Advanced Mechanisms for Service Composition, Query and Discovery

Michele Boreale[1] and Mario Bravetti[2]

[1] Dip. di Sistemi e Informatica, Università di Firenze, Italy
[2] Dip. Scienze dell'Informazione, Università di Bologna, Italy

Abstract. One of the ultimate goals of Service Oriented Computing (SOC) is to provide support for the automatic on-demand discovery of basic functionalities that, once combined, correctly compute a user defined task. To this aim, it is necessary for services to come equipped with a computer-understandable interface that allow applications to match the provided functionalities with the user needs. In this context, a prominent issue concerns the compliance between the operations invoked by the client – the client protocol – and the operations executed by the service – the service protocol. Process calculi, the theoretical tools investigated in the Work Package 2 of SENSORIA, can contribute to the solution of this problem. The idea we present in this chapter is to describe the externally observable message-passing behaviour of services as process calculi expressions; following recently adopted terminology, we call this description the *service contract*. We show how, in certain cases, service contracts can be automatically extracted out of service behaviour, and how they can be used to formally check the compliance among the communication protocols of interacting services.

1 Introduction

Service Oriented Computing (SOC) is based on services, intended as autonomous and heterogeneous components that can be published and discovered via standard interface languages and publish/discovery protocols. Web Services is the most prominent service oriented technology: Web Services publish their interface expressed in the Web Service Description Language (WSDL); they are discovered through the UDDI protocol, and they are invoked using SOAP. Even if one of the declared goal of Web Services is to support the automatic discovery of services, this is not yet practically achieved. In SENSORIA, we have addressed this problem by considering how to: (a) actually extracting a manageable description of the service interface (contract) out of a reasonably detailed service specification, and (b) guaranteeing that the services retrieved from a repository behave correctly according to the needs of the user and of the other retrieved services. In other terms, the retrieved services and the client invocation protocol should be compliant/complementary. For instance, it should be possible to check whether the overall composition of the client protocol with the invoked services is stuck-free.

In order to be able to perform this kind of checks, it is necessary for the services to expose in their interface also the description of their expected behaviour. In general, a service interface description language can expose both *static* and *dynamic* information. The former deals with the signature (name and type of the parameters) of the invocable operations; the latter deals with the correct order of invocation of the provided operations in order to correctly complete a session of interaction. The WSDL, which is the standard Web Services interface description language is basically concerned just with static information.

The aim of this paper is to report about process-algebraic techniques that could be effectively exploited in order to describe also the dynamic part of service interfaces. The choice of process-algebraic techniques for service interface descriptions is a natural one, as demonstrated also by other work by Fournet et al. [11] and Carpineti et al. [6]. The former introduces the notion of *contract* as "interface that specify the externally visible message passing behaviour" of processes, the latter refines this notion of contract considering a more specific client-service scenario.

Following the terminology adopted in this book, we call *service contract* the dynamic part of the service interface. More precisely, the service contract should describe the sequence of input/output operations that the service intends to execute within a session of interaction with other services. Let us consider two major aspects that must be addressed before service contracts become an effective technique for service publication, discovery and composition.

1. **Contract as abstraction.** As reported above, a contract is informally defined as the externally visible message passing behaviour of a service. We formally define, using a process algebraic approach, how to extract the externally observable behaviour from the description of the actual behaviour of a service. The technique that we report is based on the notion of *abstraction context*, which indicates how to statically associate to service events the corresponding observation tags. The achieved abstraction should be enough informative to enable proofs of certain properties (e.g. safety ones) of the actual service.
2. **Contract-based service composition.** One of the most important aspect that the service contract technology should be able to address is *correctness of composition*: given any set of services, it should be possible to prove that their composition is correct knowing only their contracts, i.e. in the absence of complete knowledge about (the internal details of) the services behaviour. We formalize the notion of correct composition, and show which kind of information should be exposed by a service in the corresponding contract, in order to enable proofs of service composition correctness. The notion of correctness that we consider requires that all computations in a service composition may be extended in order to reach a final state in which all services have successfully completed their activities. In other terms, it is not possible for a service to wait indefinitely for another service to send or receive a message.

Even if the two above contract-based techniques are strictly related, they are concerned with two different levels of abstractions of the service behaviour. In the first case, in order to extract an appropriate contract from a service, it is necessary to start from a detailed description of its behaviour. In the second case, it is possible to abstract away from the values actually exchanged between the service(s) and the client; indeed, only the order of invocations is relevant. These two different levels of abstractions justify the use of two different calculi for service behaviour description. In order to investigate the *contract as abstraction* concept, we start from a Pi-calculus description, while in order to investigate *contract-based service composition*, we abstract away from value passing and we consider a CCS-like process calculus.

This chapter is structured as follows. Sections 2 and 3 discuss, respectively, the two basic aspects: *contract as abstraction* and *contract-based service composition*. Section 4 contain some concluding remarks. Details of the techniques reported in this chapter can be found in [1] as far as Section 2 is concerned, and in [3,5,4] as far as Section 3 is concerned.

2 Contract as abstraction

According to the Service Oriented Computing paradigm, services can be seen as processes that provide a set of functionalities. A client can invoke a given functionality by sending an appropriate message to the corresponding operation/channel on the service side, and then waiting for a reply message, containing the computed results. More sophisticated schemes that involve complex conversations between the invoker and the service, and between the service and third parties, are also possible. When considering a system composed by several parties, be them clients or services, one is often interested in describing a *choreography*. This is the overall behaviour of the system in terms of, say, invoked service operations together with their argument types, allowable orderings among such operations, and so on. In any case, since services interact via message-passing, they can be seen as processes of some "first-order" calculus, such as Pi- or Join-calculus. Approximating a first-order (Pi, Join) process by a simpler propositional (CCS, BPP, Petri nets,...) model, not explicitly featuring message-passing, can be understood as the operation of extracting a *contract* out of a service. We consider methods to perform such approximations statically. Specifically, we describe a type system to associate Pi-calculus processes with restriction-free CCS types; such types can be thought of as contracts. A process is shown to be in simulation relation with its types, hence safety properties that hold of the types also hold of the process.

2.1 The asynchronous Pi-calculus

Processes Let $\mathcal{N}$, ranged over by $a, b, c, \ldots, x, y, \ldots$, be a countable set of *names* and $\mathcal{T}ag$, ranged over by $\alpha, \beta, \ldots$, be a set of *tags* disjoint from $\mathcal{N}$; we assume

$\mathcal{T}ag$ also contains a distinct "unit" tag (). The set $\mathcal{P}$ of processes $P, Q, \ldots$ is defined as the set of terms generated by the following grammar:

$$P, Q ::= \overline{a}\langle b\rangle \mid \sum_{i\in I} a_i(b).P_i \mid \sum_{i\in I} \tau.P_i \mid \text{if } a = b \text{ then } P \text{ else } P \mid !a(b).P \mid (\nu a : \alpha)P \mid P|P\,.$$

This language is a variation on the asynchronous Pi-calculus. Please note that the restriction operator $(\nu a : \alpha)P$ creates a new restricted name a with initial scope P *and* assigns it a tag α.

In an output action $\overline{a}\langle b\rangle$, name a is the *subject* and b the *object* of the action. Similarly, in a replicated input prefix $!a(b).P$ and in $\sum_{i\in I} a_i(b).P_i$, the names a and a_i for $i \in I$ are said to occur in *input subject position*. Binders and alpha-equivalence arise as expected and processes are identified up to alpha-equivalence of names. Substitution of a with b in an expression e is denoted by $e[^b/_a]$. In what follows, $\mathbf{0}$ stands for the empty summation $\sum_{i\in\emptyset} \tau.P_i$. We shall sometimes omit the object parts of input and output actions, when not relevant for the discussion; e.g. $\overline{a}$ stands for an output action with subject a and an object left unspecified. Similarly, we shall omit tag annotations, writing e.g. $(\nu a)P$ instead of $(\nu a : \alpha)P$, when the identity of the tag is not relevant.

Operational semantics The (early-style) semantics of processes is given by the labelled transition system in Table 1. We let ℓ, ℓ', ... represent generic elements of $\mathcal{N} \cup \mathcal{T}ag$. A transitions label μ can be a free output, $\overline{a}\langle b\rangle$, a bound output, $(\nu b : \alpha)\overline{a}\langle b\rangle$, an input, $a(b)$, or a silent move, $\tau\langle \ell, \ell'\rangle$. We assume a distinct tag ι for decorating *internal* transitions (arising from conditional and internal chioce; see Table 1) and often abbreviate $\tau\langle \iota, \iota\rangle$ simply as τ. In the following we indicate by $\mathrm{n}(\mu)$ the set of all names in μ and by $\mathrm{fn}(\mu)$, the set of free names of μ, defined as expected. The rules are standard, except for the extra book-keeping required by tag annotation of bound output and internal actions. In particular, in (RES-TAU) bound names involved in a synchronization are hidden from the observer and replaced by the corresponding tags. Note that if we erase the tag annotation from labels we get exactly the usual labelled semantics of asynchronous Pi-calculus.

