Regulating data exchange in service oriented applications*

Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi

Dipartimento di Sistemi e Informatica Università degli Studi di Firenze
{lapadula,pugliese,tiezzi}@dsi.unifi.it

Abstract. We define a type system for COWS, a formalism for specifying and combining services, while modelling their dynamic behaviour. Our types permit to express policies constraining data exchanges in terms of sets of service partner names attachable to each single datum. Service programmers explicitly write only the annotations necessary to specify the wanted policies for communicable data, while a type inference system (statically) derives the minimal additional annotations that ensure consistency of services initial configuration. Then, the language dynamic semantics only performs very simple checks to authorize or block communication. We prove that the type system and the operational semantics are sound. As a consequence, we have the following data protection property: services always comply with the policies regulating the exchange of data among interacting services. We illustrate our approach through a simplified but realistic scenario for a service-based electronic marketplace.

1 Introduction

Service-oriented computing (SOC) is an emerging paradigm for developing loosely coupled, interoperable, evolvable applications which exploits the pervasiveness of the Internet and its related technologies. SOC systems deliver application functionality as services to either end-user applications or other services. Current software engineering technologies for SOC, however, remain at the descriptive level and do not support analytical tools for checking that SOC applications enjoy desirable properties and do not manifest unexpected behaviors. To reason about and guarantee such properties, one must also be able to specify and enforce some security policies. Indeed, programming service oriented middlewares and the applications running on them without putting data at risk or compromising robustness of the whole platform requires services to be checked and their resource usage to be strictly put in relation to their capabilities.

Great efforts have been recently devoted to embed security mechanisms within standard programming features (some of these techniques are surveyed in [1]). Language-based mechanisms are a scalable way to provide evidence that a large number of applications enjoy some given properties. For example, by using type systems, one can prove the type soundness of the language as a whole, from which it follows that all well-typed applications do comply with the policies stated by their types. To facilitate the task of designing such a sound language for SOC, one can initially focus only on

* This work has been supported by the EU project SENSORIA, IST-2005-016004.

the mechanisms at the basis of SOC. Afterwards, this core formalism could hopefully be expanded into a full-fledged language by adding the high level, often redundant, constructs typical of effective programming languages.

Many researchers have hence put forward exploiting the studies on *process calculi*, a cornerstone of current foundational research on specification and analysis of concurrent, distributed and mobile systems through mathematical — mainly algebraic and logical — tools. Indeed, due to their algebraic nature, process calculi convey in a distilled form the compositional programming style of SOC. This is witnessed by the several process calculi like formalisms for SOC proposed in the literature by now (see, e.g., [2–9]). However, although capable of describing complex systems and applications, such proposals still lack those reasoning mechanisms and analytical tools, e.g. type systems and behavioural equivalences, that process calculi usually hand down.

In this paper, we tailor the type-based approach for protecting data in global computing applications put forward in [10] to COWS, a formalism for specifying service-based applications that we introduce in [9]. We thus define a typed variant of COWS that permits expressing and forcing policies regulating the exchange of data among interacting services. Programmers can indeed settle the partners usable to exchange any given datum (and, then, the services that can share it), thus avoiding the datum be accessed (by unwanted services) through unauthorized partners. The language (static and dynamic) semantics then guarantees that well-typed services always comply with the constraints expressed by the type associated to each single datum.

The rest of the paper is organized as follows. Section 2 introduces syntax, type inference and operational semantics of (our typed variant of) COWS, while Section 3 presents our main results. Section 4 demonstrates our approach through a simplified but realistic scenario for a service-based electronic marketplace. Finally, Section 5 touches upon comparisons with more strictly related work and directions for future work.

2 COWS: Calculus for Orchestration of Web Services

Before formally defining our language, we provide some insights on its main features. We refer the interested reader to [9] for further motivations on the design of COWS, for many examples illustrating its peculiarities and expressiveness, and for comparisons with other process-based and orchestration formalisms.

The design of COWS has been influenced by the principles underlying WS-BPEL [11], the *de facto* standard language for orchestration of web services. Similarly to WS-BPEL, COWS supports service instances with shared states, allows a same process to play more than one partner role and permits programming stateful sessions by correlating different service interactions. However, COWS intends to be a foundational model not specifically tight to web services' current technology. Thus, some WS-BPEL constructs, such as e.g. fault and compensation handlers and flow graphs, do not have a precise counterpart in COWS, rather they are expressed in terms of more primitive operators (see [12], Sect. 3). The design of COWS has also taken advantage of previous work on process calculi. In fact, it combines in an original way constructs and features borrowed from well-known process calculi, e.g. asynchronous communication,

polyadic synchronization, pattern matching, protection, delimited receiving and killing activities, while however resulting different from any of them.

The basic elements of COWS are *partners* and *operations*. They can be combined to designate *communication endpoints* and can be exchanged in communication, but dynamically received names cannot form endpoints used to receive further invocations. Endpoints naming mechanism is very flexible, e.g. it permits identifying a same service by means of different logic names and separately dealing with the names composing an endpoint. This is, e.g., exploited in request-response interaction, where usually the service provider knows the name of the response operation, but not the partner name of the service it has to reply to.

COWS computational entities are called *services*. Typically, a service creates one specific instance to serve each received request. Instances may run concurrently. Each instance can be composed of concurrent threads that may offer a choice among alternative receive activities. Services could be able to receive multiple messages in a statically unpredictable order and in such a way that the first incoming message triggers creation of a service instance which subsequent messages are routed to. *Pattern matching* is the mechanism used for correlating messages logically forming a same interaction 'session' by means of their same contents. It permits locating those data that are important to identify service instances and is flexible enough for allowing a single message to participate in multiple interaction sessions, each identified by separate correlation values.

Inter-service communication give rise to substitutions of variables with values. However, to enable concurrent instances or threads within an instance to share the state (or part of it), receive activities in COWS do *not* bind variables. The range of application of the substitution generated by a communication is then regulated by the *delimitation* operator, that is the only binder of the calculus. Delimitation, additionally, can be used to generate fresh private names (as the restriction operator of the π-calculus) and to delimit the field of action of the *kill* activity, a powerful orchestration construct that can be used to force termination of whole service instances. Sensitive code can however be protected from the effect of a forced termination by using the *protection* operator.

The type system we present in this paper permits to express and enforce policies for regulating the exchange of data among services. To implement such policies, programmers can annotate data with sets of partner names characterizing the services authorized to use and exchange them; these sets are called *regions*. The language operational semantics uses these annotations to guarantee that computations proceed according to them. This property, called *soundness*, can be stated as follows

> A service s is *sound* if, for any datum v in s associated to region r and for all evolutions of s, it holds that v can be exchanged only by using partners in r.

To facilitate the task of decorating COWS terms with type annotations, we let the type system partially infer such annotations *à la* ML: service programmers explicitly write only the annotations necessary to specify the wanted policies for communicable data; then, a type inference system (statically) performs some coherence checks (e.g. the partner used by an invoke must belong to the regions of all data occurring in the argument of the activity) and derives the minimal region annotations for variable declarations that ensure consistency of services initial configuration. This allows us to define an *operational semantics with types* [13] which is simpler than a full-fledged *typed operational semantics*, because it only performs simple checks (i.e. subset inclusion) using region

Table 1. COWS syntax

$s ::=$	$\mathbf{kill}(k) \;\mid\; u \bullet u' ! \overline{\{e(\bar{x})\}_r} \;\mid\; g \;\mid\; s \mid s \;\mid\; \{\!|s|\!\} \;\mid\; [d]\, s \;\mid\; *\, s$	(services)
$g ::=$	$\mathbf{0} \;\mid\; p \bullet o ? \bar{w}.s \;\mid\; g + g$	(receive-guarded choice)

annotations to authorize or block transitions. Our main results prove that the type system and the operational semantics are sound. As a consequence, we have that services always comply with the constraints expressed by the type of each single datum.

