Well-Typed Services Cannot Go Wrong

Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer, Mario Südholt

Well-Typed Services Cannot Go Wrong

Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer, Mario Südholt

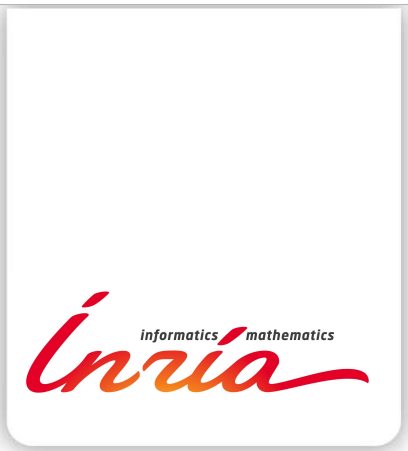
RESEARCH REPORT

N° 7899

May 2012

Project-Teams ASCOLA

ISSN 0249-6399 ISRN INRIA/RR--7899--FR+ENG



Well-Typed Services Cannot Go Wrong*

Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer,
Mario Südholt

Project-Teams ASCOLA

Research Report n° 7899 — May 2012 — 26 pages

Abstract: Service-oriented applications are frequently used in highly dynamic contexts: service compositions may change dynamically, in particular, because new services are discovered at runtime. Moreover, subtyping has recently been identified as a strong requirement for service discovery. Correctness guarantees over service compositions, provided in particular by type systems, are highly desirable in this context. However, while service oriented applications can be built using various technologies and protocols, none of them provides decent support ensuring that well-typed services cannot go wrong. An emitted message, for instance, may be dangling and remain as a ghost message in the network if there is no agent to receive it. In this article, we introduce a formal model for service compositions and define a type system with subtyping that ensures type soundness by combining static and dynamic checks. We also demonstrate how to preserve type soundness in presence of malicious agents and insecure communication channels.

Key-words: Formal Methods, Service-Oriented Computing, Type System, Type Soundness, Message Authentication

*This work has been partially supported by the CESSA ANR project (ANR 09-SEGI-002-01, see `http://cessa.gforge.inria.fr`).

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Pas de message fantôme avec des services bien typés

Résumé : Les applications orientées services (SOA) sont hautement dynamiques car la connaissance et la topologie des services évoluent au cours du temps. La découverte dynamique des services est une facilité importante des SOA, sa flexibilité et sa puissance d'utilisation a été récemment accrue par l'introduction d'une notion de sous-typage. Dans un tel contexte des garanties sur la composition des services est un besoin crucial pour le passage à l'échelle. Les systèmes de types sont connus dans le contexte de la programmation pour permettre certaines garanties minimales. Toutefois, les technologies actuelles des services sont très hétérogènes, et aucune proposition n'a vraiment été faite pour un système de type décent. Un tel système de type doit supporter des canaux de première classe, une relation de sous-typage et assurer qu'un message émis ne restera pas un fantôme dans le réseau de communication. Un message fantôme est un message émis qui ne vérifierait pas le contrat minimal pour être reçu par un agent et se trouve donc à demeurer éternellement dans le réseau. Dans cet article nous introduisons un modèle formel pour des services indépendant de leur orchestration et nous définissons un système de type *sémantique*. Ce modèle avec son système de type assure une propriété de correction : tous les messages émis par un agent pourront être reçus par l'agent destinataire. Nous prouvons également que cette propriété est satisfaite dans des contextes où certains agents ne sont pas fiables et en présence de canaux de communications non sûrs.

Mots-clés : Méthodes formelles, calcul orienté service, système de type, sûreté du typage, authentification des messages

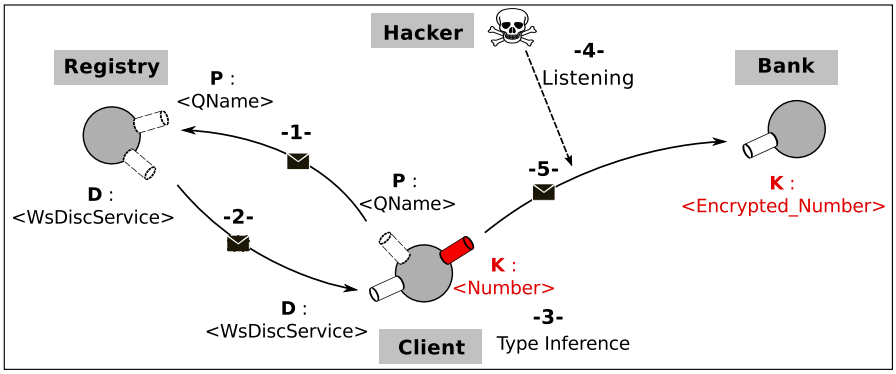


Figure 1: Bank transfer scenario: typing problems entail security issues

1 Introduction

Service-oriented applications are frequently used in highly dynamic contexts: service compositions may be modified and new services be discovered at runtime. Guaranteeing the correctness of service applications in such highly-dynamic contexts is an important open research issue. This problem is compounded by the multiples models and technologies (*e.g.,* WS-* and RESTful models, RPC-XML, JSON-RPC protocols, Apache's CXF and Axis infrastructures) from which SOAs are built, typically in a complex manner that provides only very few guarantees concerning security and safety properties of the resulting software systems.

Type checking is a well-known and proven means to (statically or dynamically) support correctness properties, such as stronger security properties and more efficient program executions, notably for compositional approaches to software development. However, none of the above frameworks or technologies provide decent support for type-based services, such that well-typed services cannot go wrong. For instance, JSON supports typing only in terms of object concepts but not in terms of services. Unfortunately, class subtyping and message subtyping may be incompatible: an object may have, *e.g.,* a single value for each field while a message may define parameters several times (which has been exploited, *e.g.,* for XML injection attacks). The lack of support of type systems for service-oriented systems [11, 12] and a notion of subtyping, in particular, has recently been recognized in several different contexts [8, 6]; type system would be useful, in particular, in order to enable required services to be provided by more specific ones.

Figure 1 shows a realistic scenario that illustrates how typing problems in existing web service standards may entail security issues in the context of communication between a client, say C, and a bank B. B expects data to be encrypted during this communication, *e.g.,* waits for data representing transaction numbers of type `Encrypted_Number`, a subtype of plain numbers `Number`. C may not know the specific service of B and, by default, assume that the service is not security sensitive and may be performed using plain data, *i.e.,* using a channel of type `<Number>`. Using the *WS-Discovery* protocol, C then contacts a registry service R in order to get access, *i.e.,* discover, B's service. Following this standard, C must send an identifier for the bank of type `QName` to R (on the channel `P`, see Fig. 1). Via channel `D`, R returns to C a channel `K` that has been provided by the bank and that should be used for communication between C and B. The *WS-Discovery* standard does not provide channel types: C just gets the information that `K` is of a (generic) channel type that is compatible with sending of plain data, *e.g.,* sending of data of type `Number`. C thus sends unencrypted data B who will recognize the (type) error and inform

[0]Test

C. However, this type error obviously causes a security issue: the unencrypted data initially sent from C to B may be read during transit by some malicious third party.

This kind of typing problem and corresponding security issue can be avoided if R provided a properly typed channel to C. The registry would have to support type-checking with subtyping to enable required services to be provided by more specific ones. This requires, in turn, that contravariant channels are used: if a service with type $< T_1 >$ is required, a type $< T_2 >$ of a service is a correct service provider (*i.e.,* $< T_2 >$ is a subtype of $< T_1 >$) if T_1 is a subtype of T_2. In the above scenario, the registry would send a channel `K` of type $<$ `Encrypted` $>$ to the client who would then recognize a problem: the type of `K` is not a subtype of channel type $<$ `Number` $>$ (which C assumed) because `Encrypted` is a subtype of `Number`. In order to avoid that R sends an ill-typed service, C must send not only the name of the service but also the (correctly typed) channel `D` that is used by the registry to return the service channel to C: the channel `P` must therefore be typed with $<$ `QName` $\times <<$ `Number` $>>>$, the name and type of `D`, instead of just $<$ `QName` $>$; the type of `D`, $<<$ `Number` $>>$, ensures that the channel sent by R has the expected type by C. With this new type information, the registry would then know that C expects to send plain data, which is incompatible with the encrypted communication required by the bank B. Hence, the unencrypted sending of transaction information would be avoided.

While this scenario is set in the context of service discovery using WS-Discovery, the underlying problems are of a general nature.

In this paper, we provide the following contributions:

- A black-box model for service interactions that abstracts from implementation details and differences of existing web service technologies. This model supports message-oriented services in the presence of channel discovery and subtyping. It is formalized in terms of a chemical semantics.
- We define, to the best of our knowledge, the first type system for service interactions that relies on the principles of semantic subtyping [4] and supports subtyping as well as type interference of first-class channels. This type system supports a more regular and general definition of set operators than previous ones, notably Carpineti's and Laneve's system [3].
- We present the extension of the type system in a context with malicious agents. A safe typing requires authentication and we show the correctness of the authentication model, as well as the soundness of the extended type system in the presence of attackers and insecure channels.

This article is structured as follows. In Sec. 2, we introduce our model for message-passing services. Sec. 3 presents a type system for messages that contain structured values. We apply our functional model and the type system to different applications related to common security issues in Sec. 4. Finally, we discuss related work in Section 5 and conclude this article.

2 A model for services with first-order channels

We first define a formal model for service interaction and service composition. Our model belongs to the class of *message-passing models* [7]: agents send and receive messages in a completely asynchronous manner using buffers; they do not share memory. In the absence of failure, messages are eventually delivered but no assumption is made about the time needed for delivery. The network is modeled through the message buffers, which are represented as multisets of messages (*i.e.,* messages are unordered) of arbitrary but finite length. A channel contains one message at a time and determines its (unique) target. Agents also execute in a completely asynchronous

fashion: agents are truly concurrent, each proceeding at its local speed. Exchanging messages is therefore the only way to coordinate agents. Initially, coordination is only possible between agents that share a channel, for instance between a server providing a channel and its clients that share the channel. Gradually, agents may discover new channels since messages can contain channels: channel mobility allows the network topology to evolve.