2.2 Γ-abstractions of processes

A *context* Γ is a finite partial function from names to tags, written $\Gamma = \{a_1 : \alpha_1, \cdots, a_n : \alpha_n\}$, with distinct a_i. In what follows $\Gamma \vdash a : \alpha$ means that $a : \alpha \in \Gamma$. A *tag sorting system* $\mathcal{E}$ is a finite subset of $\{\alpha[\beta] \mid \alpha, \beta \text{ are tags and } \alpha \neq ()\}$. Informally, $\alpha[\beta] \in \mathcal{E}$ means that subject names associated with tag α can carry object names associated with tag β. In what follows, if $\alpha[\beta_1]$, $\cdots$, $\alpha[\beta_n]$ are the only elements of $\mathcal{E}$ with subject α, we write $\alpha[\beta_1, \cdots, \beta_n] \in \mathcal{E}$.

A triple $(P, \Gamma, \mathcal{E})$, written $P_{\Gamma;\mathcal{E}}$, is called *Γ-abstraction* of P under $\mathcal{E}$. In what follows, we shall consider a fixed sorting system $\mathcal{E}$, and keep $\mathcal{E}$ implicit by writing P_Γ instead of $P_{\Gamma;\mathcal{E}}$. Next, we define a labeled transition system with process abstractions as states and transition labels λ, which can be output, $\overline{\alpha}\langle\beta\rangle$,

$$(\text{OUT})\ \ \overline{a}\langle b\rangle \xrightarrow{\ \overline{a}\langle b\rangle\ } \mathbf{0}$$

$$(\text{G-SUM})\ \textstyle\sum_{i\in I} a_i(b_i).P_i \xrightarrow{\ a_j(c)\ } P_j[c/b_j],\ j\in I \qquad (\text{I-SUM})\ \textstyle\sum_{i\in I} \tau.P_i \xrightarrow{\ \tau\ } P_j,\ j\in I$$

$$(\text{REP})\ !a(c).P \xrightarrow{\ a(b)\ } P[b/c]\,|\,!a(c).P \qquad (\text{COM})\ \frac{P \xrightarrow{\ \overline{a}\langle b\rangle\ } P' \quad Q \xrightarrow{\ a(b)\ } Q'}{P\,|\,Q \xrightarrow{\ \tau\langle a,b\rangle\ } P'\,|\,Q'}$$

$$(\text{CLOSE})\ \frac{P \xrightarrow{\ (\nu b:\beta)\overline{a}\langle b\rangle\ } P' \quad Q \xrightarrow{\ a(b)\ } Q'}{P\,|\,Q \xrightarrow{\ \tau\langle a,\beta\rangle\ } (\nu b:\beta)(P'\,|\,Q')} \qquad (\text{OPEN})\ \frac{P \xrightarrow{\ \overline{b}\langle a\rangle\ } P' \quad b\neq a}{(\nu a:\alpha)P \xrightarrow{\ (\nu a:\alpha)\overline{b}\langle a\rangle\ } P'}$$

$$(\text{IF-F})\ \text{if } a=b \text{ then } P \text{ else } Q \xrightarrow{\ \tau\ } Q,\ a\neq b \qquad (\text{IF-T})\ \text{if } a=a \text{ then } P \text{ else } Q \xrightarrow{\ \tau\ } P$$

$$(\text{PAR})\ \frac{P \xrightarrow{\ \mu\ } P' \quad \text{bn}(\mu)\cap \text{fn}(Q)=\emptyset}{P|Q \xrightarrow{\ \mu\ } P'|Q} \qquad (\text{RES})\ \frac{P \xrightarrow{\ \mu\ } P' \quad a\notin \text{n}(\mu)}{(\nu a:\alpha)P \xrightarrow{\ \mu\ } (\nu a:\alpha)P'}$$

$$(\text{RES-TAU})\ \frac{P \xrightarrow{\ \tau\langle \ell_1,\ell_2\rangle\ } P' \quad a\in\{\ell_1,\ell_2\} \quad \ell=\ell_1[\alpha/a] \quad \ell'=\ell_2[\alpha/a]}{(\nu a:\alpha)P \xrightarrow{\ \tau\langle \ell,\ell'\rangle\ } (\nu a:\alpha)P'}$$

Table 1. Operational semantics of Pi-calculus processes. Symmetric rules not shown.

input, $\alpha\langle\beta\rangle$ or annotated silent action, $\tau\langle\alpha,\beta\rangle$. The set of labels generated by this grammar is denoted by Λ. The labeled transition system is defined by the rules below. Here, μ_Γ denotes the result of substituting each $a\in \text{fn}(\mu)\cap\text{dom}(\Gamma)$ by $\Gamma(a)$ in μ. Informally, P_Γ represents the abstract behavior of P, once each concrete action μ has been mapped to an abstract action λ. Note that in both rule (A-OUT_N) and rule (A-INP_N) the context Γ grows with a new association $b:\beta$. In rule (A-INP_N), a tag for b is chosen among the possible tags specified in $\mathcal{E}$. Note that no type checking is performed by these rules, in particular (A-OUT_N) does not look up $\mathcal{E}$ to check that β can be carried by α.

$$(\text{A-OLD})\ \frac{P \xrightarrow{\ \mu\ } P' \quad \mu ::= \tau\langle\ell,\ell'\rangle | a(b) | \overline{a}\langle b\rangle \quad \text{n}(\mu)\subseteq\text{dom}(\Gamma) \quad \lambda=\mu_\Gamma}{P_\Gamma \xrightarrow{\ \lambda\ } {P'}_\Gamma}$$

$$(\text{A-OUT}_\text{N})\ \frac{P \xrightarrow{\ (\nu b:\beta)\overline{a}\langle b\rangle\ } P' \quad \Gamma\vdash a:\alpha}{P_\Gamma \xrightarrow{\ \overline{\alpha}\langle\beta\rangle\ } {P'}_{\Gamma,b:\beta}} \qquad (\text{A-INP}_\text{N})\ \frac{P \xrightarrow{\ a(b)\ } P' \quad \Gamma\vdash a:\alpha \quad \alpha[\beta]\in\mathcal{E} \quad b\notin\text{dom}(\Gamma)}{P_\Gamma \xrightarrow{\ \alpha\langle\beta\rangle\ } {P'}_{\Gamma,b:\beta}}$$

Simulation, bisimulation and modal logic Given a labelled transition system T with labels in Λ, *strong bisimulation* $\sim$ and simulation $\lesssim$ over states of T are defined as expected. The *closed* versions of simulation and bisimulation, written $\lesssim^\text{c}$ and $\sim^\text{c}$, respectively, are defined in a similar manner, but limited to silent transitions, i.e. transitions carrying labels of the kind $\tau\langle\alpha,\beta\rangle$. In the rest of the paper, we will sometimes make use of simple action-based modal logic formulae $\phi,\psi,...$ taken from modal mu-calculus [19] to formulate concisely properties of types or processes. In particular, a state s of T satisfies $\langle A\rangle\phi$, written $s\vDash\langle A\rangle\phi$,

if there is a transition $s \xrightarrow{\lambda} s'$ with $\lambda \in A$ and $s' \vDash \phi$. The interpretation of modality $\langle\langle A \rangle\rangle \phi$ is similar, but the phrase "a transition $s \xrightarrow{\lambda} s'$ with $\lambda \in A$" is changed into "a sequence of transitions $s \xrightarrow{\sigma} s'$ with $\sigma \in A^*$".

2.3 ccs$^-$ types

In the type system we propose, types are essentially CCS expressions whose behavior over-approximate the (abstract) process behavior.

The set $\mathcal{T}_{\text{CCS}}$ of types, ranged over by $\mathsf{T}, \mathsf{S}, \ldots$, is defined by the following syntax:

$$\mathsf{T} \ ::= \ \overline{\alpha}\langle\beta\rangle \mid \sum_{i \in I} \alpha_i \langle \beta_i \rangle . \mathsf{T}_i \mid \sum_{i \in I} \tau . \mathsf{T}_i \mid !\alpha\langle\beta\rangle . \mathsf{T} \mid \mathsf{T} | \mathsf{T}$$

where $\alpha, \alpha_i \neq ()$. The empty summation $\sum_{i \in \emptyset} \tau . \mathsf{T}_i$ will be often denoted by nil, and $\mathsf{T}_1 \mid \cdots \mid \mathsf{T}_n$ will be often written as $\prod_{i \in \{1, \cdots, n\}} \mathsf{T}_i$. As usual, we shall sometimes omit the object part of actions when not relevant for the discussion or equal to the unit tag (), writing e.g. $\overline{\alpha}$ and $\tau\langle\alpha\rangle$ instead of $\overline{\alpha}\langle\beta\rangle$ and $\tau\langle\alpha, \beta\rangle$.Types are essentially asynchronous, restriction-free CCS processes over the alphabet of actions Λ. The standard operational semantics of CCS, giving rise to a labelled transition system with labels in Λ, is assumed (see [1]).

