Process calculi as a tool for studying coordination, contracts and session types

Process Calculi as a Tool for Studying Coordination, Contracts and Session Types*

Mario Bravetti Gianluigi Zavattaro

Department of Computer Science and Engineering & Focus Team, INRIA
University of Bologna, Italy

Abstract

We recall techniques, mainly based on the theory of process calculi, that we used to prove results in twenty years of research, spanning across the old and the new millennium, on the expressiveness of coordination languages and on behavioural contracts for Service-Oriented Computing. Then, we show how such techniques recently contributed to the clarification of aspects that were unclear about session types, in particular, asynchronous session subtyping that was considered decidable since 2009, while it was proved to be undecidable in 2017.

Keywords: Coordination Primitives, Behavioural Contracts, Session Types

1. Introduction

Shared dataspaces and the so-called generative communication paradigm [1] attracted a lot of attention since the initial years of research about foundations of coordination models and languages. Linda [2], probably the most popular language based on this coordination model, is based on the idea that concurrent processes interact via a shared dataspace, the so-called Tuple Space (TS for short), where the information needed to coordinate the activities are introduced and retrieved. After its insertion in the TS, a datum becomes equally accessible to all processes, but it is bound to none. In this way, the interaction among concurrent processes is decoupled in space and time, principles useful in the development of modular and scalable concurrent and distributed systems.

Concerning foundational studies on Linda-like coordination languages, it appeared immediately clear that techniques borrowed from the tradition of concurrency theory could be naturally applied. At the first two editions of the Coordination conference, two process calculi based on Linda were proposed by De Nicola and Pugliese [3] and by Busi, Gorrieri and Zavattaro [4]. In particular, the latter started a line of research on the expressiveness of Linda-like

*Research partly supported by the H2020-MSCA-RISE project ID 778233 "Behavioural Application Program Interfaces (BEHAPI)".

coordination primitives that exploited, besides process calculi, also Petri nets. For instance, Petri nets were used in [5], to prove that a Linda process calculus with input, output and test-for-absence is *not* Turing complete if the semantics for output is *unordered*, i.e., there is an unpredictable delay between the execution of an output and the actual availability of the emitted datum in the TS. It is interesting to recall that, on the other hand, the same calculus is Turing complete if an *ordered* semantics is considered, i.e. an emitted datum is immediately available in the TS after the corresponding output is executed. Turing completeness was proved by showing an encoding of a Turing powerful formalism, namely Random Access Machines (RAMs), which is a computational model based on registers that can be incremented, decremented and tested for emptiness.

The success of the Linda coordination model was witnessed by the development, at the end of the 90s, of Linda-based middlewares from main ICT vendors like IBM and Sun Microsystem, which proposed T-Spaces and JavaSpaces, respectively. The basic Linda coordination model was extended with primitives for event notification, time-out based data expiration, and transactions. The techniques for evaluating the expressive power of Linda languages had to become more sophisticated to cope with these additional primitives. In particular, Petri nets with transfer and reset arcs [6] were adopted to cope also with the new coordination mechanisms for event notification [7] and for temporary data [8].

During the initial years of the new millennium, Service-Oriented Computing (SOC) emerged as an alternative model for developing communication-based distributed systems. In particular, the large diffusion of Web Services called for the development of new languages and techniques for service composition (see e.g. [9]). In SOC, the coordination model significantly changes w.r.t. Linda: services reciprocally exchange messages sending them directly to the expected receiver, instead of exploiting a shared dataspace.

The idea, at the basis of SOC, is to conceive an ecosystem of services that expose operations that can be combined to realize new applications. To support this idea, it is necessary for the services to be equipped with an interface that, besides describing the offered operations and the format of the exchanged messages, defines the conversation protocols, i.e., the expected flow of invocations of the operations. These interfaces, in particular the specification of the conversation protocol, are also called *behavioural contracts*.

Process calculi contributed to the development of theories for behavioural contracts. This line of research was initiated by Carpineti, Castagna, Laneve and Padovani [10], for the case of client-server composition, and by Bravetti and Zavattaro [11], for multiparty service compositions. The latter is particularly significant for the so-called service choreographies, i.e., systems in which there exists no central orchestrator, responsible for invoking all the other services in the system, because services reciprocally interact. Behavioural contract theories focused mainly on the investigation of appropriate notions of correctness for service compositions (i.e., define when a system based on services is free from communication errors) and on the characterization of notions of compatibility between services and behavioural contracts (i.e., define when a service conforms

to a given behavioural contract).

For multiparty composition, a fairness based notion of correctness, called *compliance*, was proposed for the first time in [11]: a system is correct if, whatever state can be reached, there exists a continuation of the computation that yields a final state in which *all* services have successfully completed. Given the notion of compliance, it is possible to define also a natural notion of *refinement* for behavioural contracts: a refinement is a relation among contracts such that, given a set of compliant contracts $C_1, \cdots, C_n$, each contract C_i can be *independently* replaced by any of its possible refinements C'_i, and the overall system obtained by composition of $C'_1, \cdots, C'_n$ is still compliant. Contract refinement can then be used to check whether a service conforms with a behavioural contract: it is sufficient to verify if the communication behaviour of the service refines the behavioural contract. This, in fact, implies that such service can be safely used wherever a service is expected with the behaviour specified by the contract.

A negative result in the theory of behavioural contracts is that, in general, the union of two refinement relations is not guaranteed to be itself a refinement. This implies the impossibility to define a maximal notion of refinement. For this reason, most of the effort in the line of research on behavioural contracts initiated in [11] has been dedicated to the identification of interesting subclasses of contracts for which the maximal refinement exists. Such classes are: contracts with *output persistence* [11] (i.e. output actions cannot be avoided when a state is entered in which they are ready to be executed), contract refinement preserving *strong compliance* [12] (i.e. as soon as an output is ready to be executed, a receiver is guaranteed to be ready to receive it), and *asynchronously communicating* contracts [13] (i.e. communication is mediated by fifo buffers). In the first two of these three cases, it has been also possible to provide a sound algorithmic characterization of the corresponding maximal refinements.

To the best of our knowledge, characterizing algorithmically the maximal contract refinement in case of asynchronous communication is still an open problem. The main source of difficulty derives from the fact that, due to the presence of unbounded communication buffers, systems of asynchronously communicating contracts are infinite-state, even if contracts are finite-state. In the light of this difficulty, we tried to take inspiration from work on session types, where asynchronous communication has been investigated since the seminal work by Honda, Yoshida and Carbone [14] (recipient of the most influential POPL'08 paper award). Session types can be seen as a simplification of contracts obtained by imposing some limitations: there are only two possible choices, *internal* choice among distinct outputs and *external* choice among distinct inputs.