We abstract from the contents of messages by means of a type system. We assume the existence of a set $\mathcal{V}$ of values, a set $\mathcal{K}$ of channels and a function $\mathrm{K} : \mathcal{V} \to 2^{\mathcal{K}}$ that maps each value v to the set $\mathrm{K}(v)$ of channels occurring in v. The two main requirements of our model, asynchronous communication and true concurrency, have led us to resort to a chemical model [1], which is frequently used for the formalization of such models. In the following, we first define the syntax of the model, which describes processes built from agents, molecules and the solution, called aether, containing them. Semantic rules then describe how the aether evolves in terms of structural rules and reduction rules.

Processes and Agents		
p	$::=$	$a[\sigma][I] \mid p \parallel p$
I	$::=$	$\mathtt{0} \mid I \,\&\, c^{io}$
c^{io}	$::=$	$k^{\iota} \mid k^{o}$
a	$\in$	$\mathcal{A}$
σ	$\in$	Σ
k	$\in$	$\mathcal{K}$

Aether		
Ω	$::=$	$\langle\, \overrightarrow{\mu} \,\rangle$
μ	$::=$	$p \mid a[\sigma] \mid a[c^{io}] \mid a[m^{io}] \mid k(v)$
m^{io}	$::=$	$k^{\iota}(v) \mid k^{o}(v)$

Figure 2: Model syntax

Processes are described on the left-hand side of Figure 2. A process p consists of a parallel composition of agents. An agent has a name a, a state σ and an interface I. The state of agents is kept abstract: their implementation is hidden. Different formalisms, like process algebras, could therefore be used to model agents. An interface declares (the names of) input and output channels. Input channels k^{ι} correspond to the channels provided by the agent: input messages are received on these channels. Output channels k^{o} correspond to the channels that go out of the agent, either as a proper message channel or as part of the contents of a message. A process may obey restrictions on agent names and input channels in order to be well-formed:

Definition 1 (Process well-formedness) *A process must satisfy two rules to be well-formed:*

1. Agent Identification*: given an agent name a, there is at most one agent with name a.*
2. Channel Univocality*: given a channel k, there is at most one occurrence of input channel k^{ι}.*

Agents interact via the (network) *aether* Ω that is defined on the right-hand side of Figure 2. A process that is deployed in the aether is called a molecule. Molecules can be decomposed into smaller ones. While the aether evolves through reduction, (light) mobile molecules appear that represent values transmitted by a channel. Light molecules correspond to output messages emitted by agents, messages in transit and input messages that are to be received by agents. As different messages can have exactly the same form (same channel, same contents), the aether is defined as a multiset of molecules.

Structural Rules. The structural rules given in Fig. 3 essentially describe the decomposition of processes in the aether. Applied from left to right as reduction rules they converge to a

$$
\begin{array}{rcl}
p_1 \parallel p_2 & \rightleftharpoons & p_1 \, , p_2 \\
a[\sigma][I \,\&\, c^{io}] & \rightleftharpoons & a[\sigma][I] \, , a[c^{io}] \\
a[\sigma][\mathsf{0}] & \rightleftharpoons & a[\sigma]
\end{array}
$$

Figure 3: Aether: structural rules

normal form. When used from left to right they are called fission (or heating) rules, used from right to left they are called fusion (or cooling) rules. The structural rules repeatedly reduce parallel processes to molecules until they have been reduced to individual agents. The interfaces of agents are then decomposed, to finally yield agents with their associated states and a multiset of molecules describing the local (input or output) channels of agents. Besides structural rules, we use two standard inference rules that are common to all chemical abstract machines [1], the *reaction law* that allows schemata to be instantiated in an aether, and the *chemical law* that defines how local reactions are fired in an aether.

$$
\frac{\overrightarrow{l} \longrightarrow \overrightarrow{r}}{\langle\, \overrightarrow{l[\sigma]} \,\rangle \longrightarrow \langle\, \overrightarrow{r[\sigma]} \,\rangle} \quad \text{reaction law}
$$

$$
\frac{S \longrightarrow S'}{S \,+\, S'' \longrightarrow S' \,+\, S''} \quad \text{chemical law}
$$

Figure 4: Chemical and reaction laws

The reaction laws states that rules are in fact schemata which can be instantiated as soon as there are in the aether molecules which match the patterns of the rules. The chemical law means that any reaction can be performed freely in any solution.

Reduction rules (see Figure 5) describe the, typically irreversible, evolution of the aether. They assume that the aether has been transformed by structural rules into normal form.

- Rule *[LOC]*: Agent a consumes a, possibly empty, multiset of input messages m^{in}, and produces another, possibly empty, multiset of output messages m^{out}, and updates its state from σ_1 to σ_2.
- Rule *[OUT]*: Agent a sends the message $k(v)$ over the network. A local condition must be met: all the channels occurring in the message ($\{k\} \cup \mathrm{K}(v)$) must be output channels ($\overrightarrow{c^{out}}$) declared by the agent.
- Rule *[IN]*: Agent a receives message $k(v)$ from the network. A local condition must be met: the message channel k must be declared as an input channel by the agent. Moreover, the agent upgrades its declaration of output channels by adding all the channels discovered in the content v of the message ($\mathrm{K}(v)$).

Actually, these three rules correspond to axiom schemata. More precisely, the rules [IN] and [OUT] are true schemata, whereas the rule [LOC] is a generic form for specifying schemata: with each agent comes a set of schemata having the form [LOC] and defining the local behavior of the agent. The axioms are therefore parametrized by the definition of agents.

An error happens when a message is *dangling* in the aether, that is, no agent can receive this message because there is no corresponding input channel. We require a notion of *interface consistency*, which informally states that for each required channel there is a corresponding and compliant provided channel defined by an agent. A first simple result can then be shown: during aether execution no dangling messages occur.

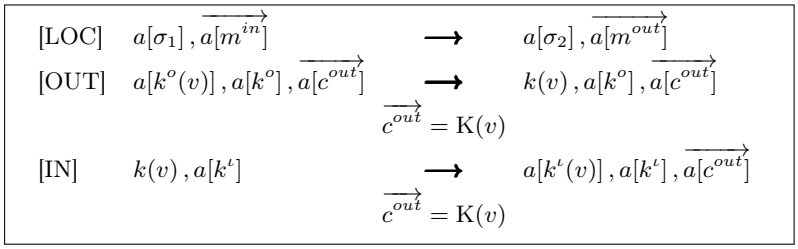


Figure 5: Aether: reduction rules

3 The type system

We now introduce a type system for the values carried by services, *i.e.,* for the untyped model of the previous section. The values are constructed according to the following rules:

$$v ::= b \mid l[v], v \mid k$$

A value v is either a primitive value b or a labeled term $l[v], v$ (that can be used, *e.g.,* to construct sequences of values) or a channel k. The syntax for types is built from value constructors ($l[\_], \_$) and a constructor for channels $< \_ >$. It includes the classic set operations ($+, \wedge$ and $\neg$).

$$t ::= B \mid l[t], t \mid \ < t > \ \mid t + t \mid t \wedge t \mid \neg t \mid \mu X.t \mid X$$

where each base type B denotes a set of values b. Note that (recursive) types may be unfolded infinitely many times, but values are finite. Some recursive types are not constructive (*e.g.,* $\mu X.X$); we therefore consider only guarded types: a constructor $l[\_], \_$ or $< \_ >$ must occur between any binder μX and an occurrence of the variable X.

3.1 Semantic typing relation

Figure 6 defines the typing relation S (and its negation $\overline{S}$) semantically, that is, in terms of set-theoretic concepts [4]. A value b is of type B when it belongs to the set of possible values (rule $Base_1$) and conversely (rule $Base_2$). A labeled value $l[v_1], v_2$ is of type $l[t_1], t_2$ when its subterms are of type t_1 and t_2 (rule Lab_1). It is not of the labeled type when one of its subterms is not of the right type (rules Lab_2 and rule Lab_3) or when the value is not properly labeled (rule Lab_4). A value k is of channel type $< t >$ when k is a channel and its values have the right type (rule Ch_1), where $\Gamma(k)$ denotes the type declared by the channel k. Note that, for the sake of readability, we use a logical implication $\Rightarrow$ in the premise of this rule: actually $\forall v.v \ S \ t \Rightarrow v \ S \ \Gamma(k)$ means $\forall v.(v \ \overline{S} \ t) \vee (v \ S \ \Gamma(k))$. Conversely, a value is not of channel type when there is a mismatch with its declared type (rule Ch_2) or when the value is not a channel (rule Ch_3). Union and intersection of types is defined directly in terms on the corresponding set-theoretic operators. For instance, a value is of type $t_1 + t_2$ if it is of type t_1 or t_2, and conversely it is not of the union type if it is of neither type. Negation of types is defined by negation of the typing relation. Finally, recursive type definitions are unfolded in order to define their semantics. We consider guarded types only: between a location μX and all occurrences of the variable X, there is a constructor $l[\_], \_$ or $< \_ >$. Recursive types thus cannot be unfolded infinitely many times.

$$\frac{b \in B}{b\ S\ B}\ Base_1 \qquad \frac{v \notin B}{v\ \overline{S}\ B}\ Base_2$$

$$\frac{v_1\ S\ t_1 \quad v_2\ S\ t_2}{l[v_1], v_2\ S\ l[t_1], t_2}\ Lab_1 \qquad \frac{v_1\ \overline{S}\ t_1}{l[v_1], v_2\ \overline{S}\ l[t_1], t_2}\ Lab_2$$

$$\frac{v_2\ \overline{S}\ t_2}{l[v_1], v_2\ \overline{S}\ l[t_1], t_2}\ Lab_3 \qquad \frac{v \notin l[v_1], v_2}{v\ \overline{S}\ l[t_1], t_2}\ Lab_4$$

$$\frac{\forall v.v\ S\ t \Rightarrow v\ S\ \Gamma(k)}{k\ S\ <t>}\ Ch_1 \qquad \frac{v\ S\ t \quad v\ \overline{S}\ \Gamma(k)}{k\ \overline{S}\ <t>}\ Ch_2 \qquad \frac{v \notin k}{v\ \overline{S}\ <t>}\ Ch_3$$

$$\frac{v\ S\ t_1}{v\ S\ t_1 + t_2}\ Union_1 \qquad \frac{v\ S\ t_2}{v\ S\ t_1 + t_2}\ Union_2 \qquad \frac{v\ \overline{S}\ t_1 \quad v\ \overline{S}\ t_2}{v\ \overline{S}\ t_1 + t_2}\ Union_3$$

$$\frac{v\ S\ t_1 \quad v\ S\ t_2}{v\ S\ t_1 \wedge t_2}\ Inter_1 \qquad \frac{v\ \overline{S}\ t_1}{v\ \overline{S}\ t_1 \wedge t_2}\ Inter_2 \qquad \frac{v\ \overline{S}\ t_2}{v\ \overline{S}\ t_1 \wedge t_2}\ Inter_3$$

$$\frac{v\ \overline{S}\ t}{v\ S\ \neg t}\ Not_1 \qquad \frac{v\ S\ t}{v\ \overline{S}\ \neg t}\ Not_2$$

$$\frac{v\ S\ t[\mu X.t/X]}{v\ S\ \mu X.t}\ Rec_1 \qquad \frac{v\ \overline{S}\ t[\mu X.t/X]}{v\ \overline{S}\ \mu X.t}\ Rec_2$$

Figure 6: Semantic typing relation S and its negation $\overline{S}$.