The typing rules We let $\mathcal{E}$ be a fixed tag sorting system and Γ a context. Judgements of the type system are of the form $\Gamma \vdash_{\mathcal{E}} P : \mathsf{T}$. The rules of the type system are presented in Table 2. Notice that rule (T-INP) has been introduced with the sake of improving the readability of the system; indeed expanding (T-SUM) would result in a much longer and more complex rule.

A brief explanation of some typing rules follows. In rule (T-OUT), the output process $\overline{a}\langle b \rangle$ gives rise to the action $\overline{a}\langle b \rangle_\Gamma = \overline{\alpha}\langle\beta\rangle$, provided this action is expected by the tag sorting system $\mathcal{E}$. The type T of an input process depends on $\mathcal{E}$: in (T-INP) all tags which can be carried by α, the tag associated with the action's subject, contribute to the definition of the summation in T as expected. In the case of (T-REP), summation is replaced by a parallel composition of replicated types, which is behaviorally – up to strong bisimulation – the same as a replicated summation. The subtyping relation $\lesssim$ is the simulation preorder over $\mathcal{E}$, (T-SUB). The rest of the rules should be self-explanatory.

Results The subject reduction theorem below establishes an operational correspondence between the abstract behavior P_Γ and any type T that can be assigned to P under Γ.

Theorem 1 (subject reduction). *$\Gamma \vdash_{\mathcal{E}} P : \mathsf{T}$ and $P_\Gamma \xrightarrow{\lambda} P'_{\Gamma'}$ imply that there is T' such that $\mathsf{T} \xrightarrow{\lambda} \mathsf{T}'$ and $\Gamma' \vdash_{\mathcal{E}} P' : \mathsf{T}'$.*

As a corollary, we obtain that T simulates P_Γ: thanks to Theorem 1, it is easy to see that the relation $\mathcal{R} = \left\{ (P_\Gamma, \mathsf{T}) \,\middle|\, \Gamma \vdash_{\mathcal{E}} P : \mathsf{T} \right\}$ is a simulation relation.

Corollary 1. *Suppose $\Gamma \vdash_{\mathcal{E}} P : \mathsf{T}$. Then $P_\Gamma \lesssim \mathsf{T}$.*

$$\text{(T-INP)}\quad \frac{\Gamma \vdash a:\alpha \quad \alpha[\beta_1,\cdots,\beta_n]\in\mathcal{E} \quad \forall i\in\{1,\cdots,n\}:\ \Gamma, b:\beta_i \vdash_{\mathcal{E}} P:\mathsf{T}_i}{\Gamma \vdash_{\mathcal{E}} a(b).P : \sum_{i\in\{1,\cdots,n\}} \alpha\langle\beta_i\rangle.\mathsf{T}_i}$$

$$\text{(T-REP)}\quad \frac{\Gamma \vdash a:\alpha \quad \alpha[\beta_1,\cdots,\beta_n]\in\mathcal{E} \quad \forall i\in\{1,\cdots,n\}:\ \Gamma, b:\beta_i \vdash_{\mathcal{E}} P:\mathsf{T}_i}{\Gamma \vdash_{\mathcal{E}} !a(b).P : \prod_{i\in\{1,\cdots,n\}} !\alpha\langle\beta_i\rangle.\mathsf{T}_i}$$

$$\text{(T-GSUM)}\quad \frac{|I|\neq 1 \quad \forall i\in I:\ \Gamma\vdash_{\mathcal{E}} a_i(b_i).P_i : \sum_{j\in J_i}\alpha_i\langle\beta_j\rangle.\mathsf{T}_{ij}}{\Gamma\vdash_{\mathcal{E}} \sum_{i\in I} a_i(b_i).P_i : \sum_{i\in I, j\in J_i}\alpha_i\langle\beta_j\rangle\mathsf{T}_{ij}}$$

$$\text{(T-OUT)}\ \frac{\Gamma\vdash a:\alpha \quad \alpha[\beta]\in\mathcal{E} \quad \Gamma\vdash b:\beta}{\Gamma\vdash_{\mathcal{E}} \overline{a}\langle b\rangle : \overline{\alpha}\langle\beta\rangle} \qquad \text{(T-ISUM)}\ \frac{\forall i\in I:\ \Gamma\vdash_{\mathcal{E}} P_i:\mathsf{T}_i}{\Gamma\vdash_{\mathcal{E}} \sum_{i\in I}\tau.P_i : \sum_{i\in I}\tau.\mathsf{T}_i}$$

$$\text{(T-PAR)}\ \frac{\Gamma\vdash_{\mathcal{E}} P:\mathsf{T} \quad \Gamma\vdash_{\mathcal{E}} Q:\mathsf{S}}{\Gamma\vdash_{\mathcal{E}} P|Q:\mathsf{T}|\mathsf{S}} \qquad \text{(T-RES)}\ \frac{\Gamma, a:\alpha\vdash_{\mathcal{E}} P:\mathsf{T}}{\Gamma\vdash_{\mathcal{E}} (\nu a:\alpha)P:\mathsf{T}}$$

$$\text{(T-IF)}\ \frac{\Gamma\vdash_{\mathcal{E}} P:\mathsf{T} \quad \Gamma\vdash_{\mathcal{E}} Q:\mathsf{S}}{\Gamma\vdash_{\mathcal{E}} \text{if } a=b \text{ then } P \text{ else } Q : \tau.\mathsf{T}+\tau.\mathsf{S}} \qquad \text{(T-SUB)}\ \frac{\Gamma\vdash_{\mathcal{E}} P:\mathsf{T} \quad \mathsf{T}\lesssim\mathsf{S}}{\Gamma\vdash_{\mathcal{E}} P:\mathsf{S}}$$

Table 2. Typing rules for CCS$^-$ types.

A consequence of the previous result is that safety properties satisfied by a type are also satisfied by the processes that inhabit that type – or, more precisely, by their Γ-abstract versions. Consider the small logic defined in Section 2.2: let us say that $\phi\in\mathcal{L}$ is a *safety* formula if every occurrence of $\langle A\rangle$ and $\langle\langle A\rangle\rangle$ in ϕ is underneath an odd number of negations. The following proposition, follows from Corollary 1 and first principles.

Proposition 1. *Suppose $\Gamma\vdash_{\mathcal{E}} P:\mathsf{T}$ and ϕ is a safety formula, with $\mathsf{T}\vDash\phi$. Then $P_\Gamma\vDash\phi$.*

As a final remark on the type system, consider taking out rule (T-SUB): the new system can be viewed as a (partial) function that for any P computes a minimal type for P, that is, a subtype of all types of P (just read the rules bottom-up).

2.4 An example

A simple printing system is considered, where users are required to authenticate themselves before being allowed to print. For simplicity, only two levels of privileges are considered for users, authorized and non-authorized. Correspondingly, two sets of credentials are given: $\{c_i \mid i\in I\}$ (also written $\tilde{c}_i$) for authorized users and $\{c_j \mid j\in J\}$ (also written $\tilde{c}_j$) for non-authorized users, with $\tilde{c}_i\cap\tilde{c}_j=\emptyset$. Process A is an authentication server that receives from any client a credential

c, a private return channel r and an error channel e, and then sends both r and e to a credential-handling process T. If the client is not authorized, T will raise an error on channel e, otherwise T establishes a private connection between the client and the printer by creating a new communication link k and passing it to the client. Process C describes the cumulative behavior of all clients: C tries nondeterministically to authenticate using a credential c_l, $l \in I \cup J$, and then waits for a link to the printer on the private channel r, or for an error, on the private channel e. After printing or receiving an error, C's execution restarts. One expects that every printing request accompanied by authorized credentials will be satisfied, and that every print is preceded by an authentication request.