The counterpart of contract refinement in the context of session types is *subtyping* [15]. If we consider asynchronous communication, both contract refinement and session subtyping can admit a refinement/subtype to perform the communication actions in a different order. For instance, given a contract/type that performs an input followed by an output, a refinement/subtype can *anticipate* the output before the input, because such output can be *buffered* and actually received afterwards. Asynchronous session subtyping was already studied

by Mostrous, Yoshida, and Honda in [16], where also an algorithm for checking subtyping was presented. Upon studying this algorithm we noticed an error in its proof of termination: if, while checking subtyping, the buffer grows unboundedly, the proposed procedure does not terminate. Subsequently, Bravetti, Carbone and Zavattaro [17] and Lange and Yoshida [18] independently proved that asynchronous session subtyping is actually undecidable. Our experience in the modeling of Turing complete formalism (see the above discussion about encoding RAMs in the Linda process calculus) helped in finding an appropriate Turing powerful model to be encoded in terms of asynchronous session subtyping. In particular, we were able to present a translation from a Queue Machine M (a computational model similar to pushdown automata, but with a queue instead of a stack) to a pair of session types that are in asynchronous subtyping relation if and only if M does not terminate. Then, undecidability of asynchronous session subtyping directly follows from the undecidability of the halting problem for Queue Machines.

These negative results opened the problem of identifying significant classes of session types for which asynchronous subtyping can be decided. Currently, the most interesting fragments have been identified in [17, 18] and [19]. In the former, an algorithm is presented for the case in which one of the two types is completely deterministic, i.e. all choices –both internal and external– have one branch only. In the latter, we have considered single-out (and single-in) session types, meaning that in both types to be checked all internal choices (resp. external choices) have one branch only. In the design of our algorithm we have been inspired by our expertise in the analysis of the expressiveness of Linda process calculi. In particular, the analysis techniques in Petri nets with transfer and reset arcs (our tools to prove decidability results) are based on the notion of well quasi ordering (wqo): while generating an infinite sequence of elements, it is guaranteed to eventually generate an element that is greater –with respect to the wqo– of an already generated element. Similarly to the procedure in [16], our algorithm checks a sequence of judgements, but differently from [16], termination is guaranteed because there exists a wqo on judgements, and the algorithm can terminate when a judgement greater than an already checked one is considered.

1.1. Structure of the paper

The remainder of this paper is divided in three parts.

In Section 2 we recall the techniques used to investigate the expressive power of the Linda coordination model: the formalization of Linda-like process calculi, the proof of Turing completeness of the calculi by reduction from Random Access Machines (RAMs), and the identification of fragments that, on the contrary, are not Turing complete. The non Turing completeness results are obtained by proving the decidability of reachability properties; these decidability results are proved by resorting to Well Structured Transition Systems (WSTS).

In Section 3 we recall the main results concerning behavioural contracts: their formalization as process calculi, the notion of correct contract composition, the definition of contract refinement, and its algorithmic characterization.

We consider both synchronous and asynchronous communication, excluding the algorithmic characterization which is available only for the synchronous case.

Finally, in Section 4 we move to the analysis of asynchronous communication in the simplified context of session types. In [16] it is stated that subtyping, which is the counterpart of contract refinement, is decidable; recently it has been proved that, on the contrary, it is undecidable. Here, we report the undecidability proof in [17]. Finally, we present the fragment of single-out session types, for which subtyping turns out to be decidable [19]. The techniques for these (un)decidability results can be seen as improvements of those developed for Linda process calculi: reduction from more adequate Turing complete computational models (queue machines instead of RAMs), and application of the notion of wqo (at the basis of WSTS) to (a variant of) the subtyping algorithm prposed in [16] that, differently from what stated therein, does not terminate in some specific cases.

2. A Linda process calculus

In this section we present the syntax and the semantics of a calculus for processes that interacts following the Linda coordination model. The calculus is original, even if inspired by calculi such as those presented at the first editions of the Coordination conference [3, 4]. The main novelty deals with the management of names which is inspired by the π-calculus [20]: processes can generated new names and pass them to other processes, simply by including such new names inside tuples that are inserted in the shared dataspace.

Definition 2.1 (Processes). *Let Name, ranged over by a, b, ..., be a denumerable set of names. With Message we denote the set of tuples of names, denoted with $\tilde{a}$, $\tilde{b}$, ... The class of processes is described by the following grammar:*

$$\begin{array}{lcl} \eta & ::= & a \mid (a) \\ P & ::= & \mathbf{1} \mid out(\tilde{a}) \mid in(\tilde{\eta})P \mid (\nu a)P \mid \\ & & P+P \mid P|P \mid P;P \mid P^* \end{array}$$

Communication is based on message emission and consumption. The operation $out(\tilde{a})$ emits a message consisting of the tuple of names $\tilde{a}$. A consumption operation $in(\tilde{\eta})P$ is based on a pattern $\tilde{\eta}$, which is a sequence of free and bound names: the bound names are those in parentheses (i.e., a is free while (a) is bound). The scope of bound names is the process P. A pattern $\langle \eta_1, \cdots, \eta_n \rangle$ matches message $\langle a_1, \cdots, a_n \rangle$ if η_i free implies $\eta_i = a_i$. In this case, we write $\tilde{\eta}[{}^{a_j}/_{b_j}]_{j \in J} = \tilde{a}$, with J the set of the indexes of bound names in $\tilde{\eta}$, and b_j such that $\eta_j = (b_j)$ for every $j \in J$. Upon consumption of the tuple $\tilde{a}$, the free occurrences of b_j in P are replaced by the corresponding a_j names (substitution denoted with $P[{}^{a_j}/_{b_j}]_{j \in J}$).

The term $\mathbf{1}$ denotes the process ready to terminate. The operator $(\nu a)P$ binds name a in P; intuitively, the notation (νa) can be seen as the generation

of a fresh name that is initially known by P only. The sum construct $+$ is used to make choice between the summands, parallel composition $|$ is used to run parallel programs, while ; executes first the left operand and then the right one. The term P^* is used to denote a process that can repeat the execution of P an arbitrary amount of times (possibly zero times). As usual, we admit α conversion of bound names, and we use $fn(P)$ to denote the set of free names occurring in P.

Example 2.1. *As an example of a simple system modeled with our Linda calculus, consider a sensor and a gateway that work in the following way. The sensor periodically wakes up and starts a measuring session; in such a session it measures temperature and the humidity, and communicate them to the gateway. Having no memory, the sensor communicates the measured temperature and the humidity in two separate messages; such messages are inserted in a common dataspace. The gateway reads the measured temperatures and humidities from the dataspace, and combines them in oder to produce pairs of measures taken from the sensor in the same session.*

The main problem in this simple system is how to guarantee that the gateway correctly combines temperature and humidity measured in the same session. We assume this problem is solved by enriching the temperature and humidity messages with a session identifier. Following this approach, the behaviour of the sensor can be modeled as follows:

$$Sensor \quad = \quad \big(\ (\nu k)(\ out(T,t,k); out(H,h,k)\ )\ \big)^*$$

where k is the session identifier which is freshly generated at the beginning of each iteration; T (resp. H) is a constant indicating that the generated tuple contains a measured temperature (resp. humidity); and t and h are the measured temperature and humidity.[1]

The gateway must combine measures that share the same session identifier. Following this approach, the behaviour of the gateway can be modeled as follows:

$$Gateway \quad = \quad \big(\ in(T,(x),(s))\big(\ in(H,(y),s)(\ out(x,y)\ )\ \big)\ \big)^*$$

Notice that s is received when the first input of a temperature is executed, and then it is used in the second operation to ensure that the corresponding humidity is consumed from the dataspace. After consumption, a new message that combines the paired temperature/humidity is placed in the dataspace.