Since this typing scheme only defines types for values and is not related to any reduction relation, type soundness in the usual sense does not apply here. However, the type system is complete in the following sense.

Theorem 1 (Completeness of the Type System) *Consider the type system defined in Fig. 6. Then for any value v and any type t, we can derive that value v has type t or that value v does not have type t:*

$$\forall v.\forall t.\ v\ S\ t \vee v\ \overline{S}\ t.$$

Proof. Indeed, it is possible to coinductively build a proof of $v\ S\ t$ or $v\ \overline{S}\ t$. ■

However, as defined, the type system is not consistent. For instance, consider a system with a unique channel k with type t equal to $\mu X. < X >$. It is easy to show that the judgements $k\ S\ t$ and $k\ \overline{S}\ t$ are both valid. To avoid this inconsistency, we modify the type system by adding a constraint (following the formal framework introduced by Brandt and Henglein [2]). Thus, on any path in the proof, the rule Ch_2 may be applied only a finite number of times. Intuitively, a refutation involves a finite number of communications. Formally, we now consider judgements $\Delta \vdash v\ S\ t$, where the environment Δ is a sequence of typing judgements $(k\ S\ t)$ or $(k\ \overline{S}\ t)$ for channels. The environment Δ accounts for the application of rules Ch_1 and Ch_2. Indeed, these

two rules are now defined as follows.

$$\frac{\forall v.(\Delta + (k\ S\ \ < t >)) \vdash v\ S\ t \Rightarrow (\Delta + (k\ S\ \ < t >)) \vdash v\ S\ \Gamma(k)}{\Delta \vdash k\ S\ \ < t >}\ ((k\ S\ \ < t >) \notin \Delta) \quad Ch_1$$

$$\frac{(\Delta + (k\ \overline{S}\ \ < t >)) \vdash v\ S\ t \quad (\Delta + (k\ \overline{S}\ \ < t >)) \vdash v\ \overline{S}\ \Gamma(k)}{\Delta \vdash k\ \overline{S}\ \ < t >}\ ((k\ \overline{S}\ \ < t >) \notin \Delta) \quad Ch_2$$

The environment $(\Delta + J)$ is an extension of Δ with J, defined if J does not occur in Δ. We also add an axiom allowing infinite proofs to be defined for judgements $k\ S\ \ < t >$.

$$\frac{(k\ S\ \ < t >) \in \Delta}{\Delta \vdash k\ S\ \ < t >}\ Ch_1'$$

The absence of a dual axiom for $(k\ \overline{S}\ \ < t >)$ means that the type system forbids the generation of an infinite proof from such a refutation judgment. The whole system is finally obtained by combining the previous three rules, Ch_1, Ch_1' and Ch_2, with the other rules, kept unchanged (except that the environment Δ is passed from the premises to the conclusion). Thanks to rule Ch_1' and the guard condition, the new type system is inductively interpreted.

With the new type system, we can check that $(k\ \overline{S}\ \mu X.\ < X >)$ cannot be proved. The following theorem shows that the type system ensures consistency while preserving completeness.

Theorem 2 (Consistency and Completeness of the Type System) *The extended type system is consistent:*

$$\forall v.\forall t.\ \neg(v\ S\ t \wedge v\ \overline{S}\ t).$$

The extended type system is complete as well:

$$\forall v.\forall t.\ v\ S\ t \vee v\ \overline{S}\ t.$$

Proof. For consistency, we can proceed by induction over the typing judgments to prove that assuming $(v\ S\ t \wedge v\ \overline{S}\ t)$ leads to a contradiction.

For completeness, we can progressively build in parallel the proof of $v\ S\ t$ and $v\ \overline{S}\ t$, in a top-down manner. The proofs are dual, with the same structure. By consistency, the first one succeeds if and only if the second one fails. Now we prove that the proof generation ends with either a success or a failure. Consider a path in the proof in construction, and assume for a contradiction that the path is infinite. Because of the guard condition, the rules $Lab_i;\ i \in \{1, \ldots, 4\}$ or Ch are infinitely often applied. If one of the rules Lab_i is infinitely often applied, the values occuring in the judgements infinitely decrease, contradiction. Otherwise, the rule Ch is applied infinitely often. Since the types are regular and the set of channels is finite, at some rank in the path, the condition $(k\ S\ \ < t >) \in \Delta$ or $(k\ \overline{S}\ \ < t >) \in \Delta$ is met, for some judgement $(k\ S\ \ < t >)$ or $(k\ \overline{S}\ \ < t >)$. This is a contradiction. Finally, thanks to the guard condition, the regularity of types, the finiteness of the set of channels and the definition of rules Ch_1, Ch_1' and Ch_2 in the extended type system, each proof either succeeds or fails. ■

3.2 Algorithms

Two algorithms are needed in order to support our type system: an algorithm that checks whether two types are in a subtyping relation $(\_ \leq \_)$ and a type interference algorithm. Type interference is needed in order to define the types of values, including channels, that are sent over the network

$$t \leq r \iff < r > \leq < t > \quad (1)$$
$$\neg t \leq r \iff < r > \leq \neg < t > \quad (2)$$
$$< \neg t > \leq \neg < t > \quad (3)$$
$$< t_1 > + < t_2 > \leq < t_1 \wedge t_2 > \quad (4)$$
$$< r + \neg t > \leq \neg < t > \wedge < r > \quad (5)$$
$$< t_1 + t_2 > \cong < t_1 > \wedge < t_2 > \quad (6)$$

Figure 7: Channel type equations

3.2.1 Subtyping algorithm

Our algorithm for subtype checking follows Castagna and Frisch [4] who have shown how to derive such an algorithm from a semantic type system. The grammar of our type system is close to theirs. The main difference is that we support labeled lists and channels instead of functions. A channel can be viewed, however, as a function of type $t \longrightarrow \bot$.[1]. Types are interpreted as sets and a disjunctive normal form exists for type expressions. The idea of the algorithm for subtype checking is to express subtyping as set inclusion and use an emptiness test on disjunctive normal forms to decide subtyping. One point is that the complexity of this algorithm is rather bad, exponential in the size of the types. Optimizations to this algorithm that are practically relevant have already been proposed, notably in the context of the *CDuce* language (`http://www.cduce.org`). Another way to optimize this algorithm is to consider restrictions on the types, notably by giving up the full power of set-theoretic type constructors (union, intersection, negation) as done by Carpineti and Laneve [3] who provide a polynomial subtype algorithm. Concretely, the exponential complexity of subtyping of semantic types comes from the availability of the disjoint union operators on different lists that are labelled with the same label. Carpineti and Laneve therefore restrict the grammar to prevent union of types to have the same starting label. But the counterpart is a type language which is not as regular as ours. We do not detail the subtype check any further here but refer the reader to Castagna's and Frisch's work [4] for details. The optimization techniques mentioned above are also applicable to our typing scheme.

3.2.2 Type inference

Type interference is crucial in the context of typed web service if, as in our case, values may be sent without type information or with type information that may have been constructed by a malicious agent. In the remainder of this section we first present several equations that relate channel types and then the type inference algorithms itself whose correctness proof relies on the equations relating channel types.

Some channel type equations. Figure 7 shows some equations that relate expressions involving channel types. Because of channel contravariance (mostly), computations with channel types may be surprising at first sight: union and intersection, for example, are not symmetric and some properties have no equivalent in classic set theory. Furthermore, the subtyping relations

[1] This correspondence supports the need for contravariant channel types.

$[\![t;\, v]\!] =$	b	k	$l'[v_1], v_2$
B	$(b \in B, \{\})$	$(false, \{\})$	$(false, \{\})$
$\neg B$	$(b \notin B, \{\})$	$(true, \{k : \neg B\})$	$(true, \{\})$
$< t >$	$(false, \{\})$	$(true, \{k :< t >\})$	$(false, \{\})$
$\neg < t >$	$(true, \{\})$	$(true, \{k : \neg < t >\})$	$(true, \{\})$
$l[t_1], t_2$	$(false, \{\})$	$(false, \{\})$	$(l = l', \{\}) \wedge$ $[\![t_1;\, v_1]\!] \wedge [\![t_2;\, v_2]\!]$
$\neg l[t_1], t_2$	$(true, \{\})$	$(true, \{k : \neg l[t_1], t_2\})$	$(l = l')?$ $[\![\neg t_1;\, v_1]\!] + [\![\neg t_2;\, v_2]\!]$ $: (true, \{\})$
$\neg\neg t$	$[\![t;\, v]\!]$		
$t_1 + t_2$	$[\![t_1;\, v]\!] + [\![t_2;\, v]\!]$		
$\neg(t_1 + t_2)$	$[\![\neg t_1;\, v]\!] \wedge [\![\neg t_2;\, v]\!]$		
$t_1 \wedge t_2$	$[\![t_1;\, v]\!] \wedge [\![t_2;\, v]\!]$		
$\neg(t_1 \wedge t_2)$	$[\![\neg t_1;\, v]\!] + [\![\neg t_2;\, v]\!]$		
$\mu X.t$	$[\![un(\mu X.t);\, v]\!]$		
$\neg\mu X.t$	$[\![\neg un(\mu X.t);\, v]\!]$		

Figure 8: Inference algorithm

can normally not be replaced by equalities. Note that, in the following proofs we use only the original rules Ch_1 and Ch_2 but our deductions are sound since we never apply Ch_2 an infinite number of times.