Syntax. COWS syntax is parameterized by three countable and pairwise disjoint sets: the set of *(killer) labels* (ranged over by $k, k', \ldots$), the set of *values* (ranged over by v, v', ...) and the set of 'write once' *variables* (ranged over by x, y, ...). The set of values is left unspecified; however, we assume that it includes the set of *names*, ranged over by n, m, ..., mainly used to represent partners and operations. COWS is also parameterized by a set of *expressions*, ranged over by e, whose exact syntax is deliberately omitted; we just assume that expressions contain, at least, values and variables. Notably, killer labels are *not* (communicable) values. Notationally, we prefer letters $p, p', \ldots$ when we want to stress the use of a name as a partner, $o, o', \ldots$ when we want to stress the use of a name as an operation. We will use w to range over values and variables, u to range over names and variables, and d to range over killer labels, names and variables.

Regions can be either finite subsets of partners and variables or the distinct element $\top$ (denoting the universe of partners). The set of all regions, ranged over by r, is partially ordered by the subset inclusion relation $\subseteq$, and has $\top$ as top element.

Notation $\bar{\cdot}$ stands for tuples of objects, e.g. $\bar{x}$ is a compact notation for denoting the tuple of variables $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$). We assume that variables in the same tuple are pairwise distinct. All notations shall extend to tuples component-wise. An expression e tagged with region r will be written as $\{e\}_r$; an untagged e will stand for $\{e\}_\top$. We will write $e(\bar{x})$ to make explicit all the variables $\bar{x}$ occurring in e (we still write e when this information is not needed), and $\bar{e}$ (resp. $\bar{r}$) to denote the tuple of the expressions (resp. regions) occurring in $\overline{\{e\}_r}$.

We will call *raw* services those COWS services written according to the syntax in Table 1. Intuitively, raw services only contain those region annotations that implement the policies for data exchange settled by the programmers. *Services* are structured activities built from basic activities, i.e. the empty activity $\mathbf{0}$, the kill activity $\mathbf{kill}(\_)$, the invoke activity $\_ \bullet \_ !\_$ and the receive activity $\_ \bullet \_ ?\_$, by means of (receive) prefixing $\_.\_$, guarded choice $\_ + \_$, parallel composition $\_ \mid \_$, protection $\{\!|\_|\!\}$, delimitation $[\_]\,\_$ and replication $*\,\_$. Notably, as in the Lπ [14], communication endpoints of receive activities are identified statically because their syntax only allows using names and not variables. We adopt the following conventions about the operators precedence: monadic operators bind more tightly than parallel composition, and prefixing more tightly than choice. We shall omit a trailing $\mathbf{0}$ and use $[d_1, \ldots, d_n]\, s$ to denote $[d_1] \ldots [d_n]\, s$.

The only *binding* construct is delimitation: $[d]\, s$ binds d in the scope s. The occurrence of a name/variable/label is *free* if it is not under the scope of a binder. We denote by $\mathrm{fd}(t)$ (resp. $\mathrm{bd}(t)$) the set of names, variables and killer labels that occur free (resp. bound) in a term t, by $\mathrm{fv}(t)$ (resp. $\mathrm{bv}(t)$) the set of free (resp. bound) variables in t,

Table 2. Type inference system

$$\Gamma \vdash \mathbf{0} \succ \Gamma \vdash \mathbf{0} \quad \textit{(t-nil)} \qquad \Gamma \vdash \mathbf{kill}(k) \succ \Gamma \vdash \mathbf{kill}(k) \quad \textit{(t-kill)}$$

$$\frac{\forall\, r' \in \{r_i\}_{i\in\{1,\ldots,n\}} \quad u_1 \in r'}{\begin{array}{c}\Gamma \vdash u_1 \bullet u_2!\langle\{e_1(\bar{y}_1)\}_{r_1},\ldots,\{e_n(\bar{y}_n)\}_{r_n}\rangle \succ \\ (\Gamma + \{x : r_1\}_{x\in\bar{y}_1} + \ldots + \{x : r_n\}_{x\in\bar{y}_n}) \vdash u_1 \bullet u_2!\langle\{e_1(\bar{y}_1)\}_{r_1},\ldots,\{e_n(\bar{y}_n)\}_{r_n}\rangle\end{array}} \textit{(t-inv)}$$

$$\frac{\Gamma + \{x : \{p\}\}_{x\in \mathrm{fv}(\bar{w})} \vdash s \succ \Gamma' \vdash s'}{\Gamma \vdash p \bullet o?\bar{w}.s \succ \Gamma' \vdash p \bullet o?\bar{w}.s'} \textit{(t-rec)}$$

$$\frac{\Gamma \vdash g_1 \succ \Gamma_1 \vdash g_1' \qquad \Gamma \vdash g_2 \succ \Gamma_2 \vdash g_2'}{\Gamma \vdash g_1 + g_2 \succ \Gamma_1 + \Gamma_2 \vdash g_1' + g_2'} \textit{(t-sum)}$$

$$\frac{\Gamma \vdash s \succ \Gamma' \vdash s'}{\Gamma \vdash \{\!|s|\!\} \succ \Gamma' \vdash \{\!|s'|\!\}} \textit{(t-prot)} \qquad \frac{\Gamma \vdash s \succ \Gamma' \vdash s'}{\Gamma \vdash *s \succ \Gamma' \vdash *s'} \textit{(t-repl)}$$

$$\frac{\Gamma \vdash s \succ \Gamma' \vdash s' \qquad n \notin \mathrm{reg}(\Gamma')}{\Gamma \vdash [n]\, s \succ \Gamma' \vdash [n]\, s'} \textit{(t-del}_{name}\textit{)} \qquad \frac{\Gamma \vdash s \succ \Gamma' \vdash s'}{\Gamma \vdash [k]\, s \succ \Gamma' \vdash [k]\, s'} \textit{(t-del}_{lab}\textit{)}$$

$$\frac{\Gamma, \{x : \emptyset\} \vdash s \succ \Gamma', \{x : r\} \vdash s' \qquad x \notin \mathrm{reg}(\Gamma')}{\Gamma \vdash [x]\, s \succ \Gamma' \vdash [\{x\}^{r-\{x\}}]\, s'} \textit{(t-del}_{var}\textit{)}$$

$$\frac{\Gamma \vdash s_1 \succ \Gamma_1 \vdash s_1' \qquad \Gamma \vdash s_2 \succ \Gamma_2 \vdash s_2'}{\Gamma \vdash s_1 \mid s_2 \succ \Gamma_1 + \Gamma_2 \vdash s_1' \mid s_2'} \textit{(t-par)}$$

and by $\mathrm{fk}(t)$ the set of free killer labels in t. Two terms are *alpha-equivalent* if one can be obtained from the other by consistently renaming bound names/variables/labels. As usual, we identify terms up to alpha-equivalence. For simplicity sake, in the sequel we assume that bound variables in services are pairwise distinct (of course, this condition is not restrictive and can always be fulfilled by possibly using alpha-conversion).

A type inference system. The annotations put by the type inference are written as superscripts, to better distinguish them from those put by the programmers. Thus, the syntax of variable delimitation becomes $[\{x\}^r]\, s$, which means that the datum that dynamically will replace x will be used at most by the partners in r. *Typed* COWS services are then generated by the syntax in Table 1 where, differently from the previous section, d ranges over killer labels, names and annotated variables as $\{x\}^r$. Notably, types may *depend* on partner variables, i.e. on parameters of receiving activities; during computation, they are therefore affected by application of substitutions that replace partner variables with partner names. We assume that the region of a partner name always contains, at least implicitly, such partner.