The calculus assumes that processes are enriched with a shared dataspace where tuples are stored and consumed.

Definition 2.2 (Systems). *A system is a pair $\langle P, \mathcal{S}\rangle$ where P is a process and $\mathcal{S}$ is a multiset over Message.*

[1] In the real system, temperature and humidity are expected to be measured at each iteration. In our simplified abstract model, we represent all these possibly different measures with the same names t and h.

We are now ready to define the operational semantics of systems; we first define a labeled transition system on processes which indicates the possible input and output actions, and then we define a transition relation on systems which indicates the effect of process actions execution. Due to the presence of the sequential composition operator $P;Q$, we need to explicitly deal with process completion, because Q can start executing only when P completes. Process completion is denoted by a dedicated transition $P \xrightarrow{\surd} P'$. We need an auxiliary process $\mathbf{0}$ that we use as the target of the termination transition from the process $\mathbf{1}$, i.e., $\mathbf{1} \xrightarrow{\surd} \mathbf{0}$.

Definition 2.3 (Process semantics). *The semantics of processes is defined by a labeled transition system on processes using three kinds of labels α: the actions $\tilde{a}^{\tilde{b}}$ indicating the emission of the message $\tilde{a}$ extruding the bound names $\tilde{b}$, the action $\tilde{\eta}^{[a_j/b_j]_{j\in J}}$ indicating the consumption of a message ($[a_j/b_j]_{j\in J}$ is the substitution concerning the bound names in $\tilde{\eta}$), and the label $\surd$ for termination transitions. The transition system is the least one satisfying the axioms and rules in Table 1 (plus the symmetric rules of* PAR *and* SUM*), where we use λ to denote labels different from $\surd$, $n(\alpha)$ to denote the names occurring in the label α, and $bn(\alpha)$ to denote the bound names in the label α. Namely, $bn(\tilde{a}^{\tilde{b}})$ contains the names in $\tilde{b}$ while $bn(\alpha)$ is empty in all other cases.*

Notice that we use two similar notations: $\tilde{\eta}^{[a_j/b_j]_{j\in J}}$ and $P[a_j/b_j]$. In the first case, $[a_j/b_j]_{j\in J}$ is to be interpreted as a tuple of pairs of names $[a_j/b_j]$, with b_j occurring bound in $\tilde{\eta}$ and a_j denoting the corresponding name expected to be retrieved from the dataspace. In the second case, $P[a_j/b_j]$ is the term obtained by applying the substitutions $[a_j/b_j]$ to P.

The OUT axiom allows for the execution of an output operation; notice the initial label $\tilde{a}^{\varepsilon}$ where ε (the empty sequence) indicates that all the names in $\tilde{a}$ are free. The IN axiom, on the other hand, deals with input operations; the substitution $[a_j/b_j]_{j\in J}$ is guessed, following the so-called early approach, and indicates the names a_j that are expected to be present in the consumed message in the positions of the template corresponding with the bound names (b_j). As discussed above, the END axiom deals with the termination transition for the process $\mathbf{1}$. Rule ENDP states that a parallel process terminates only when both its subprocesses terminates. PAR and SUM are the usual rules for the parallel and choice operators, where the additional condition in PAR checks that if the action α contains bound names, these do not occur free in Q (hence, they cannot be captured). An action can traverse (νa) only if the involved names are different from a (see RES), excluding the case of output actions including a, but in this case the extruded name is added to the tuple of bound names in the label (see OPEN). Rule SEQ1 allows the first process in a sequence to proceed[2], SEQ2 allows

[2]The side condition in SEQ1 avoids name captures between new names in the first process and free names in the second process in a sequential composition.

$$\texttt{OUT}: \quad out(\tilde{a}) \xrightarrow{\tilde{a}^{\varepsilon}} \mathbf{1} \qquad\qquad \texttt{IN}: \quad in(\tilde{\eta})P \xrightarrow{\tilde{\eta}^{[a_j/b_j]_{j\in J}}} P[a_j/b_j]$$

$$\texttt{END}: \quad \mathbf{1} \xrightarrow{\surd} \mathbf{0} \qquad\qquad \texttt{ENDP}: \quad \frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P|Q \xrightarrow{\surd} P'|Q'}$$

$$\texttt{PAR}: \quad \frac{P \xrightarrow{\lambda} P' \quad bn(\lambda) \cap fn(Q) = \emptyset}{P|Q \xrightarrow{\lambda} P'|Q} \qquad\qquad \texttt{SUM}: \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\texttt{RES}: \quad \frac{P \xrightarrow{\alpha} P' \quad a \notin n(\alpha)}{(\nu a)P \xrightarrow{\alpha} (\nu a)P'} \qquad\qquad \texttt{OPEN}: \quad \frac{P \xrightarrow{\tilde{a}^{\tilde{b}}} P' \quad a \in \tilde{a}}{(\nu a)P \xrightarrow{\tilde{a}^{\tilde{b}\cdot a}} P'}$$

$$\texttt{SEQ1}: \quad \frac{P \xrightarrow{\lambda} P' \quad bn(\lambda) \cap fn(Q) = \emptyset}{P;Q \xrightarrow{\lambda} P';Q} \qquad\qquad \texttt{SEQ2}: \quad \frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\alpha} Q'}{P;Q \xrightarrow{\alpha} Q'}$$

$$\texttt{ITER}: \quad \frac{P \xrightarrow{\lambda} P'}{P^* \xrightarrow{\lambda} P';P^*} \qquad\qquad \texttt{ENDI}: \quad P^* \xrightarrow{\surd} \mathbf{0}$$

Table 1: The transition system for processes ($\lambda \neq \surd$, symmetric rules of PAR and SUM omitted).

the second one to start assuming the first one has completed. The last two rules deal with iterations: rule ITER starts one iteration while axiom ENDI specifies the possibility for iterations to conclude, simply by performing a termination transition. This rule is necessary, for instance, to allow a process $P^*;Q$ to activate Q after an arbitrary amount of executions of process P. Notice that the termination transition of P^*, labeled with $\surd$, is always provided by axiom ENDI, so it is not necessary for rule ITER to deal with such transitions.

We can now complete the definition of the operational semantics.

Definition 2.4 (System semantics). *The semantics of systems is defined by the minimal transition system satisfying the rules in Table 2.*

The transition system $\rightarrow$ simply allow processes to consume and introduce messages to/from the dataspace; the third rule, on the other hand, allows terminating processes to perform their final $\surd$ transition. In case of output action extruding names, it is necessary to check that the bound names are actually fresh in the dataspace. In the following, we use $\rightarrow^*$ to denote the reflexive and transitive closure of $\rightarrow$.