In order to prove the above relationships, the premises of rules Ch_1 and Ch_2 from Fig. 6 can be expressed as, respectively, one of two conditions:

- $t \leq \Gamma(k)$ or $t \wedge \neg\Gamma(k) = \bot$ (which shows the contravariant effect)
- $\neg(t \leq \Gamma(k))$ or $t \wedge \neg\Gamma(k) \neq \bot$

Here, $t \leq \Gamma(k)$ and $\neg(t \leq \Gamma(k))$ simply are reformulations of the subtype relationships; the other two terms simply correspond to tests whether the intersection of t and $\Gamma(k)$ is empty or not (remember that semantic typing schemes define subtyping as subset relationships).

The proofs of the properties in Figure 7 are straightforward using these four auxiliary properties. For instance, in order to prove the subtyping relation (3) we observe that the left part is equivalent to $\neg t \leq \Gamma(k)$, thus equivalent to $t \geq \neg\Gamma(k)$, and therefore implies $t \wedge \neg\Gamma(k) \neq \bot$, a reformulation of the right-hand side.

The inference algorithm. In the following we define an algorithm that enables correct types of transmitted values to be inferred by receiving agents. The algorithm principle is similar to pattern matching: in our case a value v is used as a pattern that filters a type t. Type inference is defined by an inductively defined function $[\![\_;\, \_]\!]$ given in Fig. 8. This function has type $\mathit{Value} \times \mathit{Type} \longrightarrow \mathit{boolean} \times \mathit{Context}$ takes an argument, a pair of a value and a type, and

returns a pair consisting of a boolean that indicates whether the type matches the value and an inferred context, a mapping from channels to types.[2] In the following, we use the notations *succeed* and *context* for the projections or the result pair $boolean \times Context$. The notation $context(\llbracket \_ ; \_ \rrbracket)(k)$ denotes the type of k inferred by the algorithm.

Types (that appear in the leftmost column of Fig. 8) are closed and must respect the guarded type constraint. Values (see the top line of Fig. 8) are finite and represented by closed terms. Failure is noted $(false, \{\})$, where $\{\}$ denotes the empty context. Operations like intersection and union are promoted to inferred contexts as follows. A union fails if both of its arguments fail while an intersection fails if one of its argument fails. If both arguments of a union or intersection match successfully, the union of the resulting bindings is caculated, whereby bindings occurring in both arguments are merged as defined below. The notation $un(\_)$ stands for the unfolding of a recursive type and $\_?\_:\_$ is a shortcut for $if\ \_then\ \_\ else\ \_$.

The intersection ($\wedge$) and union ($+$) of inferred contexts with merging are formally defined as follows.

Definition 2 (Merging inferred contexts)

$$dom(\Theta_1 \wedge \Theta_2) = dom(\Theta_1 + \Theta_2) = dom(\Theta_1) + dom(\Theta_2)$$

$$\begin{array}{lcl}
\forall k\ .\ k \in (\ dom(\Theta_1) \wedge \ dom(\Theta_2)) & \Longrightarrow & (\Theta_1 \wedge \Theta_2)(k) = \Theta_1(k) \wedge \Theta_2(k) \\
\forall k\ .\ k \in (\ dom(\Theta_1) \wedge \neg dom(\Theta_2)) & \Longrightarrow & (\Theta_1 \wedge \Theta_2)(k) = (\Theta_1 + \Theta_2)(k) = \Theta_1(k) \\
\forall k\ .\ k \in (\neg dom(\Theta_1) \wedge \ dom(\Theta_2)) & \Longrightarrow & (\Theta_1 \wedge \Theta_2)(k) = (\Theta_1 + \Theta_2)(k) = \Theta_2(k) \\
\forall k\ .\ k \in (\ dom(\Theta_1) + \ dom(\Theta_2)) & \Longrightarrow & (\Theta_1 + \Theta_2)(k) = \Theta_1(k) + \Theta_2(k)
\end{array}$$

It is easy to see that $dom(context(\llbracket t; v \rrbracket)) \subseteq K(v)$. This inclusion may be strict for some v: sometimes the above algorithm infers partial types for channels because it does not dispose of precise-enough information. For instance, for the application $\llbracket \neg int; l[k], k \rrbracket$ the value $l[k], k$ is matched by the type $\neg int$ but it is not possible to assign a type to the channel k. The above algorithm may also assign a non-usable type to a channel, for instance $k : \neg bool$. In such cases, the algorithm cannot be effective since we have no positive information about which type of data circulates on the channel. The algorithm can also build types for channel like $< B_1 > + < B_2 >$ or $< B_1 > \wedge < B_2 >$ which should be interpreted depending on the base type semantics. Finally, there are some cases where the inferred channel is "empty." Consider, for instance, $\llbracket < int > + < bool >; k \rrbracket$ then $k :< int > + < bool >$ which is a subtype of $k :< int \wedge bool >$ and we can expect that this intersection is empty. The principles of subtyping checking used in [4] can be applied to solve this issue.

We now turn to the correctness proof of our type inference algorithm. First, we prove its termination. We then show two properties, Propositions 2 and 3, that establish the compatibility of type inference and the typing checking system $S, \bar{S}$.

Proposition 1 (Termination of type inference) *The function $\llbracket t; v \rrbracket$ terminates for all input values t and v.*

Proof. By contradiction. If the type interference function does not terminate, it is applied an infinite number of times. The corresponding infinite sequence of applications cannot be an infinite chain of consecutive calls with unfolding due to the guardedness condition. Furthermore, the cases of union and intersection can only be performed a finite number of times because the structural complexity of the corresponding type argument t is strictly decreasing. In all remaining cases the structural complexity of the value argument v in the sequence of applications strictly decreases. Thus no infinite chain of applications of the type inference function exists. ■

[2] Note that the inferred context is different from the typing context Γ used for type checking before, see below for a detailed discussion.

The following property shows that type inference is, if it yields a matching type, equivalent to type checking using the system $S, \bar{S}$.

Proposition 2 (Equivalence of type inference and checking)

$$\forall v, t \;.\; succeed([\![\, t;\, v \,]\!]) \iff \exists \Gamma \;.\; \Gamma \vdash v \; S \; t$$

See Section A for the proof.

In order to ensure later on that service compositions are correctly typed we have to ensure that the declared types of channels that are type checked are included in the types of channels that are inferred. This is formalized by a notion of containment of inferred and typing contexts. Note that this entails, due to channel contravariance, that type checking assigns types to transmitted values that are larger than the types inferred for these values.

Definition 3 (Context containment) *Let Θ be an inferred context and Γ a typing context. The containment relation is then defined by:*

$$\Theta \;\leq\; \Gamma \quad \textit{iff} \quad \forall k \;.\; k \in dom(\Theta) \implies < \Gamma(k) > \leq \; \Theta(k).$$

The following equivalence property that links type checking and inference shows that the context defined by type inference is minimal compared to the contexts used for type checking.

Proposition 3 (Typing and inferring)

$$\forall v, t \;.\; \forall \Gamma \;.\; (\Gamma \vdash v \; S \; t \iff (succeed([\![\, t;\, v \,]\!]) \;\wedge\; context([\![\, t;\, v \,]\!]) \;\leq\; \Gamma))$$

See Section A for the proof.

Another property links (successful) inference and subtyping

Proposition 4 (Successful inference and subtyping)

$$\forall t, v \;.\; t \;\leq\; r \;\wedge\; succeed([\![\, t;\, v \,]\!]) \implies succeed([\![\, r;\, v \,]\!]).$$

Proof. Consider the semantic typing definition $t \;\leq\; r \implies (v \; S \; t \implies v \; S \; r)$. If $succeed([\![\, t;\, v \,]\!])$, Prop. 2 yields a typing context $\exists \Gamma \;.\; \Gamma \vdash v \; S \; t$ that is also a typing context for r Prop. 2 then implies that inference succeeds for r. ■

Our inference algorithm infers correct channel type:

Proposition 5 (Correctness of channel type inference)

$$\forall t, k \;.\; succeed([\![\, t;\, k \,]\!]) \implies context([\![\, t;\, k \,]\!])(k) = t$$

Proof. We proceed by induction on the type t. The ases for B, $l[t_1], t_2$ and $< t_1 >$ and their negations satisfy the proposition straightforwardly. The case $\neg\neg$ is solved by induction. For union and intersection we apply the induction hypothesis and the corresponding set operations on the inferred types. For recursion, we unfold a finite number of times until one of the other cases is attained and then apply the induction hypothesis. ■

Finally, we show that inference is monotonic with respect to subtyping, another property that is crucial for the correctness of web service compositions.

Definition 4 (Inferred context subtyping) *Let Θ_t and Θ_r two inferred contexts,* inferred context subtyping *is defined as $\Theta_t \;\leq\; \Theta_r$ iff $\forall k \;.\; k \in dom(\Theta_t) \implies \Theta_t(k) \;\leq\; \Theta_r(k)$.*

Note that the following property holds: $(\forall \Gamma \,.\, \Gamma \geq \Theta_t \implies \Gamma \geq \Theta_r) \implies (\Theta_t \leq \Theta_r)$.

Proposition 6 (Monotonicity of inference w.r.t. subtyping)

$$\forall t, r, v \,.\, (t \leq r \wedge \mathit{succeed}([\![t; v]\!])) \implies (\mathit{context}([\![t; v]\!]) \leq \mathit{context}([\![r; v]\!]))$$

Proof. If $t \leq r$, we know from Prop. 4 that $\mathit{succeed}([\![t; v]\!])$ implies $\mathit{succeed}([\![r; v]\!])$. If Γ contains the inferred context for t from the Prop. 3 it types v and from semantic typing it is also a typing context for r. Using Prop. 3, this context contains the inferred context for r and this shows that inferred contexts are in subtyping relation as required above. ■

4 Application to message-oriented services

This section presents an application of our service interaction model and its type system to several issues related to the protection of agents that may communicate channels within messages. We first show that the type system ensures that all messages can always be received by some agent, *i.e.,* that messages don't get stuck. We then consider networks in which some agents may engage in malicious activity or use insecure channels while other parts of the network are trustworthy agents. In this context, we define a message authentication scheme and show how messages can be protected so that malicious agents cannot tamper with message types. Finally, we prove that all messages sent by a controlled agent to another controlled agent will be delivered correctly.