$$
\begin{aligned}
\mathit{Sys} &\triangleq (\nu a : \mathit{aut},\, \tilde{c}_i : \mathit{ok},\, \tilde{c}_j : \mathit{nok},\, M : (),\, \mathit{print} : \mathit{pr})\big(T \mid C \mid A \mid !\mathit{print}(d)\big)\\
T &\triangleq \textstyle\prod_{i\in I} !c_i(x,e).(\nu k : \mathit{key})\big(\overline{x}\langle k\rangle \mid k(d).\overline{\mathit{print}}\langle d\rangle\big) \mid \prod_{j\in J} !c_j(x,e).\overline{e}\\
A &\triangleq !a(c,r,e).\overline{c}\langle r,e\rangle\\
C &\triangleq (\nu i : \mathit{iter})\Big(\overline{i} \mid !i.(\nu r : \mathit{ret},\, e : \mathit{err})\big(\textstyle\sum_{l\in I\cup J} \tau.\overline{a}\langle c_l, r, e\rangle \mid r(z).((\overline{z}\langle M\rangle \mid \overline{i}) \;+\; e.\overline{i})\Big)
\end{aligned}
$$

Below, we analyze this system using CCS types. To ease the notation, we shall omit the unit tag () involved in inputs and outputs and write e.g. $\overline{\alpha}$ instead of $\alpha[()]$. We shall consider a calculus enriched with polyadic communication and values: these extensions are easy to accommodate. Consider the tag sorting system

$$
\begin{aligned}
\mathcal{E} = \{&\mathit{aut}[\mathit{ok}, \mathit{ret}, \mathit{err}],\ \mathit{aut}[\mathit{nok}, \mathit{ret}, \mathit{err}],\ \mathit{ok}[\mathit{ret}, \mathit{err}],\\
&\mathit{nok}[\mathit{ret}, \mathit{err}],\ \mathit{ret}[\mathit{key}],\ \mathit{pr}[()],\ \mathit{err}[\,],\ \mathit{key}[()],\ \mathit{iter}[\,]\}\ .
\end{aligned}
$$

It is easy to prove that $\emptyset \vdash_{\mathcal{E}} \mathit{Sys} : \mathsf{T}_T \mid \mathsf{T}_A \mid \mathsf{T}_C \mid !\mathit{pr} = \mathsf{T}$, where

$$
\begin{aligned}
\mathsf{T}_T &\triangleq !\mathit{ok}\langle \mathit{ret}, \mathit{err}\rangle.\big(\overline{\mathit{ret}}\langle \mathit{key}\rangle \mid \mathit{key}.\overline{\mathit{pr}}\big) \mid !\mathit{nok}\langle \mathit{ret}, \mathit{err}\rangle.\overline{\mathit{err}}\\
\mathsf{T}_A &\triangleq !\mathit{aut}\langle \mathit{ok}, \mathit{ret}, \mathit{err}\rangle.\overline{\mathit{ok}}\langle \mathit{ret}, \mathit{err}\rangle \mid !\mathit{aut}\langle \mathit{nok}, \mathit{ret}, \mathit{err}\rangle.\overline{\mathit{nok}}\langle \mathit{ret}, \mathit{err}\rangle\\
\mathsf{T}_C &\triangleq \overline{\mathit{iter}} \mid !\mathit{iter}.\big((\tau.\overline{\mathit{aut}}\langle \mathit{ok}, \mathit{ret}, \mathit{err}\rangle + \tau.\overline{\mathit{aut}}\langle \mathit{nok}, \mathit{ret}, \mathit{err}\rangle) \mid (\mathit{ret}\langle \mathit{key}\rangle.(\overline{\mathit{key}} \mid \overline{\mathit{iter}}) + \mathit{err}.\overline{\mathit{iter}})\big).
\end{aligned}
$$

Furthermore, it holds that

$$
\begin{aligned}
\mathsf{T} \vDash \phi &\triangleq \neg\langle\!\langle \Lambda - \{\mathit{nok}\langle \mathit{ret}, \mathit{err}\rangle, \mathit{aut}\langle \mathit{nok}, \mathit{ret}, \mathit{err}\rangle, \tau\langle \mathit{aut}, \mathit{nok}, \mathit{ret}, \mathit{err}\rangle\}\rangle\!\rangle\ \langle\overline{\mathit{err}}\rangle\\
\mathsf{T} \vDash \psi &\triangleq \neg\langle\!\langle \Lambda - \{\mathit{ok}\langle \mathit{ret}, \mathit{err}\rangle, \mathit{aut}\langle \mathit{ok}, \mathit{ret}, \mathit{err}\rangle, \tau\langle \mathit{aut}, \mathit{ok}, \mathit{ret}, \mathit{err}\rangle\}\rangle\!\rangle\ \langle\overline{\mathit{pr}}\rangle
\end{aligned}
$$

that is, *error* only arises from an authentication request containing non authorized credentials, and every print pr action is preceded by a successful authentication request. Both formulas express safety properties, hence Proposition 1 ensures that are both satisfied by the abstract process $\mathit{Sys}_\emptyset$.

2.5 Extensions

We discuss here the other type systems presented in [1]. We just outline the most relevant aspects and refer the interested reader to [1] for full details. Choreographies are generated by specific compositions of several actors, be them clients or

services. In [1] the type system reported here is adapted to choreographies, also called global or closed systems, to contrast them with the CCS$^-$ types applicable to *open processes*. In the closed case, types are BPP processes. Sufficient conditions are given under which a minimal BPP type can be computed that is bisimilar to a given process. For closed systems, it is possible to get rid of synchronization in types, obtaining a more direct and precise approximation of the behavior of P_Γ. Roughly, this is achieved by first associating each input prefix $a(b).P$ in the considered process with a labelled rewrite rule $\alpha[\beta] \xrightarrow{\alpha[\beta]} \alpha_1[\beta_1], ..., \alpha_n[\beta_n]$. Here, $\alpha[\beta]$ is the tag-representation of $a(b)$, according to Γ. The right-hand side of the rule is the multiset of observable outputs that can be triggered by a reduction involving the considered input prefix, according to Γ. An output action $\overline{a}\langle b\rangle$ of P is associated with a symbol $\alpha[\beta]$, that can be rewritten according to the rules. Types we obtain in this way are precisely Basic Parallel Processes (BPP, [7]). Results similar to the open case holds for the closed case. Furthermore, by restricting one's attention to (a generalization of) uniform receptive processes [18], it is possible to show that a bisimulation relation relates processes and their types: in this case, processes and their types satisfy the same abstract properties.

It is possible to partially extend the treatment of closed behaviour to the case of Join processes [10] (the Join open case requires additional care and we leave it for future work). The essential step one has to take, at the level of types, is moving from BPP to place/transition Petri nets (PN). Technically, this step is somehow forced by the presence of the join pattern construct in the calculus, which expresses multi-synchronization. In the context of infinite states transition systems [9,13], moving from BPP to PN corresponds precisely to moving from rewrite rules with a single nonterminal on the LHS (BPP) to rules with multisets of nonterminals on the LHS (PN).

3 Contract-based service composition

After having discussed, in the previous section, a general technique to extract a contract from service behaviours, we consider the problem of the exploitation of contracts in order to check the correctness of service compositions. In particular, in this section we take, as a starting point, services described by means of a calculus without value passing, similar to (a distributed version of) the CCS based language used in the previous section as the contract language. We express such services in the form of "plain" (language independent) contracts, i.e. finite labeled transition systems (obtained e.g. as the semantics of service terms), in order to check whether a service can be correctly introduced in a specific service composition. Service compositions, also called *choreographies*, are specified as a set of communicating contracts, in execution at specific locations, also called *roles* in choreographies. Thus, checking whether a service can be correctly introduced in a service composition coincides with checking whether its contract *refines* a contract sitting at a given location in the context of a choreography.

The remainder of this section is devoted to the formal definition of this refinement relation called *subcontract* relation. We first give a declarative definition of the subcontract relation induced by the properties that we want the refinement to satisfy. Then, we provide a testing based procedure to verify whether two contracts are in subcontract relation.

3.1 Behavioural Contracts

Definition 1. *A finite connected labeled transition system (LTS) with termination transitions is a tuple* $\mathcal{T} = (S, \mathcal{L}, \longrightarrow, s_h, s_0)$ *where* S *is a finite set of states,* L *is a set of labels, the transition relation* $\longrightarrow$ *is a finite subset of* $(S - \{s_h\}) \times (\mathcal{L} \cup \{\surd\}) \times S$ *such that* $(s, \surd, s') \in \longrightarrow$ *implies* $s' = s_h$, $s_h \in S$ *represents a halt state,* $s_0 \in S$ *represents the initial state, and it holds that every state in* S *is reachable (according to* $\longrightarrow$*) from* s_0*.*

In a finite connected LTS with termination transitions we use $\surd$ transitions (leading to the halt state s_h) to represent successful termination. On the contrary, if we get (via a transition different from $\surd$) into a state with no outgoing transitions (like, e.g., s_h) then we represent an internal failure or a deadlock.