Example 2.2. *In Example 2.1 we have introduced two processes, Sensor and Gateway. We now consider the following system $\langle Sensor|Gateway, \emptyset\rangle$, representing an initial state with an instance of Sensor in parallel with an instance of Gateway, with no message in the dataspace. According to our semantics, we*

$$\frac{P \xrightarrow{\tilde{\eta}^{[a_j/b_j]_{j\in J}}} P' \quad \tilde{\eta}[a_j/b_j]_{j\in J} = \tilde{a}}{\langle P, \mathcal{S} \uplus \tilde{a}\rangle \to \langle P', \mathcal{S}\rangle} \qquad \frac{P \xrightarrow{\tilde{a}^{b_1\cdots b_n}} P' \quad \forall i \in 1\ldots n. b_i \notin \mathcal{S}}{\langle P, \mathcal{S}\rangle \to \langle P', \mathcal{S} \uplus \tilde{a}\rangle}$$

$$\frac{P \xrightarrow{\surd} P'}{\langle P, \mathcal{S}\rangle \to \langle P', \mathcal{S}\rangle}$$

Table 2: The reduction relation for systems (brackets in singletons are omitted).

have that the following sequence of transitions are possible:

$$\begin{array}{l}
\langle\ \mathit{Sensor} \mid \mathit{Gateway}\ ,\ \emptyset\ \rangle\ \to \\
\langle\ \mathit{out}(H,h,k); \mathit{Sensor} \mid \mathit{Gateway}\ ,\ \{\langle T,t,k\rangle\}\ \rangle\ \to \\
\langle\ \mathit{Sensor} \mid \mathit{Gateway}\ ,\ \{\langle T,t,k\rangle, \langle H,h,k\rangle\}\ \rangle\ \to \\
\langle\ \mathit{out}(H,h,k'); \mathit{Sensor} \mid \mathit{Gateway}\ ,\ \{\langle T,t,k\rangle, \langle H,h,k\rangle, \langle T,t,k'\rangle\}\ \rangle
\end{array}$$

in which the sensor produces its first three messages. Notice that the name k, bound inside Sensor by the binder (k), must be renamed in the second iteration. In fact, the label $\langle T,t,k\rangle$ corresponding to the execution of the action $\mathit{out}(T,t,k)$ which is under the scope of the binder (k), becomes $\langle T,t,k\rangle^k$ when it traverses such a binder. At the time the third message is produced, the name k is already in the dataspace, hence the premises of the second rule of Table 2 are not satisfied in that k occurs bound in the label and free in the dataspace. By α-converting k in k', the label of the transition becomes $\langle T,t,k'\rangle^{k'}$, the premises are now satisfied, and the corresponding message can be introduced in the dataspace.

Suppose now that, after the above transitions, the Gateway starts consuming the last message that was produced by the Sensor:

$$\begin{array}{l}
\langle\ \mathit{out}(H,h,k'); \mathit{Sensor} \mid \mathit{Gateway}\ ,\ \{\langle T,t,k\rangle, \langle H,h,k\rangle, \langle T,t,k'\rangle\}\ \rangle \to \\
\langle\ \mathit{out}(H,h,k'); \mathit{Sensor} \mid \mathit{in}(H,(y),k')(\ \mathit{out}(t,y)\ ); \mathit{Gateway}\ , \\
\qquad\qquad \{\langle T,t,k\rangle, \langle H,h,k\rangle\}\ \rangle
\end{array}$$

Notice that in this last state, the Gateway cannot wrongly consume the message $\langle H,h,k\rangle$ as it is willing to access a humidity message with session identifier k', where k' was received by consuming $\langle T,t,k'\rangle$, and bound to the name s in the definition of Gateway (see Definition 2.1).

As discussed in the Introduction, the aim of this section is to recall techniques that were used to investigate the expressive power of the Linda coordination primitives. More precisely, such techniques were adopted to prove the decidability, or undecidability, of relevant properties for the considered calculi. In order to discuss such techniques, we define two analysis problems for the new Linda calculus presented above.

Informally, we consider the problem of checking whether a given initial system can reach a target state in which the dataspace contains certain messages.

In one case, namely *reachability*, we are interested in the dataspace to have a precisely predefined content, in the second case, namely *coverability*, we want the dataspace to contain at least some given messages.

Definition 2.5 (Reachability and Coverability problems). *Let $\langle P, \mathcal{S}\rangle$ be an initial system and let $\mathcal{T}$ be a multiset over Message containing only names that occur free in the process P or in the dataspace $\mathcal{S}$. We say that $\langle P, \mathcal{S}\rangle$* reaches *$\mathcal{T}$, written $\langle P, \mathcal{S}\rangle \downarrow \mathcal{T}$, if there exists P' such that $\langle P, \mathcal{S}\rangle \rightarrow^* \langle P', \mathcal{T}\rangle$ (assuming that no name in $\mathcal{T}$ is freshly generated during the computation). We say that $\langle P, \mathcal{S}\rangle$* covers *$\mathcal{T}$, written $\langle P, \mathcal{S}\rangle \Downarrow \mathcal{T}$, if there exists P' and $\mathcal{S}'$ such that $\langle P, \mathcal{S}\rangle \rightarrow^* \langle P', \mathcal{T} \uplus \mathcal{S}'\rangle$ (assuming that no name in $\mathcal{T}$ is freshly generated during the computation).*

Typical safety properties can be seen as instances of coverability problems. Consider a system $\langle P, \mathcal{S}\rangle$ where internal errors are communicated by placing in the dataspace a specific error message $\langle error\rangle$. We have that such system is not safe (i.e. it is not error-free) if and only if $\langle P, \mathcal{S}\rangle \Downarrow \{\langle error\rangle\}$. Reachability problems, on the other hand, coincide with stronger properties. Consider, for instance, a system in which the generation of the $\langle error\rangle$ message is not problematic if there exists also a $\langle solution\rangle$ message in the dataspace. The above property $\langle P, \mathcal{S}\rangle \Downarrow \{\langle error\rangle\}$ holds even if the dataspace $\{\langle error\rangle, \langle solution\rangle\}$ is reached. This is not the case if we consider the stronger property $\langle P, \mathcal{S}\rangle \downarrow \{\langle error\rangle\}$ that requires the only message $\langle error\rangle$ to be present in the reached dataspace.

In the remainder of this section we prove the following decidability and undecidability results of reachability/coverability on fragments of the calculus:

- if we remove the name generation operator $(\nu a)P$ from the calculus both coverability and reachability are decidable;
- for the full calculus both coverability and reachability are undecidable;
- if we consider a fragment of the calculus (that we call the *local* fragment) in which received names cannot be used in input actions, but only in outputs, coverability is decidable while reachability is not.

The above three results allow us to formally postulate expressiveness results about the three considered fragments: it is not possible to define a translation from the full calculus to the local fragment that preserves coverability, and it is not possible to define a translation from the local fragment (hence also from the full calculus) to the fragment without name generation that preserves reachability.

2.1. Decidability results without name generation

In this subsection we consider processes that do not include name generation (i.e. do not contain the (νa) operator) and we show that in this case we can encode our systems into standard Place/Transition Petri nets. Standard reachability analysis techniques for Petri nets can then be applied to solve both our

reachability and coverability problems. This proof technique, i.e., translate a Linda calculus into Petri nets, has been investigated for the first time by Nadia Busi et al. in [5].

We start by recalling some preliminaries about Petri nets.