4.1 Type checking messages

In a first step we show type soundness in the presence of benign agents and secure channels. If a message is received that does not contain a channel that is unknown at the receiving site, a type-check performed during emission is sufficient to ensure correct delivery. However, since services may be discovered dynamically, the type of unknown channels must be inferred using a type inference system, such as that presented in Sec. 3. Structural subtyping for types has to be used since this is important for interoperability as explained in [8]. Furthermore, it is necessary to check that provided and required interfaces are compliant. We ensure compliance using the subtyping algorithm that is part of our type system.

To type check service interactions we enrich our model of service interactions introduced in Sec. 2 with type information and declarations: interfaces are enriched with type declarations for channels and Θ is used to refer to inferred context as in introduced in the previous section. We consider an additional (third) property for well-formed process (see Def. 1): a sequence $\overrightarrow{c^{out}}$ of typed output channels of a well-formed process defines a functional relation from channels to types. Thus we see it as a local context and we will note Θ_e any partial context of a well-formed process.

Practically, we need a checking algorithm and an inference algorithm since we should distinguish emission of a value and its receipt. Application of the inference algorithm will be noted $[\![t; v]\!] = (b, \Theta_i)$, where b is a boolean and Θ_i the inferred context. The type checking is realized with inference and subtyping, a more dedicated algorithm is possible but it is an implementation choice. In order to formalize interface consistency and type soundness, we first define the notion of receivable and dangling messages. Informally, a message is receivable in aether Ω if an agent exists with a suitably typed input channel.

Definition 5 (Receivable, dangling message) *A message $k(v)$ is* receivable *iff:*

$$\exists\, a, t, \Theta_i \,.\, a[k^{\iota}{:}{<}t{>}] \in \Omega \;\wedge\; [\![t; v]\!] = (\mathit{true}, \Theta_i).$$

$$\text{[OUT]} \quad a[k^o(v)]\,,a[k^o{:}{<}t{>}]\,,a[\Theta_e] \longrightarrow k(v)\,,a[k^o{:}{<}t{>}]\,,a[\Theta_e] \quad [\![\, t;\, v \,]\!] = (true, \Theta_i)\,, \Theta_e \;\leq\; \Theta_i$$

$$\text{[IN]} \quad k(v)\,,a[k^\iota{:}{<}t{>}]\,,a[\Theta_e] \longrightarrow a[k^\iota(v)]\,,a[k^\iota{:}{<}t{>}]\,,a[\Theta_e \;\wedge\; \Theta_i] \quad [\![\, t;\, v \,]\!] = (true, \Theta_i) \;\wedge\; dom(\Theta_e) \;\subseteq\; dom(\Theta_i)$$

Figure 9: Communication with type inference

A message is dangling *if it is not receivable.*

We are interested in systems in which only receivable messages are sent. to this end, we use a notion of interface consistency that ensures that agent interfaces are appropriately typed. Interface consistency states that all messages sent on an output channel are receivable on a corresponding input channel modulo subtyping, a condition similar to type-based interface compatibility for software components [13]. The principle is that the type of an output channel must be a supertype of its declared input channel.

Definition 6 (Interface Consistency)

$$\forall\, a,\ \sigma,\ I\,.\ a[\sigma][I] \in \Omega \;\wedge\; \forall\, k,\ t\,.\ k^o{:}\, t \in I \implies \exists\, a',\ \sigma',\ I'\,.\ a'[\sigma'][I'] \in \Omega \;\wedge\; \exists\, t'\,.\ k^\iota{:}\, t' \in I' \;\wedge\; (a \neq a') \;\wedge\; (t' \;\leq\; t)$$

In order to avoid dangling messages, we add static and dynamic type checks to the original aether reduction rules of Fig. 5. First, a static check evaluates interface consistency for each process: each output channel declared must be associated to a corresponding input channel. Second, two dynamic checks are added to the communication rules [OUT] and [IN]. They ensure that the emitted value and the received value are well-typed. Moreover, they allow the type of the discovered channels to be correctly inferred at the receiving site. Dynamic checking is required since we are orchestration independent. In case we assume a particular type safe orchestration language this checking can be statically ensured.

The corresponding modified communication rules [OUT] and [IN] are given in Fig. 9. Let Θ_e be the local context of an emitter agent, v the emitted value, t the type of information of the outgoing channel. At emission time, the emitter uses its local context to check the type of the emitted value before putting it on the out channel. This checking is performed by first inferring Θ_i and then checking the subtype relation with Θ_e. More formally, $[\![\, t;\, v \,]\!] = (true, \Theta_i) \;\wedge\; \Theta_e \;\leq\; \Theta_i$. At receipt, the agent will infer a context and add it to its local context. However, it can type new channel as well as existing channel values. Thus we add the condition $[\![\, t;\, v \,]\!] = (true, \Theta_i) \;\wedge\; dom(\Theta_e) \;\subseteq\; dom(\Theta_i)$ and the new channel context is $\Theta_e \;\wedge\; \Theta_i$. Note that these rules require the use of an explicit channel type to perform the communication.

These communication rules with typing conditions ensure interface consistency in the presence of channel discovery.

Proposition 7 (Interface consistency invariant) *Interface consistency is preserved by channel discovery at receipt.*

Proof. Fig 10 describes the general case related to the interface consistency preservation. Consider that a_e emits a well typed message with value v which contains a channel k known or

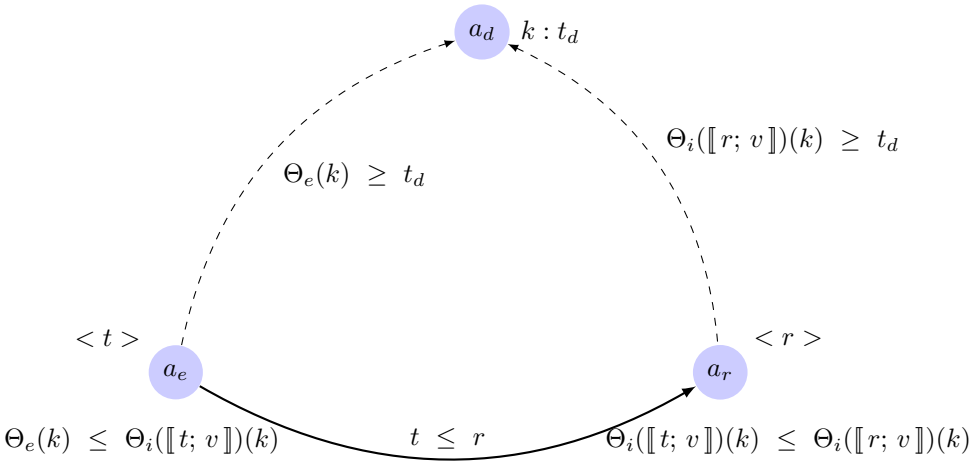


Figure 10: Interface consistency preservation

unknown by the receiver agent a_r. At receipt we assume that inference succeeds. If $k \notin dom(\Theta_i)$ is not discovered by the receiver agent then interface consistency is preserved. If k is defined at receiver side ($k \in dom(\Theta_i)$), the property 6 implies that it is also defined at the emitter side and with a smaller type.

At emission, the type of k is provided by the local context Θ_e of the emitter agent a_e. The interface consistency property states that this channel was defined by a, possibly different, agent a_d as a channel with type $\Theta_e(k) \geq t_d$. Let t be the type of this message and $\Theta_i([\![t; v]\!])(k)$ the inferred type of the transmitted channel. From the checking conditions in rule [OUT] we have the constraint $\Theta_i([\![t; v]\!])(k) \geq \Theta_e(k)$. The type of the message received by a_r is r, but interface consistency and channel contravariance imply that $t \leq r$. At reception time at a_r, the type inference mechanism initially tags the transmitted channel with, yet another, type $\Theta_i([\![r; v]\!])(k)$. By the monotonicity property of our type inference algorithm, see Property 6, $\Theta_i([\![r; v]\!])(k) \geq \Theta_i([\![t; v]\!])(k)$ follows. This channel may be new to a_r or it may be already known ($k \in dom(\Theta_e)$); in the latter case $t_k \geq t_d$ holds because of interface consistency. The channel k was defined by a_d and we have either $\Theta_i([\![r; v]\!])(k) \geq t_d$ or $(\Theta_i([\![r; v]\!])(k) \wedge t_k) \geq t_d$ thus interface consistency is satisfied for the channel k for agent a_r. ■

Finally, our system enjoys a type soundness that ensures that every message in transit is receivable, which, in turn, entails progress and preservation of message-oriented service interactions.

Theorem 3 (Type soundness) *If processes are well-formed and interfaces consistent, the following property holds:*

$$\forall\, k,\ v\,.\ k(v) \in \Omega \implies k(v) \text{ is receivable.}$$

Proof. The proof relies on two invariants of the aether: process well-formedness and interface consistency. The proof that processes remain well-formed is trivial and preservation of the second invariant follows from Prop. 7. Consider a message $k(v)$ that is in transit: it has been inserted in the aether using the [OUT] rule. There is a channel with type $k^o{:}{<}t_e{>}$ and the message was thus type checked within the context of the emitter, *i.e.,* $[\![t_e; v]\!] = (true, \Theta_i)$. Due to the interface consistency invariant, there is an input channel $a_r[k^\iota{:}{<}t_r{>}]$ and contravariance of channel types

implies $t_e \leq t_r$, thus we have $[\![t_r; v]\!] = (true, \Theta_i)$ from Prop. 4. The message is thus receivable according to Def. 5. ■

4.2 Weak message authentication

We now consider an extension of our typing scheme to aethers that may contain uncontrolled agents and monitors (monitors are always trusted). We strive for a provably correct authentication mechanism for systems involving uncontrolled and controlled agents. We consider a weak message authentication mechanism[3] that only enables the status (monitored or not) of each message emitter to be ascertained.

In order to formulate authentication properties, we distinguish between emitted and received messages. In general, messages may be of very different forms because various pieces of information may be added for authentication purposes and uncontrolled agents can forge any kinds of message. However, we can limit ourselves to messages that are of the following two forms: *i)* outgoing messages: $k(v, n, sv)$, and *ii)* incoming messages: $k(v, b)$, where v is a value, b a boolean indicating the status of the emitter (monitored or not), n a nonce and sv a tag (a sticky value). A monitor strictly respects the message format but an uncontrolled agent may not. Any message which does not conform to the above format is simply removed from the solution.