We assume a denumerable set of action names $\mathcal{N}$, ranged over by $a, b, c, \ldots$. We use $\tau \notin \mathcal{N}$ to denote an internal (unsynchronizable) computation. We consider a denumerable set *Loc* of location names, ranged over by $l, l', l_1, l_2, \cdots$. In contracts the possible transition labels are the typical internal τ action and the input/output actions $a, \overline{a}$, where the outputs (as we will see when composing contracts) are directed to a destination address denoted by a location $l \in Loc$.

Definition 2. *A contract is a finite connected LTS with termination transitions, that is a tuple* $(S, \mathcal{L}, \longrightarrow, s_h, s_0)$*, where* $\mathcal{L} = \{a, \overline{a}_l, \tau \mid a \in \mathcal{N} \wedge l \in Loc\}$*.*

In the following we introduce a process algebraic representation for contracts by using a simple extension of basic CCS [12] with successful termination **1**.

Definition 3. *We consider a denumerable set of contract variables* Var *ranged over by* $X, Y, \cdots$*. The syntax of contracts is defined by the following grammar*

$$\begin{array}{lcl} C & ::= & \mathbf{0} \mid \mathbf{1} \mid \alpha.C \mid C{+}C \mid X \mid recX.C \\ \alpha & ::= & \tau \mid a \mid \overline{a}_l \end{array}$$

where $recX.\_$ *is a binder for the process variable* X*. The set of the contracts* C *in which all process variables are bound, i.e.* C *is a closed term, is denoted by* $\mathcal{P}_{con}$*. In the following we will often omit trailing "***1***" when writing contracts.*

The operational semantics of contracts is defined in terms of a transition system labeled by $\mathcal{L} = \{a, \overline{a}_l, \tau \mid a \in \mathcal{N} \wedge l \in Loc\}$ obtained by the rules in Table 3 (plus symmetric rule for choice), where we take λ to range over $\mathcal{L} \cup \{\surd\}$. In particular the semantics of a contract $C \in \mathcal{P}_{con}$ gives rise to a finite connected

$$\mathbf{1} \xrightarrow{\surd} \mathbf{0} \qquad \alpha.C \xrightarrow{\alpha} C$$

$$\frac{C \xrightarrow{\lambda} C'}{C+D \xrightarrow{\lambda} C'} \qquad \frac{C\{recX.C/X\} \xrightarrow{\lambda} C'}{recX.C \xrightarrow{\lambda} C'}$$

Table 3. Semantic rules for contracts (symmetric rules omitted).

LTS with termination transitions $(S, \mathcal{L}, \longrightarrow, \mathbf{0}, C)$ where S is the set of states reachable from C and $\longrightarrow$ includes only transitions between states of S.

In [4] we formalize the correspondence between contracts and terms of $\mathcal{P}_{con}$ by showing how to obtain from a contract $\mathcal{T} = (S, \mathcal{L}, \longrightarrow, s_h, s_0)$ a corresponding $C \in \mathcal{P}_{con}$ such that there exists a (surjective) homomorphism from the operational semantics of $\mathcal{C}$ to $\mathcal{T}$ itself.

In the following we use $C \xrightarrow{\lambda}$ to mean $\exists C' : C \xrightarrow{\lambda} C'$ and, given a string of labels $w \in \mathcal{L}^*$, that is $w = \lambda_1\lambda_2 \cdots \lambda_{n-1}\lambda_n$ (possibly empty, i.e., $w = \varepsilon$), we use $C \xrightarrow{w} C'$ to denote the sequence of transitions $C \xrightarrow{\lambda_1} C_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_{n-1}} C_{n-1} \xrightarrow{\lambda_n} C'$ (in case of $w = \varepsilon$ we have $C' = C$, i.e., $C \xrightarrow{\varepsilon} C$).

Definition 4. (Output persistence) *A contract $C \in \mathcal{P}_{con}$ is output persistent if, given $C \xrightarrow{w} C'$ with $C' \xrightarrow{\overline{a}_l}$, then: $C' \not\xrightarrow{\surd}$ and if $C' \xrightarrow{\alpha} C''$ with $\alpha \neq \overline{a}_l$ then also $C'' \xrightarrow{\overline{a}_l}$.*

The output persistence property states that once a contract decides to execute an output, its actual execution is mandatory in order to successfully complete the execution of the contract. This property typically hold in languages for the description of service behaviours or for service orchestrations (see e.g. WS-BPEL) in which output actions cannot be used as guards in external choices (see e.g. the `pick` operator of WS-BPEL which is an external choice guarded on input actions). In these languages, when a process instance or an internal thread decides to execute an output actions, it will have to complete such action before ending successfully. In the context of service descriptions expressed by means of process algebra with parallel composition, a syntactical characterization that guarantees output persistence will be presented in the next section.

The actual impact of output persistence (in turn coming from an asymmetric treatment of inputs and outputs) in our theory is the existence of a maximal independent refinement (see Section 3.4), i.e. a maximal refinement pre-order that makes it possible to independently refine different contracts of an initial coreography (see [3] for a counter-example showing the necessity of output persistence).

3.2 An Example of Service Language

The service language that we consider in this section, that is aimed to modeling choreographies, can be seen as a distributed version of (a slightly modified version

of) the CCS$^-$ language considered for contracts in the previous section, where we also distinguish between successful and unsuccessful termination. Other differences also come from the fact that we adopt operators which are closer to choreography languages such as abstract WS-BPEL [16]. For instance, here we consider sequential composition $\_;\_$ instead of the prefix operator $\_.\_$, we use a repetition construct $\_^*$ to program unbounded computations instead of recursive definitions, and we assume that the decision to execute an output operation is taken locally, thus all outputs are of the form $\tau;\overline{b}$ (where τ is the typical internal acion of CCS and $\overline{b}$ is an output action on the name b). An important consequence of this form of output operation is that it is not possible to use output actions as guards of branches in a choice; for instance, we cannot write $a + \overline{b}$, but we have to write $a + (\tau;\overline{b})$ where performing the τ action represents the decision to perform the action $\overline{b}$: once the decision is taken we have to perform $\overline{b}$ to reach success. Contracts arising from services with such a syntax are obviously output persistent. Notice that, as long as output finite persistent contracts are derived from services (here finiteness is guaranteed by usage of the Kleene-star repetition operator instead of general recursion), our contract theory is totally independent from the particular service language considered.

Definition 5. (Services) *The syntax of services is*

$$S ::= \mathbf{0} \quad | \quad \mathbf{1} \quad | \quad \tau \quad | \quad a \quad | \quad \tau;\overline{a}_l \quad | \quad S;S \quad | \quad S+S \quad | \quad S|S \quad | \quad S^*$$

In the following we will omit trailing "$\mathbf{1}$" when writing services.

The operational semantics of services, giving rise to a contract in the form of a finite connected LTS with termination transitions for every S, is quite standard; see [5] for a formal definition.

3.3 Composing Services via their Contract

Definition 6. (Systems) *The syntax of systems (contract compositions) is*

$$P \quad ::= \quad [C]_l \quad | \quad P \| P \quad | \quad P \backslash\backslash L$$

where $L \subseteq \{a_l, \overline{a}_l \mid a \in \mathcal{N} \wedge l \in Loc\}$. A system P is well-formed if: (i) every contract subterm $[C]_l$ occurs in P at a different location l and (ii) no output action with destination l is syntactically included inside a contract subterm occurring in P at the same location l, i.e. actions $\overline{a}_l$ cannot occur inside a subterm $[C]_l$ of P. The set of all well-formed systems P is denoted by $\mathcal{P}$. In the following we will just consider well-formed systems and, we will call them just systems.

The operational semantics of systems is defined by the rules in Table 4 plus symmetric rules. We take λ to range over the set of transition labels $\mathcal{L}_{sys} = \{a_l, \overline{a}_l, \tau, \surd \mid a \in \mathcal{N} \wedge l \in Loc\}$.