A *Petri net* is a tuple $N = (P, T, \mathbf{m_0})$, where P and T are finite sets of *places* and *transitions*, respectively. A finite multiset over the set P of places is called a *marking*, and $\mathbf{m_0}$ is the initial marking. Given a marking $\mathbf{m}$ and a place p, we say that the place p contains a number of *tokens* equal to the number of instances of p in $\mathbf{m}$. A transition $t \in T$ is a pair of markings denoted with ${}^\bullet t$ and $t^\bullet$. A transition t can fire in the marking $\mathbf{m}$ if ${}^\bullet t \subseteq \mathbf{m}$ (where $\subseteq$ is multiset inclusion); upon transition firing the new marking of the net becomes $\mathbf{n} = (\mathbf{m} \setminus^\bullet \mathbf{t}) \uplus \mathbf{t}^\bullet$ (where $\setminus$ and $\uplus$ are the difference and union operators for multisets, respectively). This is written as $\mathbf{m} \mapsto \mathbf{n}$. We use $\mapsto^*$ to denote the reflexive and transitive closure of $\mapsto$. We say that $\mathbf{m'}$ is *reachable* from $\mathbf{m}$ if $\mathbf{m} \mapsto^* \mathbf{m'}$. We say that $\mathbf{m'}$ is *coverable* from $\mathbf{m}$ if $\mathbf{m} \mapsto^* \mathbf{m''}$ with $\mathbf{m'} \subseteq \mathbf{m''}$. Analysis techniques exist for both reachability [21] and coverability [22].

Our encoding of systems into Petri nets is based on the observation that if no names can be freshly generated then the set of processes that can be reached is finite, as well as the possible messages that can be ever produced. This result holds under the assumption that α conversion is not considered, otherwise the number of α convertible processes is clearly infinite. Disallowing α conversion in this fragment is not restrictive; indeed the unique bound names are those that can occur inside input operations, and in this case it is sufficient to assume that they are different from any other free name in the initial process and the initial dataspace to avoid name captures.

We define a Petri net with two kinds of places: places Q corresponding to reachable processes, and places $\tilde{a}$ corresponding to tuples (with a predefined maximal length) of names occurring in the initial process or in the initial dataspace. The transitions in the Petri net correspond to the execution of actions: they consume one token from a place Q and produces a token in Q' if there exists a transition $Q \xrightarrow{\alpha} Q'$, and if $\alpha = \tilde{a}$ then also a token is produced in the place $\tilde{a}$, while if $\alpha = \tilde{\eta}^{[a_j/b_j]_{j \in J}}$ a token is required to be consumed from the corresponding place $\tilde{a}$ (such that $\tilde{\eta}[a_j/b_j]_{j \in J} = \tilde{a}$) in order for the transition to be fired.

Formally, we first define the set of derivatives of a process P with initial multiset of messages $\mathcal{S}$.

Definition 2.6. *Let P be a process without name generation. We denote with $Der(P, \mathcal{S})$ the minimal set of processes including P and closed with respect to the process labeled transition system under the assumption that in a label $\tilde{\eta}^{[a_j/b_j]_{j \in J}}$ all the names a_j occur free in P or in $\mathcal{S}$.*

The first result that we prove is that the set of derivatives of a process P, with initial dataspace $\mathcal{S}$, is always finite.

Lemma 2.1. *Let P be a process without name generation and let $\mathcal{S}$ be a multiset of messages. Then the set of its derivatives $Der(P, \mathcal{S})$ is finite.*

Proof. First of all, let n be the maximal number of bound parameters in an *in* operation. Moreover, let d be the number of names occurring free in P or in $\mathcal{S}$. By induction on the structure of P is immediate to see that $size(P)$, defined as follows, is an upper bound to the cardinality of $Der(P, \mathcal{S})$.

$$size(\mathbf{1}) = 2 \qquad size(out(\tilde{a})) = 3 \qquad size(in(\tilde{\eta})P) = 1 + n \times d \times size(P)$$
$$size(P|Q) = size(P) \times size(Q) \qquad size(P^*) = size(P) + 2$$
$$size(P + Q) = size(P; Q) = 1 + size(P) + size(Q)$$

□

We now define the Petri net encoding and formalise its correctness.

Definition 2.7. *Let $\langle P, \mathcal{S}\rangle$ be a system with P without name generation. Let n be the maximal length of tuples of names in P or in $\mathcal{S}$. We define the Petri Net $PT(\langle P, \mathcal{S}\rangle) = (Der(P, \mathcal{S}) \uplus \{\tilde{a} \mid len(\tilde{a}) \leq n \wedge a_i \in fn(P) \cup n(\mathcal{S})\}, T, \{P\} \uplus \mathcal{S})$ where $n(\mathcal{S})$ is the set of names occurring in $\mathcal{S}$ and T is the following set of transitions:*

$$\{t \mid {}^\bullet t = \{Q\}, t^\bullet = \{Q', \tilde{a}\}, Q \xrightarrow{\tilde{a}} Q'\} \cup$$
$$\{t \mid {}^\bullet t = \{Q, \tilde{a}\}, t^\bullet = \{Q'\}, Q \xrightarrow{\tilde{\eta}^{[a_j/b_j]_{j\in J}}} Q', \tilde{\eta}^{[a_j/b_j]_{j\in J}} = \tilde{a}\} \cup$$
$$\{t \mid {}^\bullet t = \{Q\}, t^\bullet = \{Q'\}, Q \xrightarrow{\surd} Q'\}$$

Note that the Petri net $PT(\langle P, \mathcal{S}\rangle)$ is well defined as its sets of places and transitions are both finite in consequence of Lemma 2.1

Proposition 2.1. *Let $\langle P, \mathcal{S}\rangle$ be a system with P without name generation and let $PT(\langle P, \mathcal{S}\rangle)$ be its corresponding Petri net. We have that $\langle P, \mathcal{S}\rangle \rightarrow^* \langle P', \mathcal{S}'\rangle$ if and only if the marking $\{P'\} \uplus \mathcal{S}'$ is reachable in $PT(\langle P, \mathcal{S}\rangle)$.*

Proof. It is sufficient to proceed by induction on the length of the sequence of transitions in $\langle P, \mathcal{S}\rangle \rightarrow \cdots \rightarrow \langle P', \mathcal{S}'\rangle$. □

We can now conclude that both reachability problems are decidable for the fragment without name creation.

Theorem 2.1. *Let $\langle P, \mathcal{S}\rangle$ be a system with P without name generation, and let $\mathcal{T}$ be a target multiset of names. Then both $\langle P, \mathcal{S}\rangle \downarrow \mathcal{T}$ and $\langle P, \mathcal{S}\rangle \Downarrow \mathcal{T}$ are decidable.*

Proof. As a consequence of Proposition 2.1 it is sufficient to consider the Petri net $PT(\langle P, \mathcal{S}\rangle)$ and check whether one of the markings $\{Q\} \uplus \mathcal{T}$ is reachable (or coverable) for any of the processes $Q \in Der(P, \mathcal{S})$. The decidability result follows from the decidability of reachability and coverability in Petri nets. □

2.2. Undecidability results in the full calculus

We prove the undecidability of reachability and coverability for the full calculus by reduction from the halting problem in Random Access Machines (RAMs). The idea of exploiting RAMs to prove expressiveness results on Linda process

calculi traces back to [4]. A RAM [23], denoted in the following with R, is a computational model composed of a finite set of registers $r_1, \ldots, r_n$, that can hold arbitrary large natural numbers, and by a program composed by indexed instructions $(1 : I_1), \ldots, (m : I_m)$, that is a sequence of simple numbered instructions, like arithmetical operations (on the contents of registers) or conditional jumps. An internal state of a RAM is given by $(i, c_1, \ldots, c_n)$ where i is the program counter indicating the next instruction to be executed, and $c_1, \ldots, c_n$ are the current contents of the registers $r_1, \ldots, r_n$, respectively.

