion of activity diagram for this UML profile.

Acknowledgments

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

REFERENCES

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proc. 10th Annual Network and Distributed System Security Symposium*, 2003.

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

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

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

[5] A. Banerjee and D.A. Naumann. History-based access control and secure information flow. In *Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS)*, 2004.

[6] H. P. Barendregt et al. Term graph rewriting. In *Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, 1987.

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

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

[9] M. Bartoletti, P. Degano, G.L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *In Proc. Foundations of Software Science and Computation Structures (Fossacs)*, 2007.

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

[11] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Plans for service composition. In *Workshop on Issues in the Theory of Security (WITS)*, 2006.

[12] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Security issues in service composition. In *Invited talk at (FMOODS)*, volume 4037 of *Lecture Notes in Computer Science*. Springer, 2006.

[13] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Types and effects for secure service orchestration. In *Proc. 19th Computer Security Foundations Workshop (CSFW)*, 2006.

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

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

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

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

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

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

[20] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL)*, 2005.

[21] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration: A synergic approach for system design. In *International Conference on Service Oriented Computing (ICSOC)*, volume 3826 of *LNCS*. Springer, 2005.

[22] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration conformace for system design. In *COORDINATION*, volume 4038, 2006.

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

[24] SCA Consortium. Building systems using a service oriented architecture. In *White Paper*. Available from www-128.ibm.com/developerworks/library/specification/ws-sca/, 2005.

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

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

[27] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Secure Internet Programming*, volume 1603 of *Springer LNCS*, 1999.

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

[29] Jose Louis Fiadeiro, Antonia Lopez, and Laura Bocchi. A formal approach to service component architecture. In *WS-FM 2006*, volume 4184 of *LNCS*. Springer, 2006.

[30] P. W. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, 2004.

[31] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web services. In *ASE*. IEEE Computer Society, 2003.

[32] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proc. ACM SIGMOD*. ACM Press, 1987.

[33] T. Gardner and al. Uml 1.4 profile for automated business process with a mapping to the BPEL 1.0. In *White Paper*. IBM Alpha Works, 2003.

[34] Li Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, 1999.

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

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

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

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

[39] C. Montangero and L. Semini. Barbed model–driven software development: A case study. Technical report, SENSORIA, IST-2005-016004, 2007. `http://www.di.unipi.it/~monta/PcU/ttss07REP.pdf`.

[40] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

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

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

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

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

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

[46] Ioan Toma and Douglas Foxvog. *Non-functional properties in Web Services*. WSMO Deliverable, 2006.

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

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

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

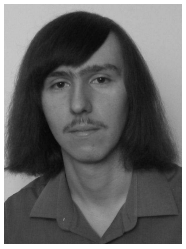
[50] Martin Wirsing et al. Semantic-based development of service-oriented systems. In *Formal Techniques for Networked and Distributed Systems - FORTE 2006*, volume 4229 of *Lecture Notes in Computer Science*. Springer, 2006.

[51] R. Yahalom, B. Klein, and Th. Beth. Trust relationships in secure systems – A distributed authentication perspective. In *Proc. IEEE Symposium on Security and Privacy*, 1993.

Massimo Bartoletti received the PhD degree in Computer Science from the University of Pisa in 2005. His research activity mainly spans over language-based security and static analysis for functional and object-oriented languages. His current research interests include type and effect systems and analysis and design of core calculi for service-oriented computing.

Pierpaolo Degano has been full professor of Computer Science since 1990, and he has been at the Department of Computer Science, University of Pisa since 1993. He served as guest editor of ”Theoretical Computer Science”, the “ACM Computing Surveys”, and “Science of Computer Programming”. He co-founded the IFIP TC1 WG 1.7 on Theoretical Foundations of Security Analysis and Design; he is member of the Board of Directors of the Microsoft Research - University of Trento Center for Computational and Systems Biology. His main areas of interest are security of concurrent and mobile systems, systems biology, semantics and concurrency, methods and tools for program verification and evaluation, and programming tools.

Gian Luigi Ferrari received the PhD degree in computer science in 1989 from the University of Pisa, where he is an Associate Professor at the Department of Computer Science. His research interests include formal specification and verification of concurrent and mobile systems, programming languages for global computing, tool support for mobile systems and theoretical aspects of distributed computing.



Roberto Zunino received the PhD degree in computer science from the University of Pisa in 2006. His main research topics include computer security, cryptographic protocols, systems verification and static analysis. He has worked on automatic verification techniques and on the algebraic properties of cryptographic primitives. Other research interests are language-based security and type systems.