The type inference system is presented in Table 2. Typing judgements are written $\Gamma \vdash s \succ \Gamma' \vdash s'$, where the type environment Γ is a finite function from variables to regions such that $\mathrm{fv}(s) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{bv}(s) \cap \mathrm{dom}(\Gamma) = \emptyset$ (the same holds for Γ'

and s'). Type environments are written as sets of pairs of the form $x : r$, where x is a partner variable and r is its assumed region annotation. The domain of an environment is defined as usual: $\mathrm{dom}(\emptyset) = \emptyset$ and $\mathrm{dom}(\Gamma, \{x : r\}) = \mathrm{dom}(\Gamma) \cup \{x\}$, where ',' denotes union between environments with disjoint domains. The *region* of Γ is the union of the regions in Γ, i.e. $\mathrm{reg}(\emptyset) = \emptyset$ and $\mathrm{reg}(\Gamma, \{x : r\}) = r \cup \mathrm{reg}(\Gamma)$. We will write $\Gamma + \Gamma'$ to denote the environment obtained by extending Γ with Γ'; + is inductively defined by

$$\Gamma + \emptyset = \Gamma$$

$$\Gamma + \{x : r\} = \begin{cases} \Gamma', \{x : r \cup r'\} & \text{if } \Gamma = \Gamma', \{x : r'\} \\ \Gamma, \{x : r\} & \text{otherwise} \end{cases}$$

$$\Gamma + (\{x : r\}, \Gamma') = (\Gamma + \{x : r\}) + \Gamma'$$

Hence, the judgement $\emptyset \vdash s \succ \emptyset \vdash s'$ can be derived only if s is a closed raw service (because the initial environment is empty); if it is derivable, then s' is the typed service obtained by decorating s with the region annotations describing the use of each variable of s in its scope. Type inference determines such regions by considering the invoking and receiving partners where the variables occur.

We now comment on the most significant typing rules. Rule *(t-inv)* checks if the invoked partner u_1 belongs to the regions of the communicated data. If it succeeds, the type environment Γ is extended by associating a proper region to each variable used in the expressions argument of the invoke activity. Rule *(t-rec)* tries to type s in the type environment Γ extended by adding the receiving partner to the regions of the variables in $\bar{w}$. Rules *(t-sum)* and *(t-par)* yield the same typing; this is due to the sharing of variables. For instance, service $[x]\,(p \bullet o?\langle x\rangle \mid p' \bullet o'!\langle\{x\}_r\rangle)$ with $p' \in r$ is annotated as $[\{x\}^{r'}]\,(p \bullet o?\langle x\rangle \mid p' \bullet o'!\langle\{x\}_r\rangle)$ with $r' = (\{p\} \cup r - \{x\})$. In rule *(t-del$_{name}$)*, premise $n \notin \mathrm{reg}(\Gamma')$ prevents a new name n to escape from its binder $[n]$ in the inference. As an example, consider the closed raw service

$$[z]\, p \bullet o?\langle z\rangle \,.\, [p']\, p'' \bullet o''!\langle\{z\}_{\{p'',p'\}}\rangle \qquad (*)$$

Without the premise $n \notin \mathrm{reg}(\Gamma')$, the service resulting from the type inference would be $[\{z\}^{\{p'',p',p\}}]\, p \bullet o?\langle z\rangle \,.\, [p']\, p'' \bullet o''!\langle\{z\}_{\{p'',p'\}}\rangle$. The problem with this service is that the name p' occurring in the annotation associated to z by the inference system escapes from the scope of its binder and, thus, represents a completely different name. Although, service $(*)$ is not typable, by a simple semantics preserving manipulation one can get a typable service as, e.g., the following one $[p']\,[z]\, p \bullet o?\langle z\rangle \,.\, p'' \bullet o''!\langle\{z\}_{\{p'',p'\}}\rangle$.

Similarly, in rule *(t-del$_{var}$)*, premise $x \notin \mathrm{reg}(\Gamma')$ prevents initially closed services to become open at the end of the inference. Otherwise, e.g., the type inference would transform the closed raw service

$$[x]\, p \bullet o?\langle x\rangle \,.\, [y]\, p' \bullet o'?\langle y\rangle \,.\, p'' \bullet o''!\langle\{x\}_{\{p'',y\}}\rangle \qquad (**)$$

into the open service $[\{x\}^{\{p,p'',y\}}]\, p \bullet o?\langle x\rangle \,.\, [\{y\}^{\{p'\}}]\, p' \bullet o'?\langle y\rangle \,.\, p'' \bullet o''!\langle\{x\}_{\{p'',y\}}\rangle$. Also in this case, we can easily modify the untypable service $(**)$ to get a typable one with a similar semantics like, e.g., the service $[y]\,[x]\, p \bullet o?\langle x\rangle \,.\, p' \bullet o'?\langle y\rangle \,.\, p'' \bullet o''!\langle\{x\}_{\{p'',y\}}\rangle$.

Furthermore, in *(t-del$_{var}$)*, x is annotated with $r - \{x\}$, rather than with r, otherwise initially closed services could become open. E.g., the closed raw service $[x]\, p \bullet o?\langle x\rangle \,.$

Table 3. Structural congruence

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
* \mathbf{0} & \equiv & \mathbf{0} & * s & \equiv & s \mid * s & \{\!|\mathbf{0}|\!\} & \equiv & \mathbf{0} \\
\{\!|\{\!|s|\!\}|\!\} & \equiv & \{\!|s|\!\} & \{\!|[d]\, s|\!\} & \equiv & [d]\, \{\!|s|\!\} & [d]\, \mathbf{0} & \equiv & \mathbf{0} \\
[d_1]\,[d_2]\, s & \equiv & [d_2]\,[d_1]\, s & & & \text{if } d_1 \neq \{x\}^{r_1} \text{ and } d_2 \neq \{y\}^{r_2} & & & \\
[n]\,[\{x\}^r]\, s & \equiv & [\{x\}^r]\,[n]\, s & & & \text{if } n \notin r & & & \\
[\{x\}^{r_1}]\,[\{y\}^{r_2}]\, s & \equiv & [\{y\}^{r_2}]\,[\{x\}^{r_1}]\, s & & & \text{if } y \notin r_1 \text{ and } x \notin r_2 & & & \\
s_1 \mid [d]\, s_2 & \equiv & [d]\,(s_1 \mid s_2) & & & \text{if } d \notin \mathrm{fd}(s_1) \cup \mathrm{fk}(s_2) & & &
\end{array}
$$

Table 4. Matching rules

$$
\mathcal{M}(v, \{v\}_r) = \emptyset \qquad \mathcal{M}(x, \{v\}_r) = \{x \mapsto \{v\}_r\} \qquad \frac{\mathcal{M}(w_1, \{v_1\}_{r_1}) = \sigma_1 \qquad \mathcal{M}(\bar{w}_2, \overline{\{v_2\}_{r_2}}) = \sigma_2}{\mathcal{M}((w_1, \bar{w}_2), (\{v_1\}_{r_1}, \overline{\{v_2\}_{r_2}})) = \sigma_1 \uplus \sigma_2}
$$

$p' \cdot o'!\langle\{x\}_{\{p',x\}}\rangle$ would be transformed into the open service $[\{x\}^{\{p,p',x\}}]\, p \cdot o?\langle x\rangle \,.\, p' \cdot o'!\langle\{x\}_{\{p',x\}}\rangle$ (indeed, x occurs in the annotation associated to its declaration). Notice that, although the region associated to x by the inference does never record that a service possibly transmits x with regions containing x, rule *(t-del$_{var}$)* is sound because we assumed that the region of a partner name, at least implicitly, contains the partner name.