Example 1. **(Travel Agency Service)** As a running example, we consider a travel agency service which, upon invocation, sends parellel invocations to an

$$\frac{C \xrightarrow{a} C'}{[C]_l \xrightarrow{a_l} [C']_l} \qquad \frac{C \xrightarrow{\lambda} C' \quad \lambda = \overline{a}_{l'}, \tau, \surd}{[C]_l \xrightarrow{\lambda} [C']_l} \qquad \frac{P \xrightarrow{\lambda} P' \quad \lambda \neq \surd}{P \| Q \xrightarrow{\lambda} P' \| Q}$$

$$\frac{P \xrightarrow{a_l} P' \quad Q \xrightarrow{\overline{a}_l} Q'}{P \| Q \xrightarrow{\tau} P' \| Q'} \qquad \frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P \| Q \xrightarrow{\surd} P' \| Q'} \qquad \frac{P \xrightarrow{\lambda} P' \quad \lambda \notin L}{P \backslash L \xrightarrow{\lambda} P' \backslash L}$$

Table 4. Semantic rules for contract compositions (symmetric rules omitted).

airplane reservation service and a hotel reservation service in order to complete the overall organization of a trip. The travel agency service can be defined as:

$$\begin{aligned}[Reservation; (\ \tau; \overline{Reserve}_{AirCompany}; ConfirmFlight \mid \\ \tau; \overline{Reserve}_{Hotel}; ConfirmRoom\ \ ); \\ \tau; \overline{Confirmation}_{Client}]_{TravelAgency}\end{aligned}$$

A possible client for this service can be as follows:

$$[\tau; \overline{Reservation}_{TravelAgency}; Confirmation]_{Client}$$

while the two reservation services could be:

$$[Reserve; \tau; \overline{ConfirmFlight}_{TravelAgency}]_{AirCompany}$$
$$[Reserve; \tau; \overline{ConfirmRoom}_{TravelAgency}]_{Hotel}$$

3.4 Subcontract Relation

Intuitively, a system composed of contracts is correct if all possible computations may guarantee completion; this means that the system is both deadlock and livelock free (there can be an infinite computation, but given any possible prefix of this infinite computation, it must be possible to extend it to reach a successfully completed computation).

Definition 7. (Correct contract composition) *A system P is a correct contract composition, denoted $P\downarrow$, if for every P' such that $P \xrightarrow{\tau}{}^{*} P'$ there exists P'' such that $P' \xrightarrow{\tau}{}^{*} P'' \xrightarrow{\surd}$.*

We are now ready to formalize the notion of pre-order allowing for the refinement of contracts preserving the correctness of contract compositions. We call these class of pre-orders *Independent Subcontracts*. Given a contract $C \in \mathcal{P}_{con}$, we use $oloc(C) \subset Loc$ to denote the set of the locations of the destinations of all the output actions occurring inside C.

Definition 8. (Independent Subcontract pre-order) *A pre-order $\leq$ over $\mathcal{P}_{con}$ is an independent subcontract pre-order if, for any $n \geq 1$, contracts $C_1, \ldots, C_n \in \mathcal{P}_{con}$ and $C'_1, \ldots, C'_n \in \mathcal{P}_{con}$ such that $\forall i.\, C'_i \leq C_i$, and distinguished location names $l_1, \ldots, l_n \in Loc$ such that $\forall i.\, l_i \notin oloc(C_i) \cup oloc(C'_i)$, we have*

$$([C_1]_{l_1} \parallel \ldots \parallel [C_n]_{l_n})\downarrow \quad \Rightarrow \quad ([C'_1]_{l_1} \parallel \ldots \parallel [C'_n]_{l_n})\downarrow$$

We will show that the maximal independent subcontract pre-order can be achieved defining a coarser form of refinement in which, given any system composed of a set of contracts, refinement is applied to one contract only (thus leaving the other unchanged). We call this form of refinement *singular subcontract pre-order*.

Intuitively a pre-order $\leq$ over $\mathcal{P}_{con}$ is a singular subcontract pre-order whenever the correctness of systems is preserved by refining just one of the contracts. More precisely, for any $n \geq 1$, contracts $C_1, \ldots, C_n \in \mathcal{P}_{con}$, $1 \leq i \leq n$, $C'_i \in \mathcal{P}_{con}$ such that $C'_i \leq C_i$, and distinguished location names $l_1, \ldots, l_n \in Loc$ such that $\forall k \neq i.\, l_k \notin oloc(C_k)$ and $l_i \notin oloc(C_i) \cup oloc(C'_i)$, we require

$$([C_1]_{l_1} \parallel \ldots \parallel [C_i]_{l_i} \parallel \ldots \parallel [C_n]_{l_n})\downarrow \quad \Rightarrow \quad ([C_1]_{l_1} \parallel \ldots \parallel [C'_i]_{l_i} \parallel \ldots \parallel [C_n]_{l_n})\downarrow$$

By exploiting commutativity and associativity of parallel composition we can group the contracts which are not being refined and get the following cleaner definition. We let $\mathcal{P}_{conpar}$ denote the set of systems of the form $[C_1]_{l_1} \parallel \ldots \parallel [C_n]_{l_n}$, with $C_i \in \mathcal{P}_{con}$, for all $i \in \{1, \ldots, n\}$.

Definition 9. (Singular subcontract pre-order) *A pre-order $\leq$ over $\mathcal{P}_{con}$ is a singular subcontract pre-order if, for any $C, C' \in \mathcal{P}_{con}$ such that $C' \leq C$, $P \in \mathcal{P}_{conpar}$, $l \in Loc$ such that $l \notin oloc(C) \cup oloc(C') \cup loc(P)$, we have $([C]_l \| P)\downarrow$ implies $([C']_l \| P)\downarrow$*

From the simple structure of their definition we can easily deduce that singular subcontract pre-order have maximum, i.e. there exists a singular subcontract pre-order that includes all the other singular subcontract pre-orders.

Definition 10. (Subcontract relation) *A contract C' is a subcontract of a contract C denoted $C' \preceq C$, if and only if for all $P \in \mathcal{P}_{conpar}$, $l \in Loc$ such that $l \notin oloc(C) \cup oloc(C') \cup loc(P)$, we have $([C]_l \| P)\downarrow$ implies $([C']_l \| P)\downarrow$*

It is trivial to verify that the pre-order $\preceq$ is a singular subcontract pre-order and is the maximum of all the singular subcontract pre-orders.

Theorem 2. *A pre-order $\leq$ is an independent subcontract pre-order if and only if $\leq$ is a singular subcontract pre-order.*

We can, therefore, conclude that there exists a maximal independent subcontract pre-order and it corresponds to "$\preceq$".

Example 2. It is not difficult to see that the parallel composition of the contracts of services $TravelAgency$, $Client$, $AirCompany$ and $Hotel$ defined in the Example 1 is a correct composition according to the Definition 7. It is also interesting

to observe that the travel agency service could invoke sequentially the service without breaking the correctness of the system:

$$[\ Reservation; \tau; \overline{Reserve}_{AirCompany}; ConfirmFlight;$$
$$\tau; \overline{Reserve}_{Hotel}; ConfirmRoom; \tau; \overline{Confirmation}_{Client}\ ]_{TravelAgency}$$

Nevertheless, the contract of this new service is not in general a subcontract of the one of the travel agency service proposed in Example 1 because there exist context in which it cannot be a correctness preserving substitute. Consider, for instance, the two following interacting reservation services:

$$[Reserve; HotelConfirm; \tau; \overline{ConfirmFlight}_{TravelAgency}]_{AirCompany}$$
$$[Reserve; \tau; \overline{ConfirmRoom}_{TravelAgency}; \tau; \overline{HotelConfirm}_{AirCompany}]_{Hotel}$$

In this case, the confirmation of the hotel reservation service is always sent to the travel agency before the confirmation of the airplane company; this is not problematic for the travel agency in the Example 1 that performs the invocation in parallel, while the above sequential invocation deadlocks.

3.5 Subcontract Relation Characterization

The definition of the subcontract relation reported in Definition 10 cannot be directly used to check whether two contracts are in relation due to the universal quantification on all possible locations l and contexts P. In this section we first prove that it is not necessary to range over all possible contexts P in order to check whether a contract C' is in subcontract relation with a contract C, but it is sufficient to consider a *restricted class of relevant contexts* in which all input and output operations are performed on channels on which the contract C can perform outputs or inputs, respectively. Then we present an actual procedure, achieved resorting to the theory of *should-testing* [17], that can be used to prove that two contracts are in subcontract relation.

To characterize the restricted class of relevant contexts we have to introduce a subcontract relation parameterized on the set of inputs and outputs executable by the context. In the following we use $\mathcal{N}_{loc} = \{a_l \mid a \in \mathcal{N}, l \in Loc\}$ to denote the set of located action names and we assume that, given $I \subset \mathcal{N}_{loc}$, $\overline{I} = \{\overline{a}_l \mid a_l \in I\}$.