Without loss of generality, we assume that the registers contain the value 0 at the beginning and at the end of the computation, and that the execution of the program begins with the first instruction $(1 : I_1)$. In other words, the initial configuration is $(1, 0, \ldots, 0)$. The computation continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when the instruction *Halt* is reached. More formally, we indicate by $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$ the fact that the configuration of the RAM R changes from $(i, c_1, \ldots, c_n)$ to $(i', c'_1, \ldots, c'_n)$ after the execution of the i-th instruction.

In [24] it is shown that the following two instructions are sufficient to model every recursive function:

- $(i : Succ(r_j))$: adds 1 to the content of register r_j;
- $(i : DecJ(r_j, s))$: if the contents of register r_j is not zero then decreases it by 1 and go to the next instruction, otherwise jumps to instruction s.

We now discuss how to model RAMs in our calculus. The basic idea deals with the representation of the natural numbers in registers: in order to model number c we use a list of concatenated names of corresponding length: $\langle x_c, x_{c-1}\rangle$, $\langle x_{c-1}, x_{c-2}\rangle$, $\cdots$ $\langle x_1, x_0\rangle$, $\langle x_0\rangle$. The entry point of the list for r_j is stored inside a message $\langle r_j, x_c\rangle$. The initial empty lists, one for each register $r_1, \ldots, r_n$, are generated by a bootstrap process

$$\begin{array}{rcl} BootStrap & = & (\nu x_0^1)\big(out(r_1, x_0^1); out(x_0^1)\big); \\ & & \cdots \\ & & (\nu x_0^n)\big(out(r_n, x_0^n); out(x_0^n)\big); \\ & & out(p_1) \end{array}$$

that also generates the message p_1 that will trigger the program execution starting from the first instruction.

The program instructions are modeled as follows: an increment simply adds a new pair in front of the list representing the register content, a decrement by removing the initial pair, and a test-for-zero by checking whether the entry point of the list corresponds with the ending tuple $\langle x_0^i\rangle$. More precisely, we

model the program instructions as follows:

$$
\begin{array}{lcl}
[\![(i : Succ(r_j))]\!] & = & \big(\ in(p_i); in(r_j,(x))\big((\nu y) out(r_j,y); out(y,x)\big); \\
& & \qquad out(p_{i+1})\ \big)^* \\
[\![(i : DecJ(r_j,s))]\!] & = & \big(\ in(p_i); in(r_j,(x))\big(in(x)(out(x); out(r_j,x)); out(p_s)\ + \\
& & \qquad in(x,(y)) out(r_j,y); out(p_{i+1})\ \big)^* \\
[\![(i : Halt)]\!] & = & in(p_i); in(r1,(x_1)) in(x_1); \\
& & \qquad \cdots \\
& & \qquad in(r_n,(x_n)) in(x_n); out(h)
\end{array}
$$

The i-th instruction is modeled by a process that can repeatedly consume a program counter datum p_i, require the execution of a register action (either increment, decrement or test for zero), and finally produce the subsequent program counter p_{i+1} (or p_s in case of jump to the s-th instruction). In case of increment, it simply adds a pair to the list corresponding to the register to increment. In case of decrement or jump instruction, the process checks whether the list is empty or not: in the first case the empty register is re-produced and p_s produced, in the second one the first pair of the list is removed and p_{i+1} produced. A final *Halt* instruction removes the registers from the encoding and produce the datum h, that will remain the unique message in the dataspace at the end of the computation. In this case, we use the assumption that registers are empty at the end of the computation.

We complete with the full definition of our encoding of a RAM R:

$$[\![R]\!] \ = \ BootStrap \mid \prod_{1 \le i \le m} [\![(i : I_i)]\!]$$

We can now state the correctness of our encoding.

Theorem 2.2. *Let R be a RAM. We have that R terminates if and only if $\langle [\![R]\!], \emptyset\rangle \Downarrow \{\langle h\rangle\}$. Moreover, $\langle [\![R]\!], \emptyset\rangle \Downarrow \{\langle h\rangle\}$ if and only if $\langle [\![R]\!], \emptyset\rangle \downarrow \{\langle h\rangle\}$.*

Proof. The encoding proceeds deterministically by reproducing faithfully the corresponding RAM instructions. When a $Halt$ instruction is reached, according with the assumption that the registers will be empty at the end of the computation, we can assume that the register lists will be removed and the tuple $\langle h\rangle$ will be produced. Moreover, as the lists have been removed, the tuple $\langle h\rangle$ will be the unique datum inside the dataspace. □

As a corollary we have the undecidability of both reachability and coverability for our calculus.

2.3. The local fragment

In this subsection we identify an intermediary fragment of the calculus for which reachability is undecidable while coverability is decidable. The technique used in undecidability proof is still by encoding RAMs in the calculus, but in this case the encoding in nondeterministic: there are several computations that

can be executed by the encoding, but all wrong computations either block before reaching the *Halt* instruction or leave in the dataspace some "garbage" messages. On the other hand, the decidability proof exploits techniques borrowed from the theory of Well Structured Transition Systems (WSTSs) [25]. To the best of our knowledge, one of the first applications of WSTS to calculi based on the notion of shared dataspace, is by Busi and Zavattaro who proved decidability results for the Mobile Ambients calculus [26]. Mobile Ambients can be seen as an extension of Linda with multiple nested dataspaces and mobility primitives that can be used to dynamically modify the dataspace nesting topology: one dataspace can enter in a sibling space, one dataspace can exit from an enclosing space, a dataspace can open an enclosed dataspace to acquire its contents.

We first characterize the fragment of the calculus considered in this subsection, then we prove the undecidability of reachability, and then the decidability of coverability, for such fragment.

The idea behind the definition of the fragment is to limit the use of names that are received from the dataspace. When a name is received by means of a bound name in the input pattern, then such name can be used only in output. Formally, given a term $in(\tilde{\eta})P$, for every $(b) \in \eta$, we impose that b cannot occur free in input operations of P. Under this restriction, we have that even if fresh names can be passed around, these cannot be used to specify new input patterns. We call this the *local* fragment of the calculus because new names can be used in input patterns only locally, in the initial scope of the name. In the remainder of this section we will consider only processes of the local fragment.

Notice, for instance, that the process *Gateway* of the Example 2.1 does not belong to the local fragment of the calculus. In fact, it receives a session identifier while reading a temperature message, and then it uses such an identifier in the subsequent input, in order to be sure to consume the corresponding humidity message.

2.3.1. Undecidability of reachability for the local fragment

Also in this case, the proof is by reduction from the halting problem in RAMs.