Definition 1. *A service s is* well-typed *if $\emptyset \vdash s' \succ \emptyset \vdash s$ for some raw service s'.*

Operational semantics. COWS operational semantics is defined only for *closed* services, i.e. services without free variables/labels (similarly to many real compilers, we consider terms with free variables/labels as programming errors), but of course the rules also involve non-closed services (see e.g. the premises of rules *(del_)*). Formally, the semantics is given in terms of a structural congruence and of a labelled transition relation.

The structural congruence $\equiv$ identifies syntactically different services that intuitively represent the same service. It is defined as the least congruence relation induced by a given set of equational laws. We explicitly show in Table 3 the laws for replication, protection and delimitation, while omit the (standard) laws for the other operators stating that parallel composition is commutative, associative and has $\mathbf{0}$ as identity element, and that guarded choice enjoys the same properties and, additionally, is idempotent. All the presented laws are straightforward. Only notice that the last law can be used to extend the scope of names (like a similar law in the π-calculus), thus enabling communication of restricted names, except when the argument d of the delimitation is a free killer label of s_2 (this avoids involving s_1 in the effect of a kill activity inside s_2).

To define the labelled transition relation, we need a few auxiliary functions. First, we exploit a function $[\![\_]\!]$ for evaluating *closed* expressions (i.e. expressions without variables): it takes a closed expression and returns a value. However, $[\![\_]\!]$ cannot be explicitly defined because the exact syntax of expressions is deliberately not specified.

Then, through the rules in Table 4, we define the partial function $\mathcal{M}(\_,\_)$ that permits performing *pattern-matching* on semi-structured data thus determining if a receive

Table 5. Is there an active **kill**(k)? / Are there conflicting receives along $p \bullet o$ matching $\bar{v}$?

$$\mathbf{kill}(k) \downarrow_{kill} \qquad \frac{s \downarrow_{kill} \ \vee \ s' \downarrow_{kill}}{s \mid s' \downarrow_{kill}} \qquad \frac{s \downarrow_{kill}}{\{\!|s|\!\} \downarrow_{kill}} \qquad \frac{s \downarrow_{kill}}{[d]\, s \downarrow_{kill}} \qquad \frac{s \downarrow_{kill}}{* s \downarrow_{kill}}$$

$$\frac{|\mathcal{M}(\bar{w}, \bar{v})| < \ell}{p \bullet o?\bar{w}.s \downarrow^{\ell}_{p \bullet o, \bar{v}}} \qquad \frac{s \downarrow^{\ell}_{p \bullet o, \bar{v}} \quad d \notin \{p, o\}}{[d]\, s \downarrow^{\ell}_{p \bullet o, \bar{v}}} \qquad \frac{s \downarrow^{\ell}_{p \bullet o, \bar{v}}}{\{\!|s|\!\} \downarrow^{\ell}_{p \bullet o, \bar{v}}}$$

$$\frac{g \downarrow^{\ell}_{p \bullet o, \bar{v}} \ \vee \ g' \downarrow^{\ell}_{p \bullet o, \bar{v}}}{g + g' \downarrow^{\ell}_{p \bullet o, \bar{v}}} \qquad \frac{s \downarrow^{\ell}_{p \bullet o, \bar{v}} \ \vee \ s' \downarrow^{\ell}_{p \bullet o, \bar{v}}}{s \mid s' \downarrow^{\ell}_{p \bullet o, \bar{v}}} \qquad \frac{s \downarrow^{\ell}_{p \bullet o, \bar{v}}}{* s \downarrow^{\ell}_{p \bullet o, \bar{v}}}$$

and an invoke over the same endpoint can synchronize. The rules state that two tuples match if they have the same number of fields and corresponding fields have matching values/variables. Variables match any annotated value, and a value matches an annotated value only if, apart for the region annotation, they are identical. When tuples $\bar{w}$ and $\overline{\{v\}_r}$ do match, $\mathcal{M}(\bar{w}, \overline{\{v\}_r})$ returns a substitution, that also records region annotations of values exchanged in communication, for the variables in $\bar{w}$; otherwise, it is undefined. *Substitutions* (ranged over by σ) are functions mapping variables to annotated values and are written as collections of pairs of the form $x \mapsto \{v\}_r$. Application of substitution σ to s, written $s \cdot \sigma$, has the effect of replacing every free occurrence of x in s with v, for each $x \mapsto \{v\}_r \in \sigma$, by possibly using alpha-conversion for avoiding v to be captured by name delimitations within s. We use $|\sigma|$ to denote the number of pairs in σ and $\sigma_1 \uplus \sigma_2$ to denote the union of σ_1 and σ_2 when they have disjoint domains.

We also define a function, named $halt(\_)$, that takes a service s as an argument and returns the service obtained by only retaining the protected activities inside s. $halt(\_)$ is defined inductively on the syntax of services. The most significant case is $halt(\{\!|s|\!\}) = \{\!|s|\!\}$. In the other cases, $halt(\_)$ returns $\mathbf{0}$, except for parallel composition, delimitation and replication operators, for which it acts as an homomorphism.

Finally, in Table 5, we inductively define two predicates: $s \downarrow_{kill}$ checks if s can immediately perform a kill activity; $s \downarrow^{\ell}_{p \bullet o, \bar{v}}$, with ℓ natural number, checks existence of potential communication conflicts, i.e. the ability of s of performing a receive activity matching $\bar{v}$ over the endpoint $p \bullet o$ that generates a substitution with fewer pairs than ℓ.

The labelled transition relation $\xrightarrow{\alpha}$ is the least relation over services induced by the rules in Table 6, where α is generated by the following grammar:

$$\alpha ::= \dagger k \quad | \quad (p \bullet o) \lhd \overline{\{v\}_r} \quad | \quad (p \bullet o) \rhd \bar{w} \quad | \quad p \bullet o \lfloor \sigma \rfloor \bar{w}\, \overline{\{v\}_r} \quad | \quad \dagger$$

In the sequel, we use $d(\alpha)$ to denote the set of names, variables and killer labels occurring in α, except for $\alpha = p \bullet o \lfloor \sigma \rfloor \bar{w}\, \overline{\{v\}_r}$ for which we let $d(p \bullet o \lfloor \sigma \rfloor \bar{w}\, \overline{\{v\}_r}) = d(\sigma)$, where $d(\{x \mapsto \{v\}_r\}) = \{x, v\} \cup r$ and $d(\sigma_1 \uplus \sigma_2) = d(\sigma_1) \cup d(\sigma_2)$. The meaning of labels is as follows: $\dagger k$ denotes execution of a request for terminating a term from within the delimitation $[k]$, $(p \bullet o) \lhd \overline{\{v\}_r}$ and $(p \bullet o) \rhd \bar{w}$ denote execution of invoke and receive activities over the endpoint $p \bullet o$, respectively, $p \bullet o \lfloor \sigma \rfloor \bar{w}\, \overline{\{v\}_r}$ (if $\sigma \neq \emptyset$) denotes execu-