In the following, we develop the authentication mechanism in two steps, first considering systems that only use secure channels and then presenting the general case that includes insecure channels.

4.2.1 Systems with secure channels only

As a first step we consider systems in which all channels are secure and do not modify messages. The weak authentication protocol is defined in terms of the monitors (set $\mathcal{M}$) and the uncontrolled agents (set $\mathcal{U}$) by the rules respectively given in Figures 11 and 12. Uncontrolled agents may modify messages (we denote such messages using the notation $*(*)$): their values, channels and argument values may change arbitrarily. Furthermore, we assume that an uncontrolled agent knows all the provided channels in the aether. Since all channels are secure, nonces are not used in this step.

Monitors share a secret `s` information, which is assumed to be *computation-resistant* [10]. At each instant a monitor can use `s` in its messages, but other agents cannot use it and furthermore they cannot compute it from the received information. The set of tag values is defined as $TagValue = \{\mathtt{s}\} + \{\bot\}$. In Figure 11, four rules define the new behavior of a monitor when sending messages to, or receiving messages from, other monitors or uncontrolled agents ($\mathcal{M}$ and $\mathcal{U}$ in the rule names). The first modification is that a message from a monitor to a monitor incorporates the secret, and at receipt by a monitor the presence of the secret is checked.

To prove the correctness of this authentication scheme we have to compare the status inferred from the checking of the secret, *i.e.,* the status component in the messages, with the true status of the emitter, *i.e.,* the superscript boolean values. All messages, emitted and received ones, are annotated either with *true* denoting a controlled emitter or *false* that characterizes an uncontrolled one.

Hence, the rules take into account the status computation as follows: *i)* $[\mathrm{OUT}]_{\mathcal{M}\to *}$ the status of the emitted message is *true* since the emitter is a monitor, *ii)* $[\mathrm{OUT}]_{\mathcal{U}\to *}$ the status of the emitted message is *false*, and *iii)* $[\mathrm{IN}]_{*\to *}$ keep the status of the received message.

[3]More precisely, we support data origin authentication [10] that guarantees integrity of the origin of messages and their uniqueness.

$$[\text{OUT}]_{\mathcal{M}\to\mathcal{M}} \quad a[k^o(v)]\,, a[k^o{:}<t>]\,, a[\Theta_e] \longrightarrow k(v, \_, \mathtt{s})^{true}\,, a[k^o{:}<t>]\,, a[\Theta_e] \quad a \in \mathcal{M}\,, [\![\, t;\, v \,]\!] = (true, \Theta_i)\,, \Theta_e \;\leq\; \Theta_i$$

$$[\text{OUT}]_{\mathcal{M}\to\mathcal{U}} \quad a[k^o(v)]\,, a[k^o{:}<t>]\,, a[\Theta_e] \longrightarrow k(v, \_, \bot)^{true}\,, a[k^o{:}<t>]\,, a[\Theta_e] \quad a \in \mathcal{M}\,, [\![\, t;\, v \,]\!] = (true, \Theta_i)\,, \Theta_e \;\leq\; \Theta_i$$

$$[\text{IN}]_{\mathcal{M}\to\mathcal{M}} \quad k(v, \_, \mathtt{s})^{true}\,, a[k^\iota{:}<t>]\,, a[\Theta_e] \longrightarrow a[k^\iota(v, true)^{true}]\,, a[k^\iota{:}<t>]\,, a[\Theta_e \wedge \Theta_i] \quad a \in \mathcal{M}\,, [\![\, t;\, v \,]\!] = (true, \Theta_i) \wedge dom(\Theta_e) \subseteq dom(\Theta_i)$$

$$[\text{IN}]_{\mathcal{U}\to\mathcal{M}} \quad k(v, \_, \bot)^{false}\,, a[k^\iota{:}<t>] \longrightarrow a[k^\iota(v, false)^{false}]\,, a[k^\iota{:}<t>] \quad a \in \mathcal{M}$$

Figure 11: Secure channels, monitors

$$[\text{OUT}]_{\mathcal{U}\to *} \quad a[*^o(*)] \longrightarrow *(*)^{false} \quad a \in \mathcal{U}$$

$$[\text{IN}]_{*\to\mathcal{U}} \quad *(*)^{true|false} \longrightarrow a[*^\iota(*)^{true|false}] \quad a \in \mathcal{U}$$

Figure 12: Secure channels, uncontrolled agents

Informally, correctness of the authentication scheme is defined by the following property: for each message received from a monitor, the real status is the same as the computed one.

Theorem 4 (Weak Message Authentication)

$$\forall\, a \in \mathcal{M}\,.\, \forall\, k, v,\ b,\ r,\ st\,.\, a[k^\iota(v, b)^{st}] \in \Omega \implies b = st$$

Proof. The authentication protocol is expressed by the rules in Figures 11 and 12. Considering the incoming message $a[k^\iota(v, b)^{st}]$, where $a \in \mathcal{M}$, two subcases have to be handled depending on the value of b. If $b = true$ then the message was received according to the rule $[\text{IN}]_{\mathcal{M}\to\mathcal{M}}$ and the transmitted message was $k(v, n, \mathtt{s})^{st}$ with $\neg(\mathtt{s} = \bot)$. The only applicable rule then is $[\text{OUT}]_{\mathcal{M}\to\mathcal{M}}$, the emitter was a monitor and $st = true$. If $b = false$ then the rule $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$ was used, and the rule $[\text{OUT}]_{\mathcal{U}\to\mathcal{M}}$ emitted the message, thus $st = false$. ■

4.2.2 Systems with insecure channels

We now generalize the authentication scheme to systems with insecure channels. An insecure channel between two monitors is equivalent to the transmission of messages via uncontrolled agents which act as malicious routers. The challenge is to force the uncontrolled agents to act with integrity. When a message from a monitor reaches an uncontrolled agent, the latter may suppress it or send another message to monitors or uncontrolled agents. We use hash keys in messages to be able to detect modifications to messages. The behavior of monitors has to be extended so that messages going to or coming from uncontrolled agents are checked for integrity

$$[\text{OUT}]_{\mathcal{M}\to\mathcal{U}} \quad \begin{array}{r} a[k^o(v)]\,,a[k^o{:}<t>,(n)]\,, \\ a[{k_r}^o{:}<t_r>]\,,a[\Theta_e]\,, \\ r[{k_r}^\iota{:}<t_r>]\,,m[k^\iota{:}<t_m>] \end{array} \longrightarrow \begin{array}{r} k_r(v,n,\mathtt{H}(v,k,n))^{true}\,,a[k^o{:}<t>,(n+1)]\,, \\ a[{k_r}^o{:}<t_r>]\,,a[\Theta_e]\,, \\ r[{k_r}^\iota{:}<t_r>]\,,m[k^\iota{:}<t_m>] \end{array}$$

$$a,m\in\mathcal{M}\,,r\in\mathcal{U}\,,[\![\,t;\,v\,]\!]=(true,\Theta_i)\,,\Theta_e\ \leq\ \Theta_i$$

$$[\text{IN}]_{\mathcal{U}\to\mathcal{M}} \quad \begin{array}{r} k(v,n,\mathtt{H}(v,n,k))^{true}\,, \\ a[k^\iota{:}<t>,(N)]\,,a[\Theta_e] \end{array} \longrightarrow \begin{array}{r} a[k^\iota(v,true))^{true}]\,, \\ a[k^\iota{:}<t>,(N\cup\{n\})]\,,a[\Theta_e\ \wedge\ \Theta_i] \end{array}$$
$$a\in\mathcal{M}\,,n\notin N\,,[\![\,t;\,v\,]\!]=(true,\Theta_i)\ \wedge\ dom(\Theta_e)\ \subseteq\ dom(\Theta_i)$$

$$[\text{IN}]_{\mathcal{U}\to\mathcal{M}} \quad k(v,n,h)^{false} \underset{a\in\mathcal{M}\,,\neg(h=\mathtt{H}(v,n,k))}{\longrightarrow} a[k^\iota(v,false)^{false}]$$

$$[\text{OUT}]_{\mathcal{U}\to *} \quad a[*^o(*)] \underset{a\in\mathcal{U}}{\longrightarrow} k(*)^{h=\mathtt{H}(v,n,k)}$$

$$[\text{IN}]_{*\to\mathcal{U}} \quad k(*)^{true|false} \underset{a\in\mathcal{U}}{\longrightarrow} a[*^\iota(*)^{true|false}]$$

Figure 13: Weak Message Authentication Rules – Insecure Channels

violations. We also need to review the status computation. The new rules should express the specific behavior of routers, which are not controlled but: *i)* receive messages with correct hash keys from monitors, and *ii)* if the hash key is copied with the right values, the status component of the outgoing message is the true status, or else it is *false*. We use nonces to avoid that a same message will be received several times. Otherwise, agents could replay messages. This implies that the syntax definition has to be modified: emitted messages now have a mandatory integer argument for nonces. Furthermore, the context of an agent stores the known channels with their nonces.

We assume that monitors share a secret cryptographic function making the secrets unforgeable by uncontrolled agents. We need to distinguish the hashed values from the secret $\mathtt{s}$ and $\bot$. The set of tag values is thus redefined to $TagValue = \{\bot\}\ +\ \{\mathtt{s}\}\ +\ range(\mathtt{H})$. Figure 13 adds two new rules related to insecure message transit and replaces the previous rules for uncontrolled agents, see Figure 12, to take into account the modified status computation.

Rule $[\text{OUT}]_{\mathcal{M}\to\mathcal{U}}$ allows a monitor a to send a message to another monitor m via an uncontrolled agent r owning the channel k_r. To do this a hash value $H(v,k,n)$ is added to the message. Rule $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$ formalizes the case where the hash key value received by a monitor is compliant with the channel k, the value v and the nonce n. Rule $[\text{OUT}]_{\mathcal{U}\to *}$ specifies that the original emitter was a monitor if the hash value is correct. The remaining rules are immediate.

The correctness of the extended authentication scheme is formulated as a theorem analogous to the first scheme for secure channels. Its proof has to take into account that messages may be forwarded by uncontrolled routers, see Figures 11 and 13. Hence, it is possible that messages with hash keys are routed in complex manners within the uncontrolled world. However, each sequence resulting from such message routing is finite and starts from a monitor.