We start by presenting the encoding of program instructions:

$$\begin{array}{lcl}
[\![(i : Succ(r_j))]\!] & : & \big(\ in(p_i); out(inc_j); in(ack); out(p_{i+1})\ \big)^* \\
[\![(i : DecJ(r_j, s))]\!] & : & \big(\ in(p_i); out(dec_j); in(ack); out(p_{i+1}) + \\
 & & \quad in(p_i); out(zero_j); in(ack); out(p_s)\ \big)^* \\
[\![(i : Halt)]\!] & : & in(p_i); out(h)
\end{array}$$

Also in this case, we use a program counter datum p_i that triggers the execution of the i-th instruction. Differently, in this case the instructions request the execution of a register action (either increment, decrement or test for zero) to other processes (that we will describe in the following), wait for an acknowledgement about the actual execution of the register action, and finally produce the subsequent program counter p_{i+1} or p_s. Notice that in the case of a $DecJ(r_j, s)$ instruction, the process non deterministically selects whether to decrement or

test for zero the register r_j. In case of a *Halt* instruction the program counter is not produced thus program execution is terminated (termination is notified also in this case by producing h).

We now consider the modeling of registers:

$$[\![r_j]\!] \;:\quad \Big(\; in(nr_j); out(ack); (\nu u)\; \Big(\quad \big(in(inc_j); out(u); out(ack)\big)^* \;\mid \big(in(dec_j); in(u); out(ack)\big)^* \;\mid in(zero_j); out(nr_j)\;\Big)\;\Big)^*$$

Each register r_j is activated by a request nr_j whose effect is (after generation of the acknowledgement) to generate a private name u used to represent the register contents. Namely, the content c_j of the register r_j is represented by c_j instances of the private datum u. Being private, the messages u do not generate interferences among the encodings of the different registers (i.e. it is not possible to wrongly consider a datum u associated to a register different from the one for which it was generated). An instance of u is produced when an increment request is received, and it is consumed in case of decrement operations. It is worth noticing that it could happen that a decrement request is received when the register is empty. In this case the computation blocks because the $in(u)$ action cannot be executed.

Notice also that when a test for zero request is received, a new local name u is generated by emitting nr_j whose effect is to reset the register. If a non empty register is reset, instances of the previous private datum u will indefinitely remain in the dataspace.

The full definition of our encoding of a RAM R is as follows:

$$\begin{array}{rcl}[\![R]\!] & = & \prod_{1\leq i\leq m} [\![(i : I_i)]\!] \mid \prod_{1\leq j\leq n} [\![r_j]\!] \mid \\ & & out(nr_1); in(ack); \cdots ; out(nr_n); in(ack); out(p_1)\end{array}$$

Besides the processes modeling the instructions and the registers we need a bootstrap process that activates the registers and then produces the initial program counter datum p_1.

We can now conclude by stating the correctness of our encoding.

Theorem 2.3. *Let R be a RAM. We have that R terminates if and only if $\langle [\![R]\!], \emptyset\rangle \downarrow \{h\}$.*

Proof. We start with the *only if* part. Assume R terminates. The system $\langle [\![R]\!], \emptyset\rangle$ can faithfully reproduce the same sequence of increment, decrement and test for zero actions until the *Halt* instruction is reached. In the finally reached system the dataspace will contain the h datum only, as we have assumed RAMs that starts and ends with all the registers empty.

We now consider the *if* part. Assume that $\langle [\![R]\!], \emptyset\rangle$ reaches a system in which the dataspace contains only h. The presence of h indicates that the last executed instruction is *Halt*. The absence of data different from h guarantees not only that the registers are empty at the end of the computation, but also that the sequence of executed register actions is correct. In fact, as observed

above, the encoding $[\![R]\!]$ could wrongly simulate *DecJ* instructions, by deciding to decrement a register when it is empty, or test it for zero when it is not empty. The first kind of error cannot occur because otherwise the computation blocks (hence no datum h could be produced). The second kind of error cannot occur because otherwise some private data u will remain in the dataspace (representing the non empty content of the register at the time a wrong test for zero is executed). □

As a corollary, from the undecidability of the halting problem for RAMs, we can conclude the undecidability of reachability for our calculus.

2.4. Decidability of coverability for the local fragment

As mentioned above, the proof of decidability of coverability for the local fragment exploits results taken from the theory of WSTS. We now recall the main concepts of WSTS that are useful in our proof.

We start by recalling the notion of well quasi ordering (see, e.g., [25]).

Definition 2.8. *A reflexive and transitive relation on a set X is called* quasi ordering. *A* well quasi ordering *(wqo) is a quasi ordering $(X, \leq)$ such that, for every infinite sequence $x_1, x_2, \cdots$, there exist $i < j$ with $x_i \leq x_j$.*

In the following we will use the following well known results for wqo:

- The identity on a finite set is a wqo.
- Consider a finite set S and the set of its multisets $\mathcal{M}(S)$. We have that multiset inclusion is a well quasi ordering for the latter, namely $(\mathcal{M}(S), \subseteq)$ is a wqo, where $\subseteq$ denotes multiset inclusion.
- Consider k well quasi orderings $(X_1, \leq_1), \cdots, (X_k, \leq_k)$. Let Π be the cartesian product $X_1 \times \cdots \times X_k$ and $\leq^k$ be the natural extension of the orderings $\leq_1, \cdots, \leq_k$ to Π, i.e., $(x_1, \cdots, x_k) \leq^k (y_1, \cdots, y_k)$ if and only if $x_1 \leq_1 y_1, \cdots, x_k \leq_k y_k$. We have that $(\Pi, \leq^k)$ is a wqo.

We now recall, taking it from [25], the notion of compatibility[3] of a transition system w.r.t. an ordering.

Definition 2.9. *A transition system $(X, \rightarrow)$ is* compatible *with respect to an ordering $(X, \leq)$ if, given two states $s, t \in X$ of the transition system such that $s < t$ and $s \rightarrow s'$ for some s', then there exists t' such that $s' \leq t'$ and $t \rightarrow t'$.*

In transition systems compatible w.r.t. a wqo, the coverability problem can be solved by applying a backward algorithm; this is possible under the additional assumption that given a state t it is possible to compute its *pred_basis*, i.e., a finite characterization of the possible predecessors of states covering t, that is, those states s such that $s \rightarrow t'$ with t' above t (i.e., $t \leq t'$).

[3] The compatibility notion used in this paper is named *strict* compatibility in [25].

Definition 2.10. *Consider a transition system $(X, \rightarrow)$ and a corresponding ordering $(X, \leq)$. Given $t \in X$ and $T \subseteq X$ we define $Pred(t) = \{s | s \rightarrow t\}$, $Pred(T) = \cup_{t \in T} Pred(t)$, $\uparrow t = \{t' | t \leq t'\}$ and $\uparrow T = \cup_{t \in T} \uparrow t$. A pred_basis of state t is a finite set of states T such that $\uparrow T = \uparrow Pred(\uparrow T)$.*

We now recall the main decidability result for WSTSs that we will use in the following

Theorem 2.4 (Theorem 3.6 in [25]). *Let $(X, \rightarrow)$ be a transition system compatible w.r.t. a wqo $(X, \leq)$ and with* effective *pred_basis (i.e. given a state it is possible to compute its pred_basis). Given an initial state s and a target state t, the existence of a computation starting from s and reaching any state t', such that $t \leq t'$, is decidable.*

To exploit this result, we need to define a wqo on systems of our calculus which is compatible for the transition system defining the system semantics. Concerning the dataspaces, which are multisets of messages, one could think of using the above result stating that multiset inclusion, over multisets with a finite domain, is a wqo. Unfortunately, due to the presence of the new name generation operator $(\nu a)P$, unboundedly many different messages can be generated, hence the domain of such multisets is not finite. Also for processes, the new name generation mechanism allows for the generation of unboundedly many distinct processes. This difficulty in applying WSTS theory directly on the calculus can be overtaken by introducing an alternative semantics for the local fragment. Such alternative semantics has the following property: the set of used names is finite as well as the set of possible processes.