Table 6. Operational semantics

$$\mathbf{kill}(k) \xrightarrow{\dagger k} \mathbf{0} \quad (kill) \qquad\qquad p \bullet o?\bar{w}.s \xrightarrow{(p \bullet o) \rhd \bar{w}} s \quad (rec)$$

$$\frac{[\![\bar{e}]\!] = \bar{v} \qquad \mathrm{fv}(\bar{r}) = \emptyset}{p \bullet o!\overline{\{e\}_r} \xrightarrow{(p \bullet o) \lhd \overline{\{v\}_r}} \mathbf{0}} \ (inv) \qquad\qquad \frac{g_1 \xrightarrow{\alpha} s}{g_1 + g_2 \xrightarrow{\alpha} s} \ (choice)$$

$$\frac{s \xrightarrow{p \bullet o \lfloor \sigma \uplus \{x \mapsto \{v\}_r\} \rfloor \bar{w}\, \overline{\{v'\}_{r'}}} s' \qquad r'' \cdot \sigma \subseteq r}{[\{x\}^{r''}]\, s \xrightarrow{p \bullet o \lfloor \sigma \rfloor \bar{w}\, \overline{\{v'\}_{r'}}} s' \cdot \{x \mapsto \{v\}_r\}} \ (del_{sub}) \qquad\qquad \frac{s \xrightarrow{\dagger k} s'}{[k]\, s \xrightarrow{\dagger} [k]\, s'} \ (del_{kill})$$

$$\frac{s \xrightarrow{\alpha} s' \quad d \neq \{x\}^r \quad d \notin d(\alpha) \quad s \downarrow_{kill} \Rrightarrow \alpha = \dagger, \dagger k}{[d]\, s \xrightarrow{\alpha} [d]\, s'} \ (del_{pass}) \qquad \frac{s \xrightarrow{\alpha} s' \quad x \notin d(\alpha)}{[\{x\}^r]\, s \xrightarrow{\alpha} [\{x\}^r]\, s'} \ (x_{pass})$$

$$\frac{s_1 \xrightarrow{(p \bullet o) \rhd \bar{w}} s_1' \qquad s_2 \xrightarrow{(p \bullet o) \lhd \bar{v}} s_2' \qquad \mathcal{M}(\bar{w}, \bar{v}) = \sigma \qquad \neg(s_1 \mid s_2 \downarrow^{|\sigma|}_{p \bullet o, \bar{v}})}{s_1 \mid s_2 \xrightarrow{p \bullet o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \mid s_2'} \ (com)$$

$$\frac{s_1 \xrightarrow{p \bullet o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \qquad \neg(s_2 \downarrow^{|\mathcal{M}(\bar{w}, \bar{v})|}_{p \bullet o, \bar{v}})}{s_1 \mid s_2 \xrightarrow{p \bullet o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \mid s_2} \ (par_{conf}) \qquad\qquad \frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \mid s_2 \xrightarrow{\dagger k} s_1' \mid halt(s_2)} \ (par_{kill})$$

$$\frac{s_1 \xrightarrow{\alpha} s_1' \qquad \alpha \neq (p \bullet o \lfloor \sigma \rfloor \bar{w}\, \overline{\{v\}_r}), \dagger k}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2} \ (par_{pass}) \qquad\qquad \frac{s \xrightarrow{\alpha} s'}{\{\!\{s\}\!\} \xrightarrow{\alpha} \{\!\{s'\}\!\}} \ (prot)$$

$$\frac{s \equiv s_1 \quad s_1 \xrightarrow{\alpha} s_2 \quad s_2 \equiv s'}{s \xrightarrow{\alpha} s'} \ (cong)$$

tion of a communication over $p \bullet o$ with receive parameters $\bar{w}$ and matching values $\overline{\{v\}_r}$ and with substitution σ to be still applied, $\dagger$ and $p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \overline{\{v\}_r}$ denote *computational steps* corresponding to taking place of forced termination and communication (without pending substitutions), respectively. Hence, a *computation* from a closed service s_0 is a sequence of connected transitions of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} s_3 \ldots$$

where, for each i, α_i is either $p \bullet o \lfloor \emptyset \rfloor \bar{w}\, \overline{\{v\}_r}$ or $\dagger$, and s_i is called *reduct* of s_0.

We comment on salient points. Activity **kill**(k) forces termination of all unprotected parallel activities (rules *(kill)* and *(par$_{kill}$)*) inside an enclosing $[k]$, that stops the killing effect by turning the transition label $\dagger k$ into $\dagger$ (rule *(del$_{kill}$)*). Existence of such delimitation is ensured by the assumption that the semantics is only defined for closed services. Sensitive code can be protected from killing by putting it into a protection $\{\!\{\_\}\!\}$; this way, $\{\!\{s\}\!\}$ behaves like s (rule *(prot)*). Similarly, $[d]\, s$ behaves like s, except when the transition label α contains d or when a kill activity is active in s and α does not correspond to a kill

activity (rules *(del$_{pass}$)* and *(x$_{pass}$)*): in such cases the transition should be derived by using rules *(del$_{kill}$)* or *(del$_{sub}$)*. In other words, kill activities are executed *eagerly*. A service invocation can proceed only if the expressions in the argument can be evaluated and their regions do not contain variables (rule *(inv)*). A receive activity offers an invocable operation along a given partner name (rule *(rec)*). The execution of a receive permits to take a decision between alternative behaviours (rule *(choice)*). Communication can take place when two parallel services perform matching receive and invoke activities (rules *(com)*). Communication generates a substitution that is recorded in the transition label (for subsequent application), rather than a silent transition as in most process calculi. If more then one matching is possible the receive that needs fewer substitutions is selected to progress (rules *(com)* and *(par$_{conf}$)*). This mechanism permits to correlate different service communications thus implicitly creating interaction sessions and can be exploited to model the precedence of a service instance over the corresponding service specification when both can process the same request. A substitution $\{x \mapsto \{v\}_r\}$ for a variable x is applied to a term (rule *(del$_{sub}$)*) when the delimitation for x is encountered, i.e. the whole scope s of x is determined, provided that the region annotations of the variable declaration and of the substituent datum v do comply i.e. $r' \cdot \sigma \subseteq r$. This condition also means that as a value is received it gets annotated with a smaller region. The substitution for x is then applied to s and x disappears from the term and cannot be reassigned a value. Execution of parallel services is interleaved (rule *(par$_{pass}$)*), but when a kill activity or a communication is performed. Indeed, the former must trigger termination of all parallel services (according to rule *(par$_{kill}$)*), while the latter must ensure that the receive activity with greater priority progresses (rule *(com)* and *(par$_{conf}$)*). The last rule states that structurally congruent services have the same transitions.

3 Main Results

Our main results are standard and state that well-typedness is preserved along computations (*subject reduction*) and that well-typed services do respect region annotations (*type safety*). Together, these results imply the *soundness* of our theory, i.e. no violation of data regions will ever occur during the evolution of well-typed services. The formal account of these results follow. To save space, we only outline the techniques used in the proofs and refer the interested reader to [15] for a full account.

For the proof of subject reduction, we need some standard lemmata concerning substitution and weakening. The substitution lemma handles the substitution of partner variables with partner names. Application of a substitution σ to a type environment Γ, written $\Gamma \cdot \sigma$, is defined only when $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\Gamma) = \emptyset$ and, for each $x \mapsto \{v\}_r \in \sigma$, has the effect of replacing every occurrence of x in the regions of Γ with v, i.e.

$$\emptyset \cdot \{x \mapsto \{v\}_r\} = \emptyset \quad \text{and} \quad (\Gamma, \{y : r'\}) \cdot \{x \mapsto \{v\}_r\} = \Gamma \cdot \{x \mapsto \{v\}_r\}, \{y : (r' \cdot \{x \mapsto \{v\}_r\})\}.$$

Lemma 1 (Substitution Lemma). *If* $\Gamma, \{x : r\} \vdash s \succ \Gamma', \{x : r'\} \vdash s'$ *and* $\sigma = \{x \mapsto \{v\}_{r''}\}$*, then* $\Gamma \cdot \sigma \vdash s \cdot \sigma \succ \Gamma' \cdot \sigma \vdash s' \cdot \sigma$.