Proposition 8 (Routing Control) *Any message sequence in the aether* $\forall\ k_i(v_i,n_i,\mathtt{H}_i)$ *that is temporally ordered, such that* $\exists\,H\,.\ \forall\,i\,.\ \mathtt{H}_i=H$*, and that ends in a monitor,* i.e., *a controlled*

agent, is finite and starts from a monitor.

Proof. If such a message reaches a monitor then it comes from an uncontrolled agent, since monitors are trustable and cannot send hash key values to other monitors. Now if an uncontrolled agent has received a hash key value it comes in one step from a monitor or it comes from another uncontrolled agent. Hash key values are not forgeable by uncontrolled agents, agents can only pass them to others. Any hashed message has to be initially emitted from a monitor and intermediate nodes cannot forge but always relay the hash value. ■

A consequence is that all the messages in the sequence have status *true*. Furthermore, messages with hash key are received at most once due to the nonce mechanism. Overall, this scheme amounts to the use of so-called time-variant parameters [10].

While weak mesage authentication (Theorem 4) still holds in the presence of uncontrolled agents, its proof has to be adapted.

Proof. (of Theorem 4 in the presence of uncontrolled agents)
Consider a message that enters a monitor $a[k^\iota(v,b)^{st}]$, where $a \in \mathcal{M}$, $k \in \mathcal{K}(a)$.

- If $b =$ *true* the accepting rule was $[\text{IN}]_{\mathcal{M}\to\mathcal{M}}$ and we are in the same situation as with only secure channels. The accepting rule might also have been $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$. In this case the monitor receives a correct hashed key message $k(v, n, \mathtt{H}(v, n, k))$. Property 8 states that the original hash value was emitted by a monitor and the status is *true*.
- If $b =$ *false* either the rule was $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$ but this rule exists in two instances: one with $\perp$ and the other with a hash key h. The first case is the same as above without insecure channel. In the second case h is not a correct hash key value, thus it was put in the aether by $[\text{OUT}]_{\mathcal{U}\to *}$ and then the status is *false*.

■

4.3 Type soundness

In the previous section we have introduced two kinds of attackers: uncontrolled agents and insecure channels. A countermeasure to possible attacks on types is to add control at reception time during channel discovery.

Definition 7 (Origin-check) *At message reception time, the type inference mechanism checks the origin of each message (using weak message authentication). If the message was sent by an uncontrolled agent, no discovery of channels is allowed.*

This countermeasure is formalized in the rules $[\text{IN}]_{\mathcal{U}\to\mathcal{M}}$ in Figures 11 and 13. When the computed status is *false*, no channel discovery is performed.

Theorem 5 (Type soundness with attackers) *In presence of uncontrolled agents and insecure channels, the origin-check countermeasure ensures that messages in transit that have been sent by a monitor are receivable.*

Proof. The proof is similar to the previous one in the setting of controlled agents and channels 4.1. In the context of insecure channels, the well-formedness invariant still holds but we have to review the interface consistency invariant because interface consistency can be broken by an attacker. However, the origin-check ensures that channel discovery is modified so that interface consistency is preserved. At message receipt, if authentication fails, no discovery is performed for channel values in the message thus interface consistency remains valid. ■

Our type system supports the dynamic creation of new channels since it can be viewed as a channel discovery mechanism.

5 Related Work

We are not the first to advocate typing support in SOAs. The need for sophisticated service discovery, in particular supported by a sound type system with subtyping, has been advocated by Kourtesis and Paraskakis [6]. In a related context, testing for subschema relationships of XML Schemas is known as strong requirement for service interoperability [8].

A wide variety of formal models exists for service-oriented computing. There are principally two kinds of formalization: process calculus models for expressing and analyzing service based-systems [14] and formal models of standard orchestration languages (*e.g.,* BPEL [9]). These orchestration models are less abstract than our black box approach: they require to detail the service implementations while our approach is language and network independent.

There are only few references discussing type checking in the context of web services. Pu [11] defines a type system for semi-structured data with applications to a wide range of data models and query systems: relational data bases, XML documents, web services, etc. The type system is based on a nested record type system with collection and universal polymorphism. This type system is neither recursive nor does it allow channel mobility; its checking algorithm is expensive even in this restricted setting. Sans and Cervesato [12] deal with an abstract model that covers code mobility which we do not address, we are only concerned with remote procedure calls. On the other hand, they consider functions rather than channels and they do not support sum types, nor recursive types. They require a centralized typing table collecting types of services published everywhere in the Internet and assume that this repository can be trusted. A distributed and typed π-calculus for mobile agents is described in [5]. The type system considers malicious agents with erroneous types. Type safety is enforced by dynamically type checking agents when they enter a site. In contrast to our work they do not consider channel discovery or subtyping.

The principle of semantic subtyping is nicely presented in [4]. Semantic subtyping simplify the type systems: there is no need of syntactic normal form, and any set operation, for instance negation, can be integrated smoothly in the type language. Moreover, subtyping is directly interpreted as set inclusion. Our work directly exploit these results. We have still to study how their optimizations of the typing algorithm as done in the $\mathcal{C}Duce$ language can be translated into our system. Another way to optimize the algorithm has been proposed by Carpineti and Laneve [3].

6 Conclusion

SOAs are frequently used in highly dynamic environments, in particular because of the use of dynamic service discovery. Formal guarantees of properties over service compositions, notably in terms of type systems, are very useful in this context. However, few type systems for services have been put forward, frequently used infrastructures provide only minimal typing guarantees, and no existing type system support advanced features such dynamic service discovery in the presence of contravariant channels.

In this article, we have motivated the need for an advanced type system for web services. We have introduced a high level chemical semantics for service interactions and dynamic service discovery with first-class channels. The resulting system supports contravariant types for channels and type interference. We have formally shown fundamental correctness properties of the type system and shown how to apply it to make SOA safer and more secure. Our formal model allows the type-checking of messages and message authentication with both secure and insecure channels. Finally, we have formally proven different type-related properties of service interactions.

In the context of the CESSA research project, we plan to integrate such a type system into existing web service engines, use it to make the evolution of service systems and to generate execution monitors in order to secure web-applications.

A Proofs

Proposition 9 (Equivalence of type inference and checking)

$$\forall v, t \ . \ succeed(\llbracket\, t;\, v \,\rrbracket) \iff \exists \Gamma \ . \ \Gamma \vdash v \ S \ t$$

Terminology: in order to simplify the formulation of the proofs, we devolve terms introduced for type inference to type checking: in the following we say that *"type checking succeeds"* for value v and type t if $v \ S \ t$ can be derived using the type system of Fig. 6; conversely, we say that *"type checking fails"* if $v \ \bar{S} \ t$ can be derived.

Proof. The proof is done by induction on the type arguments (the lines in Fig. 8) and value arguments (the columns in the figure).

- Case $t = B$:
  - If $v = b$ then type inference and checking succeed iff $b \in B$, the latter because of rule $Base_1$.
  - If $v = k$ then inference fails; furthermore, $\neg(k \ S \ B)$ can be derived because of $Base_2$
  - If $v = l[v_1], v_2$ then inference fails and v does not type check because of $Base_2$.
- Case $t = \neg B$:

  Inference succeeds iff $v \notin B$; for type checking, $Not_1, Base_2$ imply $v \ S \ \neg B$ is also equivalent to $v \notin B$.
- Case $t =< t_1 >$:
  - If $v = b$, type inference and checking fail, the latter because of Ch_3.
  - Inferring $v = k$ against $< t_1 >$ always succeeds. If $k \ S \ < t_1 >$ then $t_1 \ \leq \ \Gamma(k)$ is a necessary and sufficient condition and thus Γ exists.
  - If $v = l[v_1], v_2$, type inference and checking fail, the latter because of Ch_3.
- Case $t = \neg < t_1 >$:
  - If $v = b$, inference and checking fail, the letter because Not_1, Ch_3 imply $v \ \bar{S} \ \neg < t_1 >$ is equivalent to $b \notin k$.
  - Inference of k against $\neg < t_1 >$ always succeeds. Type checking succeeds as well because $k \ S \ \neg < t_1 >$ is equivalent to $k \ \bar{S} \ < t_1 >$ which is equivalent to $\neg(t_1 \ \leq \ \Gamma(k))$ from the Ch_2 rule.
  - If $v = l[v_1], v_2$, inference and checking fail, the latter because Not_1, Ch_3 imply $l[v_1], v_2 \ S \ \neg < t_1 >$ is equivalent to $l[v_1], v_2 \notin k$.
- Case $t = l[t_1], t_2$:
  - If $v = b$ or $v = k$, inference and checking fail, the latter because of Lab_4 in both cases.

  - If $v = l[v_1], v_2$, inference succeeds iff $(l = l' \wedge succeed([\![t_1; v_1]\!]) \wedge succeed([\![t_2; v_2]\!]))$. As to type checking, Lab_1 yields $l[v_1], v_2 \ S \ l'[t_1], t_2$ which is equivalent to $(l = l' \wedge succeed([\![t_1; v_1]\!]) \wedge succeed([\![t_2; v_2]\!]))$.

- Case $\neg l[t'_1], t'_2$:

  - If $v = b$ or $v = k$, inference and checking succeed, the latter because of Not_1, Lab_4 in both cases.
  - If $v = l[v_1], v_2$, inference succeeds iff $(\neg l = l' + succeed([\![\neg t_1; v_1]\!]) + succeed([\![\neg t_2; v_2]\!]))$; as to type checking, $Not_1, Lab_2, Lab_3, Lab_4$ imply $l[v_1], v_2 \ S \ \neg l[t'_1], t'_2$, which is equivalent to $l \neq l' + succeed([\![\neg t_1; v_1]\!]) + succeed([\![\neg t_2; v_2]\!]))$.

- Case $\neg\neg t'$ is satisfied by induction on t' and rules Not_1 and Not_2.

- Cases $+$ and $\wedge$ and their negations follow from De Morgan's laws, induction hypothesis and rules for union and intersection of sets.

- The recursion cases (operator μ) are proven by induction provided that the function terminates which is the case from Prop. 1.