Definition 11. (Input and Output sets) *Given $C \in \mathcal{P}_{con}$, we define $I(C)$ (resp. $O(C)$) as the subset of $\mathcal{N}$ (resp. $\mathcal{N}_{loc}$) of the potential input (resp. output) actions of C. Formally, we define $I(C)$ as follows ($O(C)$ is defined similarly):*

$$I(\mathbf{0}) = I(\mathbf{1}) = I(X) = \emptyset \qquad I(\tau.C) = I(\overline{a}_l.C) = I(recX.C) = I(C)$$
$$I(a.C) = \{a\} \cup I(C) \qquad I(C+C') = I(C) \cup I(C')$$

Given the system P, we define $I(P)$ (resp. $O(P)$) as the subset of $\mathcal{N}_{loc}$ of the potential input (resp. output) actions of P. Formally, we define $I(P)$ as follows ($O(P)$ is defined similarly):

$$I([C]_l) = \{a_l \mid a \in I(C)\} \qquad I(P \| P') = I(P) \cup I(P') \qquad I(P \backslash L) = I(P) - L$$

We have now to consider slightly more complex contexts than $\mathcal{P}_{conpar}$ systems. We let $\mathcal{P}_{conpres}$ denote the set of systems of the form $([C_1]_{l_1} \| \ldots \| [C_n]_{l_n}) \backslash\backslash L$ with $C_i \in P_{con}$ for all $i \in \{1, \ldots, n\}$ and $L \in \{a_l, \overline{a}_l \mid a \in \mathcal{N} \wedge l \in Loc\}$. Note that, given $P = ([C_1]_{l_1} \| \ldots \| [C_n]_{l_n}) \backslash\backslash I \cup \overline{O} \in \mathcal{P}_{conpres}$, we have $I(P) = (\bigcup_{1 \leq i \leq n} I([C_i]_{l_i})) - I$ and $O(P) = (\bigcup_{1 \leq i \leq n} O([C_i]_{l_i})) - O$. In the following we let $\mathcal{P}_{conpres,I,O}$, with $I, O \subseteq \mathcal{N}_{loc}$, denote the subset of systems of $\mathcal{P}_{conpres}$ such that $I(P) \subseteq I$ and $O(P) \subseteq O$.

Definition 12. (Input-Output Subcontract relation) *A contract C' is a subcontract of a contract C with respect to a set of input located names $I \subseteq \mathcal{N}_{loc}$ and output located names $O \subseteq \mathcal{N}_{loc}$, denoted $C' \preceq_{I,O} C$, if and only if for all $P \in \mathcal{P}_{conpres,I,O}$, $l \in Loc$ such that $l \notin oloc(C) \cup oloc(C') \cup loc(P)$, we have $([C]_l \| P)\downarrow$ implies $([C']_l \| P)\downarrow$*

It is not difficult to see, looking at Definition 10, that $\preceq = \preceq_{\mathcal{N}_{loc},\mathcal{N}_{loc}}$. Moreover, notice that, given $\preceq_{I',O'}$, and larger sets I and O (i.e. $I' \subseteq I$ and $O' \subseteq O$) we obtain a smaller pre-order $\preceq_{I,O}$ (i.e. $\preceq_{I,O} \subseteq \preceq_{I',O'}$). This follows from the fact that extending the sets of input and output actions means considering a larger set of discriminating contexts.

The following Proposition shows conditions on the input and output sets under which also the opposite direction holds; more precisely, the set of potential inputs and outputs of the other contracts in the system (as long as it includes those needed to interact with the contract) is an information that does not influence the subcontract relation.

Proposition 2. *Let $C \in \mathcal{P}_{con}$ be contracts, $I, I' \subseteq \mathcal{N}_{loc}$ be two sets of located input names such that $O(C) \subseteq I, I'$ and $O, O' \subseteq \mathcal{N}_{loc}$ be two sets of located output names such that for every $l \in Loc$ we have $I([C]_l) \subseteq O, O'$. We have that for every contract $C' \in \mathcal{P}_{con}$,*

$$C' \preceq_{I,O} C \iff C' \preceq_{I',O} C \iff C' \preceq_{I,O'} C$$

It is interesting to note that the above results depend on the systems being output persistent. Consider, e.g., the trivially correct system $[a]_{l_1} \| [\tau.\overline{a}_{l_1}]_{l_2}$. We have that the contract a could be replaced by $a + c.\mathbf{0}$ as well as the contract $\tau.\overline{a}_{l_1}$ that could be replaced by $\tau.\overline{a}_{l_1} + c.\mathbf{0}$; this because it is easy to see that

$$a + c.\mathbf{0} \preceq_{\emptyset,\{a_l,c_l \mid l \in Loc\}} a \quad \text{and} \quad \tau.\overline{a}_{l_1} + c.\mathbf{0} \preceq_{\{a_{l_1}\},\{c_l \mid l \in Loc\}} \tau.\overline{a}_{l_1}$$

thus, as a consequence of the last two propositions and the fact that $\preceq = \preceq_{\mathcal{N}_{loc},\mathcal{N}_{loc}}$, we have that

$$a + c.\mathbf{0} \preceq a \quad \text{and} \quad \tau.\overline{a}_{l_1} + c.\mathbf{0} \preceq \tau.\overline{a}_{l_1}$$

But these two examples of subcontracts are not correct if we can write output operations without a previous internal τ action. In fact, the two correct systems $[a]_{l_1} \| [\overline{a}_{l_1} + \overline{c}_{l_1}]_{l_2}$ and $[a + \overline{c}_{l_2}]_{l_1} \| [\overline{a}_{l_1}]_{l_2}$, are no longer correct if we replace the contracts a and $\overline{a}_{l_1}$ with their abovely discussed subcontracts.

The above example show that a subcontract may contain additional inputs (see the additional input on channel c). This property is formalized by the following Lemma that is a direct consequence of the fact that $C' \preceq_{\mathcal{N}_{loc},\bigcup_{l\in Loc} I([C]_l)} C$ if and only if $C' \preceq C$ as stated by Proposition 2. In the Lemma (and in the following) we use the abuse of notation "$C\backslash\backslash M$" to stand for the contract "$C\{\mathbf{0}/a | a \in M\}$" achieved replacing all input actions in the set $M \in \mathcal{N}$ with the failed process $\mathbf{0}$.

Lemma 1. *Let $C, C' \in \mathcal{P}_{con}$ be contracts. We have*

$$C'\backslash\backslash(I(C') - I(C)) \preceq C \quad \Leftrightarrow \quad C' \preceq C$$

The remainder of this subsection is devoted to the definition of an actual procedure for proving that two contracts are in subcontract relation. This is achieved resorting to the theory of *should-testing* [17]. We denote with $\preceq_{test}$ the *should-testing* pre-order defined in [17] where we consider the set of actions used by terms as being $\Lambda = \mathcal{L}_{sys} \cup \{a, \overline{a} \mid a \in \mathcal{N}\}$ (i.e. we consider located and unlocated input and output actions and $\surd$ is included in the set of actions of terms under testing as any other action). We denote here with $\surd'$ the special action for the success of the test (denoted by $\surd$ in [17]). Should testing is a variant of must testing which ensures correctness under a fairness assumption, similarly as for our correct contract compositions. Given two processes P and P', $P' \preceq_{test} P$ iff for every test t, P **shd** t implies P' **shd** t, where Q **shd** t iff

$$\forall w \in \Lambda^*, Q'. \qquad Q\|_{\Lambda-\{\tau\}} t \xrightarrow{w} Q' \qquad \Rightarrow \qquad \exists v \in \Lambda^*, Q'' : Q' \xrightarrow{v} Q'' \xrightarrow{\surd'}$$

where $\|_S$ is the CSP parallel operator: in $R\|_S R'$ transitions of R and R' with the same label $\lambda \in S$ must synchronize and yield a λ transition.

In order to resort to the theory of [17], we just have to consider "normal form" terms $\mathcal{NF}(C)$, where $\mathcal{NF}(C)$ is obtained from contract terms C by replacing $\surd.\mathbf{0}$ for $\mathbf{1}$ and by using the notation for prefix and recursion used in [17].

Theorem 3. *Let $C, C' \in \mathcal{P}_{con}$ be two contracts. We have*

$$\mathcal{NF}(C'\backslash\backslash I(C') - I(C)) \preceq_{test} \mathcal{NF}(C) \quad \Rightarrow \quad C' \preceq C$$

Note that the opposite implication

$$C' \preceq C \;\Rightarrow\; \mathcal{NF}(C'\backslash\backslash I(C') - I(C)) \preceq_{test} \mathcal{NF}(C)$$

does not hold in general. For example if we take contracts $C = a + a.c$ and $C' = b + b.c$ we have that $C' \preceq C$ (and $C \preceq C'$) (there is no location l and context P such that $([C]_l \| P)\downarrow$ or $([C']_l \| P)\downarrow$), but obviously $\mathcal{NF}(C'\backslash\backslash\{b\}) \preceq_{test} \mathcal{NF}(C)$ (and $\mathcal{NF}(C\backslash\backslash\{a\}) \preceq_{test} \mathcal{NF}(C')$) does not hold.