Proof. By induction on the length of the type derivation, with a case analysis on the last rule used in the derivation. □

Lemma 2 (Weakening Lemma). *Let* $\Gamma' \vdash s' \succ \Gamma \vdash s$ *and* $x \notin \mathrm{bd}(s)$*, then* $\Gamma' + \{x : r\} \vdash s' \succ \Gamma + \{x : r\} \vdash s$*.*

Proof. By a straightforward induction on the length of the type derivation, with a case analysis on the last used rule, and by exploiting the fact that extending Γ by adding $\{x : r\}$ does not affect the premise of rule *(t-inv)*. □

We also need a few auxiliary results. The first one states that function *halt*(_) preserves well-typedness and can be easily proved by induction on the definition of *halt*(_).

Lemma 3. *If* s *is well-typed then* $halt(s)$ *is well-typed.*

The next results establish well-typedness preservation by the structural congruence and by the labelled transition relation, respectively. We use the following preorder $\sqsubseteq$ on type environments: we write $\Gamma \sqsubseteq \Gamma'$ if there exists a Γ'' such that $\Gamma + \Gamma'' = \Gamma'$.

Lemma 4. *If* $\Gamma' \vdash s'_1 \succ \Gamma \vdash s_1$ *and* $s_1 \equiv s_2$ *then there exists a raw service* s'_2 *such that* $\Gamma' \vdash s'_2 \succ \Gamma \vdash s_2$*.*

Proof. By a straightforward induction on the derivation of $s_1 \equiv s_2$. □

Theorem 1. *If* $\Gamma'_1 \vdash s'_1 \succ \Gamma_1 \vdash s_1$ *and* $s_1 \xrightarrow{\alpha} s_2$ *then there exist a raw service* s'_2 *and two type environments* Γ_2 *and* Γ'_2 *such that* $\Gamma_2 \sqsubseteq \Gamma_1$*,* $\Gamma'_1 \sqsubseteq \Gamma'_2$ *and* $\Gamma'_2 \vdash s'_2 \succ \Gamma_2 \vdash s_2$*.*

Proof. By induction on the length of the inference of $s_1 \xrightarrow{\alpha} s_2$, with a case analysis on the last used rule. □

We can now easily prove that well-typedness is preserved along computations.

Corollary 1 (Subject Reduction). *If service* s *is well-typed and* $s \xrightarrow{\alpha} s'$ *with* $\alpha \in \{\dagger, \hat{n}\lfloor\emptyset\rfloor\bar{w}\,\overline{\{v\}_r}\}$*, then* s' *is well-typed.*

To characterize the errors that our type system can capture we use predicate $\Uparrow$: $s \Uparrow$ holds true when s can immediately generate a runtime error. This happens when in an active context there is an invoke activity on a partner not included in the region annotation of some of the expressions argument of the activity. Formally, $\Uparrow$ is defined as the least predicate closed under the following rules

$$\frac{\exists\, r' \in \bar{r}\,.\;\; p \notin r'}{p \bullet o!\overline{\{e\}_r} \Uparrow} \qquad \frac{s \Uparrow}{\mathbb{A}[\![s]\!] \Uparrow} \qquad \frac{s \equiv s' \quad s' \Uparrow}{s \Uparrow}$$

We remark that the runtime errors that our type discipline can capture are related to the policies for the exchange of data. We skip such runtime errors as 'unproper use of variables' (e.g. in $x \bullet o!\bar{v}$ the variable x is not replaced by a partner name) that can be easily dealt with standard type systems.

We can now prove that well-typed services do respect region annotations, from which it follows that the type system and the operational semantics are *sound*.

Theorem 2 (Type Safety). *If* s *is a well-typed service then* $s \Uparrow$ *holds false.*

Proof. By induction on the derivation of $s \Uparrow$, with a case analysis on the last used rule, we prove that if $s \Uparrow$ then s is not well-typed, from which the thesis follows. □

Corollary 2 (Type Soundness). *Let* s *be a well-typed service. Then* $s' \Uparrow$ *holds false for every reduct* s' *of* s*.*

Proof. Corollary 1 can be repeatedly applied to prove that s' is well-typed, then Theorem 2 permits to conclude. □

4 A case study by W3C

In this section we illustrate an application of our framework to a simplified but realistic electronic marketplace scenario inspired by [16]. To show usefulness of our approach, we focus on the central part of the protocol where sensitive data are exchanged, i.e. we omit the initial bartering and the concluding interactions, and expand the part relative to the payment process. We will write $Z \triangleq s$ to assign a symbolic name Z to service s.

Suppose a service *buyer* invokes a service *seller* to purchase some goods. Once *seller* has received an order request, it sends back the partner name of the service *credit_agency* to be used for the payment. *buyer* can then check the information on *credit_agency* and, possibly, confirm the payment by sending its credit card data to *seller*. In this case, *seller* forwards the received data to *credit_agency* and passes the order to the service *shipper*. In the end, the whole system is

$$EMP \triangleq buyer \mid credit\_agency \mid [p_{sh}]\ (seller \mid shipper)$$

When fixing the policies for data exchange, services can (safely) assume that, at the outset, partner names p_s, p_{ca} and p_b are publicly available for invoking *seller*, *credit_agency* and *buyer*, respectively. Instead, the partner name p_{sh} for invoking *shipper* is private and only shared with *seller*. Of course, due to the syntactical restrictions, the 'locality' condition for partner names is preserved by the semantics. Thus, the initials assumptions remain true forever.

The *buyer* service is defined as

$$\begin{aligned} buyer \triangleq\ & [id]\,(\ p_s \cdot o_{ord}!\langle \{id\}_{\{p_s,p_b\}}, p_b, order\rangle \\ & \quad \mid\ [x_{ca}]\, p_b \cdot o_{ca\_info}?\langle id, x_{ca}\rangle . \\ & \qquad [p,o]\,(\ p \cdot o!\langle\rangle \mid p \cdot o?\langle\rangle . p_s \cdot o_{pay}!\langle \{id\}_{\{p_s,p_b\}}, \{cc\_data\}_{\{p_s,x_{ca}\}}\rangle \\ & \qquad\quad + \ p \cdot o?\langle\rangle . p_s \cdot o_{canc}!\langle \{id\}_{\{p_s,p_b\}}\rangle\ )\ ) \end{aligned}$$

The endpoint $p_s \cdot o_{ord}$ is used for invoking the seller service and transmitting the order together with the *buyer*'s partner name p_b. The (restricted) name *id* represents the order identifier and is used for correlating all those service interactions that logically form a same session relative to the processing of *order*. For example, the specification of *buyer* could be slightly modified to allow the service to simultaneously make multiple orders: of course, although all such parallel threads must use the same partner p_s to interact with *seller*, they can exploit different order identifiers as a means to correlate messages belonging to different interaction sessions. The type attached to *id* only allows *buyer* and *seller* to exchange and use it, since they are the only services that can receive along p_s and p_b. Instead, p_b comes without an attached policy, since it is publicly known (it is transmitted to indicate the service making the invocation for the call-back operation). For simplicity, also *order* has no attached policy; thus, it could be later on communicated to any other service. Variable x_{ca} is used to store the partner name of the credit agency service to be used to possibly finalize the purchase and also to implement the policy for *buyer*'s credit card data. After the information on the credit agency service are verified, *buyer* sends a message to *seller* either to confirm or to cancel the order. This is simply modelled as an internal non-deterministic choice, by exploiting the private endpoint $p \cdot o$ (a more precise model can be obtained by exploiting the encodings shown in [9]).