■

Proposition 10 (Typing and inferring)

$$\forall v, t \ . \ \forall \Gamma \ . \ (\Gamma \vdash v \ S \ t \Longleftrightarrow (succeed([\![t; v]\!]) \ \wedge \ context([\![t; v]\!]) \ \leq \ \Gamma))$$

Proof. The property is proven using the same induction principle as Prop. 2. For simplicity, we do not present here the (straightforward) cases in which type inference fails.

- Case $t = B$:

  If $v = b$ then typing succeeds for all the typing contexts and the inferred context is empty. Inference succeeds iff $b \in B$; furthermore v typechecks in this case for all type contexts because rule $Base_1$ implies $v \ S \ B$.

- Case $t = \neg B$:

  Type checking succeeds for all typing contexts and the inferred context is empty.

- Case $t =< t_1 >$:

  Type checking and inference succeed iff $v = k$ and the inferred context is reduced to $k :< t_1 >$ while the typing context states that $t_1 \ \leq \ \Gamma(k)$. This is equivalent to $< t_1 > \geq < \Gamma(k) >$, thus containment is satisfied.

- Case $t = \neg < t_1 >$:

  - For $v = b$ or $v = l[v_1], v_2$ typing succeeds in all typing contexts and the inferred context is empty.
  - For $v = k$ the inferred context contains one binding $k : \neg < t_1 >$. Type checking adds the constraint $\neg(t_1 \ \leq \ \Gamma(k))$. Equivalence then follows from the already proven channel property $\neg(t \leq r) \Longleftrightarrow < r > \leq \ \neg < t >$ (Eq. 2 of Fig. 7)

- Case $t = l[t_1], t_2$:

$\Longrightarrow$: If typing succeeds we have $v = l'[v_1], v_2$, $l = l'$ and typing succeeds on both parts v_1 and v_2. By induction we have $\Gamma \geq \mathit{context}([\![t_1; v_1]\!])$ and $\Gamma \geq \mathit{context}([\![t_2; v_2]\!])$ then $\Gamma \geq (\mathit{context}([\![t_1; v_1]\!] \wedge \mathit{context}([\![t_2; v_2]\!])$ thus $\Gamma \geq \mathit{context}([\![t; v]\!])$.

$\Longleftarrow$: The inference of $l[v_1], v_2$ succeeds iff $(l = l' \wedge \mathit{succeed}([\![t_1; v_1]\!]) \wedge \mathit{succeed}([\![t_2; v_2]\!]))$ and $\mathit{context}([\![l'[t_1], t_2; l'[v_1], v_2]\!]) = \mathit{context}([\![t_1; v_1]\!]) \wedge \mathit{context}([\![t_2; v_2]\!])$. Let Γ a typing context $\mathit{context}([\![l'[t_1], t_2; l'[v_1], v_2]\!]) \leq \Gamma$, we can show that $\mathit{context}([\![t_1; v_1]\!]) \leq \Gamma$.

First, domains are included: $\mathit{dom}(\mathit{context}([\![t_1; v_1]\!])) \subseteq \mathit{dom}(\Gamma)$.

If $k \in (\mathit{dom}(\mathit{context}([\![t_1; v_1]\!])) \wedge \neg \mathit{dom}(\mathit{context}([\![t_2; v_2]\!]))$ then its type obeys $\mathit{context}([\![t_1; v_1]\!])(k) = \mathit{context}([\![l'[t_1], t_2; l'[v_1], v_2]\!])(k) \geq < \Gamma(k) >$.

If $k \in (\mathit{dom}(\mathit{context}([\![t_1; v_1]\!])) \wedge \mathit{dom}(\mathit{context}([\![t_2; v_2]\!]))$ then its type obeys $\mathit{context}([\![t_1; v_1]\!])(k) \geq \mathit{context}([\![l'[t_1], t_2; l'[v_1], v_2]\!])(k) \geq < \Gamma(k) >$. This property holds also for $\mathit{context}([\![t_2; v_2]\!])$.

From the induction hypothesis we have $\Gamma \vdash v_1 \; S \; t_1$ and $\Gamma \vdash v_2 \; S \; t_2$ and applying the Lab_1 rule we get $\Gamma \vdash l'[t_1], t_2 \; S \; l'[v_1], v_2$.

- Case $t = \neg l[t_1], t_2$:
  - Case $v = b$ succeeds without constraints on contexts.
  - Case $v = k$ succeeds with an inferred context $k = \neg l[t_1], t_2$. Equivalence follows because any Γ such that $\Gamma(k) = r$ entails $< \Gamma(k) > \leq \neg l[t_1], t_2$.
  - Case $v = l'[v_1], v_2$:

    $\Longrightarrow$: If typing succeeds then either Lab_4, Lab_2 or Lab_3 implies that $l \neq l'$, $v_1 \; \bar{S} \; t_1$ or $v_2 \; \bar{S} \; t_2$. In the first case no condition on the typing context is generated and the implication is satisfied.
    Otherwise, if $l = l'$, we have to consider three cases for v_1 and v_2 depending on whether k is defined exclusively by inference based on either v_1 or v_2, or defined by inferences on both values. Consider the first case (the second is symmetric). By induction we have $< \Gamma(k) > \leq \mathit{context}([\![\neg t_1; v_1]\!])(k)$; since $\mathit{context}([\![t; v]\!]) = \mathit{context}([\![\neg t_1; v_1]\!]) + \mathit{context}([\![\neg t_2; v_2]\!])$, $< \Gamma(k) > \leq \mathit{context}([\![t; v]\!])(k)$ follows. If k is defined by inference of both values the merged context are calculated as the union of both contexts; $< \Gamma(k) > \leq \mathit{context}([\![t; v]\!])(k)$ then follows from the fact that subtyping is compatible with the union of types.

    $\Longleftarrow$: The inference of $l'[v_1], v_2$ succeeds iff $(l \neq l' + \mathit{succeed}([\![\neg t_1; v_1]\!]) + \mathit{succeed}([\![\neg t_2; v_2]\!]))$. If $l \neq l'$ then $l'[v_1], v_2 \; S \; \neg l'[t_1], t_2$ for all typing contexts. If $l = l'$ the inferred context is $\mathit{context}([\![\neg t_1; v_1]\!]) + \mathit{context}([\![\neg t_2; v_2]\!]))$. Let Γ a typing context $\Gamma \geq \mathit{context}([\![l'[t_1], t_2; l'[v_1], v_2]\!])$: this context contains the union as well as the intersection of the inferred contexts. As in the case with $l[t_1], t_2$ we have Γ which contains both $\mathit{context}([\![\neg t_1; v_1]\!])$ and $\mathit{context}([\![\neg t_2; v_2]\!])$. From induction hypothesis we have $\Gamma \vdash v_1 \; S \; \neg t_1$ and $\Gamma \vdash v_2 \; S \; \neg t_2$ and applying the Lab_2 or Lab_3 rules we get $\Gamma \vdash \neg l'[t_1], t_2 \; S \; l'[v_1], v_2$.

- Case $\neg\neg t$ is satisfied by induction on t.
- Case $t_1 + t_2$.

  $\Longrightarrow$: If typing succeeds with Γ it succeeds either on t_1 or on t_2. Consider the three cases depending of the definition of k and the reasoning is similar to the one use for the case $l[t_1], t_2$.

$\Longleftarrow$: Inference succeeds with $succeed(\llbracket t_1; v \rrbracket) + succeed(\llbracket t_2; v \rrbracket)$. The inferred context is $context(\llbracket t_1; v \rrbracket) + context(\llbracket t_2; v \rrbracket)$. For all context Γ containing $context(\llbracket t_1; v \rrbracket) + context(\llbracket t_2; v \rrbracket)$ we can show that it contains both contexts. Thus applying the induction hypothesis and rule $Union_1$ or $Union_2$ we get that $v\ S\ (t_1 + t_2)$.

- Case $\wedge$ is similar to the case of $l[v_1], v_2$ without the label condition.
- Negation cases for union and intersection use induction and De Morgan's laws.
- The recursion case is satisfied using induction.

■

References

[1] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1), 1992.

[2] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, 1998.

[3] S. Carpineti and C. Laneve. A basic contract language for web services. In *ESOP, volume 3924 of LNCS*, pages 197–213. Springer, 2006.

[4] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In *Proc. of ICALP*, volume 3580 of *LNCS*, pages 30–34. Springer, 2005.

[5] M. Hennessy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *SIP*, volume 1603 of *LNCS*, pages 95–115, 1999.

[6] D. Kourtesis and I. Paraskakis. *Semantic Enterprise Application Integration for Business Processes: Service-Oriented Frameworks*, chapter IV. Business Science Reference, Hersley, 2009.

[7] L. Lamport and N. Lynch. Distributed computing: Models and methods. pages 1157–1199. 1990.

[8] T. Y. Lee Lee and D. W. Cheung. Formal models and algorithms for XML data interoperability. *JCSE*, 4(4):313–349, 2010.

[9] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. In *Journal of Logic and Algebraic Programming*. Elsevier press, 2005.

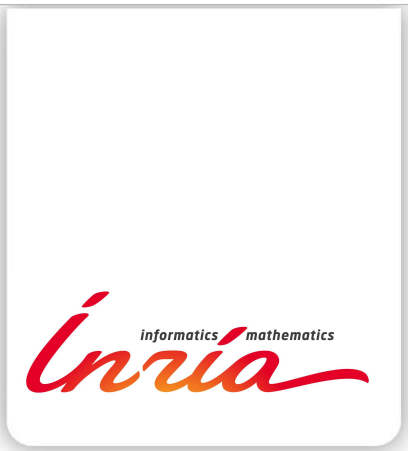
[10] A. J. Menezes, S. A. Vanstone, and P. C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[11] K. Q. Pu. Service description and analysis from a type theoretic approach. In *ICDE Workshops*, pages 379–386, 2007.

[12] T. Sans and I. Cervesato. Qwesst for type-safe web programming. LAM'10, `http://www.qatar.cmu.edu/ tsans/index.php?page= research`, 2010.

[13] João Costa Seco and Luís Caires. A basic model of typed components. In *Proc. of ECOOP 2000*, LNCS 1850, pages 108–128. Springer, 2000.

[14] H. T. Vieira, L. Caires, and J.o C. Seco. The conversation calculus: a model of service oriented computation. In *In Proc. of ESOP'08, LNCS*. Springer, 2008.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399