In [17] it is proved that, for finite transition systems like our contracts, *should-testing* preorder is decidable and an actual verification algorithm is presented. This algorithm, in the light of our Theorem 3, represents a sound approach to prove also our subcontract relation.

Example 3. We complete the analysis of our running example, showing a subcontract of the original travel agency service. For instance, the following alternative travel agency gives rise to a subcontract of the one proposed in the Example 1:

$$[Reservation; (\ \tau; \overline{Reserve}_{AirCompany}; ConfirmFlight \mid$$
$$\tau; \overline{Reserve}_{Hotel}; (ConfirmRoom + CancelRoom)\ \ );$$
$$\tau; \overline{Confirmation}_{Client}]_{TravelAgency}$$

as it simply differs for an additional input on the $CancelRoom$ channel, modelling the possibility for the hotel reservation to fail.

4 Conclusion

We have reported an overview of mechanisms that paves the way for service publication, discovery and composition based on the notion of *service contract*, that is an abstract description of the message-passing behaviour of the service. These mechanisms have been developed also in the light of applications to the core calculi described in Chapter 2-1. Also related to service publication, discovery and composition are 2-4 and 3-1. In Chapter 2-4 the authors present a framework for designing and composing services in a "call-by-contract" fashion, i.e. according to their behaviour and show how to correctly plan service compositions in some relevant classes of services and behavioural properties. They propose a core functional calculus for services and a type and effect system over-approximating the actual run-time behaviour of services. A further static analysis step finds the plan that drive the selection of services matching the behavioural requirements on demand. In Chapter 3-1 the authors overview the CC-pi calculus, a model for specifying QoS negotiations in service composition that also allows to study mechanisms for resource allocation and for joining different QoS parameters.

Concerning related work, Igarashi and Kobayashi's work [14] on generic type systems is the first instance of the processes-as-types approach. The work [1] is mostly inspired by [14], with a few important differences. In particular, [1] considers an asynchronous version of the pi-calculus, and types account for a *tag*-wise, rather than *channel*-wise, view of the behaviour of processes. On one hand, this simplification leads to some loss of information, which prevents one from capturing certain liveness properties such as race- and deadlock-freedom. On the other hand, it allows one to make the connection between different kinds of behavior (open/closed) and different type models (CCS/BPP) direct and explicit. As an example, in the case of BPP [1] spells out reasonably simple conditions under which the type analysis is "precise" (Γ-uniform receptiveness). Also, this approach naturally carries over to the Join calculus, by moving to Petri nets types. The paradigm type-as-abstraction is also the subject of [2]. There, sufficient conditions are given under which certain liveness and safety properties, expressed in a simple spatial logic, can be transferred from types to the inhabiting processes.

Another work strongly related to our concept of type-as-abstraction is [15], where the generic type system of Igarashi and Kobayashi is extended in order to

guarantee certain safety properties in a resource-access scenario. A pi-calculus enriched with resources and related access primitives is introduced; resources are decorated with access policies formulated as regular languages. CCS types are used to check the behaviour of processes against those policies. The main result ensures that no well-typed process violates at runtime the prescribed policies.

As far as contract-based service composition is concerned, it is important to say that, even if we have characterized our notion of compliance resorting to the theory of testing, there are some relevant differences between our form of testing and the traditional one proposed by De Nicola-Hennessy [8]. The most relevant difference is that, besides requiring the success of the test, we impose also that the tested process should successfully complete its execution. This further requirement has important consequences; for instance, we do not distinguish between the always unsuccessful process $\mathbf{0}$ and other processes, such as $a.\mathbf{1} + a.b.\mathbf{1}$, for which there are no guarantees of successful completion in any possible context. Another relevant difference is in the treatment of divergence: we do not follow the traditional catastrophic approach, but the fair approach introduced by the theory of should-testing of Rensink-Vogler [17]. In fact, we do not impose that all computations must succeed, but that all computations can always be extended in order to reach success.

Contracts have been investigated also by Fournet et al. [11] and Carpineti et al. [6]. In [11] contracts are CCS-like processes; a generic process P is defined as compliant to a contract C if, for every tuple of names $\tilde{a}$ and process Q, whenever $(\nu\tilde{a})(C|Q)$ is stuck-free then also $(\nu\tilde{a})(P|Q)$ is. Our notion of contract refinement differs from stuck-free conformance mainly because we consider a different notion of stuckness. In [11] a process state is stuck (on a tuple of channel names $\tilde{a}$) if it has no internal moves (but it can execute at least one action on one of the channels in $\tilde{a}$). In our approach, an end-state different from successful termination is stuck (independently of any tuple $\tilde{a}$). Thus, we distinguish between internal deadlock and successful completion while this is not the case in [11]. Another difference follows from the exploitation of the restriction $(\nu\tilde{a})$; this is used in [11] to explicitly indicate the local channels of communication used between the contract C and the process Q. In our context we can make a stronger *closed-world* assumption (corresponding to a restriction on all channel names) because service contracts do not describe the entire behaviour of a service, but the flow of execution of its operations inside one session of communication.

The closed-world assumption is considered also in [6] where, as in our case, a service oriented scenario is considered. In particular, in [6] a theory of contracts is defined for investigating the compatibility between one client and one service. Our paper consider multi-party composition where several services are composed in a peer-to-peer manner. Moreover, we impose service substitutability as a mandatory property for our notion of refinement; this does not hold in [6] where it is not in general possible to substitute a service exposing one contract with another one exposing a subcontract. Another, related, significant difference is that contracts in [6] comprise also mixed choices.

References

1. L. Acciai and M. Boreale. Type abstractions of name-passing processes. In Proc. of IPM International Symposium on Fundamentals of Software Engineering (FSEN), volume 4767 of *LNCS*, pages 302-317, 2007.
2. L. Acciai and M. Boreale. Spatial and Behavioral Types in the Pi-Calculus. In Proc. CONCUR 2008 - Concurrency Theory, 19th International Conference, volume 5201 of *LNCS*, pages 372-386, 2008. Full version to appear in *Information and Computation*.
3. Bravetti, M., Zavattaro, G.: A Foundational Theory of Contracts for Multi-party Service Composition, *Fundamenta Informaticae* 89(4):451–478, IOS Press, 2008.
4. M. Bravetti and G. Zavattaro. Contract-Based Discovery and Composition of Web Services. In Proc. of Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM 2009), Bertinoro, Italy, volume 5569 of *LNCS*, pages 261-295, 2009.
5. M. Bravetti and G. Zavattaro. Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In Proc. of the 6th International Symposium on Software Composition (SC'07), LNCS 4829:34-50, Springer, Braga (Portugal), March 2007. Full version available at: `http://www.cs.unibo.it/~bravetti/html/techreports.html`
6. S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A Formal Account of Contracts for Web Services. In *WS-FM'06*, *LNCS*, 4184:148–162, 2006.
7. S. Christensen, Y. Hirshfeld and F. Moller. Bisimulation equivalence is decidable for basic parallel processes. In *Proc. of CONCUR*, *LNCS*, 715:143–157, 1993.
8. R. De Nicola and M. Hennessy, Testing Equivalences for Processes. *Theoretical Computer Science*, volume 34: 83–133, 1984.
9. J. Esparza. More Infinite Results. *Current trends in Theoretical Computer Science: entering the 21st century*, pp.480-503, 2001.
10. C. Fournet and G. Gouthier. The Reflexive Chemical Abstract Machine and the Join Calculus. In *Proc. of POPL*, ACM press, pp. 372-385, 1996.
11. C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof. Stuck-Free Conformance. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 242–254, 2004.
12. R. Milner, *"A complete axiomatization for observational congruence of finite-state behaviours"*, in Information and Computation 81:227-247, 1989
13. Y. Hirshfeld and F. Moller. Decidability Results in Automata and Process theory. *LNCS*, 1043:102–148, 1996.
14. A. Igarashi and N. Kobayashi. A Generic Type System for the Pi-Calculus. In *Proc. of POPL*, ACM press, pp.128-141, 2001. Full version appeared in *Theoretical Computer Science*, 311(1-3):121–163, 2004.
15. N. Kobayashi, K. Suenaga and L. Wischik. Resource Usage Analysis for the Pi-Calculus. In *Proc. of VMCAI*, volume 3855 of *LNCS*, pages 298–312, 2006.
16. OASIS. *Web Services Business Process Execution Language Version 2.0*.
17. A. Rensink and W. Vogler. Fair testing. In Information and Computation 205(2):125-198, 2007.
18. D. Sangiorgi. The name discipline of uniform receptiveness. In *Proc. of ICALP*, 1997. *Theoretical Computer Science*, 221(1-2):457–493, 1999.
19. C. Stirling. Modal Logics for Communicating Systems. *Theoretical Computer Science*, 49(2-3):311–347, 1987.