The *seller* service is defined as

$$\begin{aligned} \mathit{seller} \triangleq *[x_b, x_{id}, x_{ord}, k]\, p_s \cdot o_{ord}?\langle x_{id}, x_b, x_{ord}\rangle. \\ (\, x_b \cdot o_{ca\_info}!\langle \{x_{id}\}_{\{x_b\}}, p_{ca}\rangle \\ \mid\ [x_{cc}]\, p_s \cdot o_{pay}?\langle x_{id}, x_{cc}\rangle.(\, p_{ca} \cdot o_{cr\_req}!\langle x_{ord}, \{x_{cc}\}_{\{p_{ca}\}}\rangle \\ \qquad\qquad \mid\ p_{sh} \cdot o_{sh\_req}!\langle x_{ord}\rangle\,) \\ \mid\ p_s \cdot o_{canc}?\langle x_{id}\rangle.\mathbf{kill}(k)\,) \end{aligned}$$

Once *seller* receives an order along $p_s \cdot o_{ord}$, it creates one specific instance that sends back to *buyer* (via x_b) the partner name p_{ca} of the credit agency service where the payment will be made. Whenever the seller instance receives the credit card data correlated to x_{id}, it forwards them to *credit_agency* and passes the order to the (internal) shipper service. Instead, if *buyer* demands cancellation of the order, the corresponding instance of *seller* is immediately terminated. Name k is used to delimit the effect of the kill activity only to the relevant instance.

The remaining two services are defined as

$$\mathit{credit\_agency} \triangleq *[x, y]\, p_{ca} \cdot o_{cr\_req}?\langle x, y\rangle. < \texttt{execute\_the\_payment} >$$

$$\mathit{shipper} \triangleq *[z]\, p_{sh} \cdot o_{sh\_req}?\langle z\rangle. < \texttt{process\_the\_order} >$$

Let now consider the type inference phase. Service *seller* gets annotated as follows:

$$\begin{aligned} \mathit{seller}' \triangleq *[\{x_b\}^{\{p_s\}}]\,[\{x_{id}\}^{\{p_s, x_b\}}, \{x_{ord}\}^{\top}, k]\, p_s \cdot o_{ord}?\langle x_{id}, x_b, x_{ord}\rangle. \\ (\, x_b \cdot o_{ca\_info}!\langle \{x_{id}\}_{\{x_b\}}, p_{ca}\rangle \\ \mid\ [\{x_{cc}\}^{\{p_s, p_{ca}\}}]\, p_s \cdot o_{pay}?\langle x_{id}, x_{cc}\rangle.(\, p_{ca} \cdot o_{cr\_req}!\langle x_{ord}, \{x_{cc}\}_{\{p_{ca}\}}\rangle \\ \qquad\qquad \mid\ p_{sh} \cdot o_{sh\_req}!\langle x_{ord}\rangle\,) \\ \mid\ p_s \cdot o_{canc}?\langle x_{id}\rangle.\mathbf{kill}(k)\,) \end{aligned}$$

The type inference has the task of checking consistency of region annotations of the arguments occurring within invoke activities and that of deriving the annotations for variable declarations. As regards consistency, there are only two explicitly typed expressions used as arguments of invoke activities, i.e. x_{id} and x_{cc}, and their types $\{x_b\}$ and $\{p_{ca}\}$ satisfy the consistency constraint (see rule *(t-inv)*). The remaining expressions occurring as arguments of invoke activities, i.e. the only x_{ord}, have implicitly assigned type $\top$ (indeed, recall that we assumed that an untagged e stands for $\{e\}_{\top}$) and are thus trivially consistent. As regards type derivation, when a variable is put in the environment (rule *(t-del$_{var}$)*), it is assigned type $\emptyset$. Later on, when a variable is used as an argument of an invoke or receive, its type can possibly be enriched (rules *(t-inv)* and *(t-rec)*). Thus, at the end of the inference, declaration of variable x_b, that is only used in $p_s \cdot o_{ord}?\langle x_{id}, x_b, x_{ord}\rangle$, will have assigned region $\{p_s\}$ (application of rule *(t-rec)*). Instead, declaration of x_{ord} has assigned type $\top$ (rule *(t-inv)* is used) while that of x_{cc} has assigned type $\{p_s, p_{ca}\}$ and, similarly, declaration of x_{id} gets annotated with $\{p_s, x_b\}$ (in both cases rules *(t-inv)* and *(t-rec)* are used). Notably, in *seller*′, delimitation $[\{x_b\}^{p_s}]$ does not commute any longer with delimitations $[\{x_{id}\}^{\{p_s, x_b\}}, \{x_{ord}\}^{\top}, k]$ (otherwise the service would become opened).

The variable declarations of the other services are annotated in a trivial way: x_{ca} with $\{p_b\}$, x and y with $\{p_{ca}\}$, and z with $\{p_{sh}\}$ (we assume that *credit_agency* and *shipper* do not re-transmit the received data). Thus, if we call *buyer*′, *credit_agency*′ and *shipper*′ the other typed services, then the system resulting from the type inference is

$$buyer' \mid credit\_agency' \mid [p_{sh}]\ (seller' \mid shipper')$$

After some computation steps, the system can become

$$\begin{array}{l}
[id]\,(\ p_s \bullet o_{pay}!\langle \{id\}_{\{p_s,p_b\}}, \{cc_data\}_{\{p_s,p_{ca}\}}\rangle \ \mid\ [p_{sh}]\,(\ seller' \ \mid \\
\qquad [k, \{x_{cc}\}^{\{p_s,p_{ca}\}}]\,(\ p_s \bullet o_{pay}?\langle id, x_{cc}\rangle.(\ p_{ca} \bullet o_{cr_req}!\langle order, \{x_{cc}\}_{\{p_{ca}\}}\rangle \\
\qquad\qquad\qquad\qquad \mid\ p_{sh} \bullet o_{sh_req}!\langle order\rangle\ ) \\
\qquad\qquad\qquad \mid\ p_s \bullet o_{canc}?\langle id\rangle.\mathbf{kill}(k)\ ) \\
\qquad \mid\ *\,[\{x\}^{\{p_{ca}\}}, \{y\}^{\{p_{ca}\}}]\ p_{ca} \bullet o_{cr_req}?\langle x, y\rangle.\ <\texttt{execute_the_payment}> \\
\qquad \mid\ *\,[\{z\}^{\{p_{sh}\}}]\ p_{sh} \bullet o_{sh_req}?\langle z\rangle.\ <\texttt{process_the_order}>\ )\ )
\end{array}$$

Thus, after $buyer'$ sends the credit card data, we get

$$\begin{array}{l}
[id, p_{sh}]\,(\ seller' \\
\qquad \mid\ [k]\,(\ p_{ca} \bullet o_{cr_req}!\langle order, \{cc_data\}_{\{p_{ca}\}}\rangle \mid p_{sh} \bullet o_{sh_req}!\langle order\rangle \\
\qquad\qquad \mid\ p_s \bullet o_{canc}?\langle id\rangle.\mathbf{kill}(k)\ ) \\
\qquad \mid\ *\,[\{x\}^{\{p_{ca}\}}, \{y\}^{\{p_{ca}\}}]\ p_{ca} \bullet o_{cr_req}?\langle x, y\rangle.\ <\texttt{execute_the_payment}> \\
\qquad \mid\ *\,[\{z\}^{\{p_{sh}\}}]\ p_{sh} \bullet o_{sh_req}?\langle z\rangle.\ <\texttt{process_the_order}>\ )
\end{array}$$

At this point, $seller'$ can safely communicate credit card data of $buyer'$ to $credit\_agency'$ and, then, forward the order to $shipper'$.

Suppose now that service $seller'$ also contains such a malicious invocation as $p_{sh} \bullet o!\langle \ldots, \{x_{cc}\}_r, \ldots\rangle$. In order to successfully pass the type inference phase, it should be that $p_{sh} \in r$ (otherwise rule *(t-inv)* could not be applied). Therefore, in the resulting typed service we would have the variable declaration $[\{x_{cc}\}^{r'}]$, with $r \subseteq r'$. Now, communication with $buyer'$ would be blocked by the runtime checks because the datum is tagged as $\{cc\_data\}_{\{p_s,p_{ca}\}}$, and $p_{sh} \in r \subseteq r'$ implies that $r' \not\subseteq \{p_s, p_{ca}\}$.

5 Concluding remarks

We have introduced a first analytical tool for checking that COWS specifications enjoy some desirable properties concerning the partners, and hence the services, that can safely access any given datum and, in that respect, do not manifest unexpected behaviors. Our type system is quite simple: types are just sets and operations on types are union, intersection, subset inclusion, etc. The language operational semantics only involves types in efficiently implementable checks, i.e. subset inclusions. While implementation of our framework is currently in progress, we are also working on the definition of a completely static variant where all dynamic checks have been moved to the static phase.

The types used in this paper are essentially inspired by the 'region types' for Confined-λ of [17] and for global computing calculi of [10]. There are however some noticeable differences. In fact, COWS permits describing not necessarily distributed systems and exchanging heterogeneous data along endpoints, which calls for a more dynamic typing mechanism than communication channels. Moreover, COWS permits annotating only the relevant data while Confined-λ requires typing any constant, function and channel. The group types, originally proposed for the Ambients calculus [18] and then recast to the π-calculus [19], have purposes similar to our region annotations,

albeit they are only used for constraining the exchanges of ambient and channel names. Confinement has been also explored in the context of Java, and related calculi, for confining classes and objects within specific packages [20, 21].

More expressive type disciplines based, e.g., on session types and behavioural types are emerging as powerful tools for taking into account behavioural and non-functional properties of computing systems. In the case of services, they could permit to express and enforce many relevant policies for, e.g., regulating resources usage, constraining the sequences of messages accepted by services, ensuring service interoperability and compositionality, guaranteeing absence of deadlock in service composition, checking that interaction obeys a given protocol. Some of the studies developed for the π-calculus (see e.g. [22–26]) are promising starting points, but they need non trivial adaptations to deal with all COWS peculiar features. For example, one of the major problems we envisage concerns the treatment of killing and protection activities, that are not commonly used in process calculi.

Many efforts have been devoted to develop analytical tools for SOC foundational languages. Some works study mechanisms for comparing global descriptions (i.e. choreographies) and local descriptions (i.e. orchestrations) of a same system. Means to check conformance of these different views have been defined in [5] and, by relying on session types, in [22]. COWS, instead, only considers service orchestration and focuses on modelling the dynamic behaviour of services without the limitations possibly introduced by a layer of choreography. Some other works [27, 28] have concentrated on modelling web transactions and on studying their properties in programming languages based on the π-calculus, while [29, 30] formalize long running transactions with special care for the *Sagas* mechanism [31]. A type system specifying security policies for orchestration has been introduced in [32] for a very basic formalism based on the λ-calculus. Finally, a type system for checking compliance between (simplified) WS-BPEL terms and the associated WSDL documents has been defined in [7].

Acknowledgements. We thank the anonymous referees for their useful comments.

References

1. Schneider, F.B., Morrisett, G., Harper, R.: A language-based approach to security. In *Informatics: 10 Years Ahead, 10 Years Back*, *LNCS* 2000, pp. 86–101, 2000.
2. Brogi, A., Canal, C., Pimentel, E., Vallecillo, A.: Formalizing web service choreographies. *ENTCS*, 105:73–94, Elsevier, 2004.
3. Viroli, M.: Towards a formal foundational to orchestration languages. *ENTCS*, 105:51–71. Elsevier, 2004.
4. Geguang, P., Xiangpeng, Z., Shuling, W., Zongyan, Q.: Towards the semantics and verification of bpel4ws. In *WLFM*. Elsevier, 2005.
5. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In *COORDINATION*, *LNCS* 4038, pp. 63–81, 2006.
6. Laneve, C., Padovani, L.: Smooth orchestrators. In *FoSSaCS*, *LNCS* 3921, pp. 32–46, 2006.
7. Lapadula, A., Pugliese, R., Tiezzi, F.: A WSDL-based type system for WS-BPEL. In *COORDINATION*, *LNCS* 4038, pp. 145–163, 2006.
8. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: a calculus for service oriented computing. In *ICSOC*, *LNCS* 4294, pp. 327–338, 2006.

9. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In *ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
10. De Nicola, R., Gorla, D., Pugliese, R.: Confining data and processes in global computing applications. *Science of Computer Programming*, 63:57–87, 2006.
11. OASIS. Web Services Business Process Execution Language Version 2.0. Technical report, WS-BPEL TC OASIS, August 2006. `http://www.oasis-open.org/`.
12. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2007. `http://rap.dsi.unifi.it/cows`.
13. Goguen, H.: Typed operational semantics. In *Typed Lambda Calculi and Applications*, *LNCS* 902, pp. 186–200, 1995.
14. Merro, M., Sangiorgi, D.: On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.
15. Lapadula, A., Pugliese, R., Tiezzi, F.: Regulating data exchange in service oriented applications (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2007. `http://rap.dsi.unifi.it/cows`.
16. Ross-Talbot, S., Fletcher, T.: Web services choreography description language: Primer (working draft). Technical report, W3C, June 2006.
17. Kirli, Z.D.: Confined mobile functions. In *CSFW*, *IEEE*, pp. 283–294, 2001.
18. Cardelli, L., Ghelli, G., Gordon, A.D.: Types for the ambient calculus. *Inf. Comput.*, 177(2):160–194, 2002.
19. Cardelli, L., Ghelli, G., Gordon, A.D.: Secrecy and group creation. *Inf. Comput.*, 196(2):127–155, 2005.
20. Vitek, J., Bokowski, B.: Confined types in java. *SPE*, 31(6):507–532. Wiley, 2001.
21. Zhao, T., Palsber, J., Vitek, J.: Lightweight confinement for featherweight java. In *OOPSLA*, pp. 135–148. ACM Press, 2003.
22. Carbone, M., Honda, K., Yoshida, N.: A calculus of global interaction based on session types. In *DCM*, 2006. To appear as ENTCS, Elsevier.
23. Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *SecReT*, ENTCS. Elsevier, 2006.
24. Kobayashi, N.: Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, *LNCS* 2757, pp. 439–453, 2003.
25. Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.
26. Kobayashi, N., Suenaga, K., Wischik, L.: Resource usage analysis for the π-calculus. In *VMCAI*, *LNCS* 3855, pp. 298–312, 2006.
27. Laneve, C., Zavattaro, G.: Foundations of web transactions. In *FoSSaCS*, *LNCS* 3441, pp. 282–298, 2005.
28. Mazzara, M., Lucchi, R.: A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2006.
29. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In *POPL*, pp. 209–220. ACM, 2005.
30. Bruni, R., Butler, M., Ferreira, C., Hoare, T., Melgratti, H.C., Montanari, U.: Comparing two approaches to compensable flow composition. In *CONCUR*, *LNCS* 3653, pp. 383–397, 2005.
31. Garcia-Molina, H., Salem, K.: Sagas. In *SIGMOD*, pp. 249–259. ACM Press, 1987.
32. Bartoletti, M., Degano, P., Ferrari, G.: Security Issues in Service Composition. In *FMOODS*, *LNCS* 4037, pp. 1–16, 2006.