An alternative semantics for the local fragment

We now define an alternative semantics for the local fragment of our calculus in which the *active* names are finite. By active, we mean a name which occurs free in input actions. The number of active names turns out to be bound by the initial number of free names plus the number of occurrences of the (νa) operator in the initial process. In fact, when an instance of the (νa) operator inside a cyclic process generates a new name, the name it generated at the previous cycle is no longer active. The alternative semantics that we define, instead of generating always fresh names, re-use the same name considered at the previous cycle. Namely, each time an instance of (νa) requires the generation of a new name, we always use a corresponding name a'. When a' must be re-generated inside a new cycle, the instances of a' left in the dataspace or in other processes are all renamed with a dummy name $\bullet$, to indicate that they are no longer active. In this way, only active names are faithfully represented while the names that are no-longer active are all represented with the same dummy name $\bullet$.

This specific discipline in the re-use of names has also another effect: it is no longer necessary to use α-conversion under the assumption that all the names a used by (νa) occurring in the initial process are pairwise distinct and different from the initial free names.

Definition 2.11 (Alternative semantics). *Let P be a process: without loss of generality we assume that all the names occurring in binders are pairwise distinct and different from the free names in P. Moreover, for each a in a (νa) operator occurring in P, consider a fresh name a'. Consider now the labeled transition system for processes $\stackrel{\alpha}{\Longrightarrow}$ defined as the one in Definition 2.3, but without considering α-conversion. The alternative system reduction relation $\Rightarrow$ is defined as in Definition 2.4 where $\stackrel{\alpha}{\Longrightarrow}$ is used instead of $\stackrel{\alpha}{\longrightarrow}$ and the second rule of Table 2 is replaced by the following one:*

$$\frac{P \stackrel{\tilde{b}^{a_1 \cdots a_n}}{\Longrightarrow} P'}{\langle P, \mathcal{S} \rangle \Rightarrow \langle (P'[{}^{\bullet}/_{a'_i}]_{i=1\cdots n})[{}^{a'_i}/_{a_i}]_{i=1\cdots n}, \mathcal{S}[{}^{\bullet}/_{a'_i}]_{i=1\cdots n} \uplus \tilde{b}[{}^{a'_i}/_{a_i}]_{i=1\cdots n} \rangle}$$

where the substitutions are used to replace all previous instances of names a'_i with $\bullet$, and the fresh instances of a_i with a'_i.

We now formalize the correspondence between the alternative and the standard semantics. We consider two distinct propositions, one showing how standard transitions are mimicked by the alternative semantics, and another one for the vice versa. In these propositions we apply renaming functions to processes and messages in the dataspace: $P[f]$ (resp. $\mathcal{S}[f]$) is the process (resp. the dataspace) obtained by replacing names according to the renaming function f. More formally, f is a function from *Name* to *Name*, and $P[f]$ (resp. $\mathcal{S}[f]$) is obtained by replacing in P (resp. $\mathcal{S}$) each occurrence of $a \in$ *Name* with $f(a)$. With an abuse of notation, we use $f(\alpha)$ to indicate the label obtained by replacing in α each occurrence of $a \in$ *Name* with $f(a)$.

Proposition 2.2. *Let $\langle P, \mathcal{S} \rangle$ be a system with P such that all the names occurring in binders are pairwise distinct and different from the free names in P and from those in $\mathcal{S}$. If $\langle P, \mathcal{S} \rangle \rightarrow^* \langle P', \mathcal{S}' \rangle$ then there exists a renaming function f such that $\langle P, \mathcal{S} \rangle \Rightarrow^* \langle P'[f], \mathcal{S}'[f] \rangle$ with f identity for the free names in the initial configuration $\langle P, \mathcal{S} \rangle$ and injective on the free names occurring in input actions in P'.*

Proof. The proof is by induction on the length of the sequence of transitions $\langle P, \mathcal{S} \rangle \rightarrow^* \langle P', \mathcal{S}' \rangle$. The base case is trivial. In the inductive case we have $\langle P, \mathcal{S} \rangle \rightarrow^* \langle P'', \mathcal{S}'' \rangle \rightarrow \langle P', \mathcal{S}' \rangle$ with $\langle P, \mathcal{S} \rangle \Rightarrow^* \langle P''[f'], \mathcal{S}''[f'] \rangle$ for some function f' identity for the free names in $\langle P, \mathcal{S} \rangle$ and injective for the free names occurring in input actions in P''. We proceed with a case analysis on the transition $P'' \stackrel{\alpha}{\longrightarrow} P'$ used to infer the last transition $\langle P'', \mathcal{S}'' \rangle \rightarrow \langle P', \mathcal{S}' \rangle$.

We start from the simplest case: $P'' \stackrel{\surd}{\longrightarrow} P'$. The possibility to terminate, i.e., perform a $\surd$ transition, is independent from the actually used names because the terminating processes are $\mathbf{1}$ and P^* (and combinations of them). Hence we simply observe that also $P''[f'] \stackrel{\surd}{\Longrightarrow} P'[f']$. This implies the thesis $\langle P''[f'], \mathcal{S}''[f'] \rangle \Rightarrow \langle P'[f'], \mathcal{S}'[f'] \rangle$.

Consider now $P'' \stackrel{\tilde{\eta}^{[a_j/b_j]_{j\in J}}}{\longrightarrow} P'$. In this case we have the corresponding transition $P''[f'] \stackrel{f'(\tilde{\eta}^{[a_j/b_j]_{j\in J}})}{\Longrightarrow} P'[f']$, from which $\langle P''[f'], \mathcal{S}''[f']\rangle \Rightarrow \langle P'[f'], \mathcal{S}'[f']\rangle$ holds because f' renames the processes as well as the dataspace. Hence, a message that could be consumed by an input operation before renaming, could be consumed also after application of the renaming. We have that f' continue to be the identity for the free names in $\langle P, \mathcal{S}\rangle$ and injective for the free names occurring in input actions in P'. In fact, the names received by the executed input, i.e. $\{f'(b_j) \mid j \in J\}$, cannot replace parameters of input operations. This follows from the restrictions imposed in the local fragment.

The last case is for $P'' \stackrel{\tilde{a}^{b_1\cdots b_n}}{\longrightarrow} P'$. In this case we have the corresponding transition $P''[f'] \stackrel{f'(\tilde{a}^{b_1\cdots b_n})}{\Longrightarrow} P'[f']$, from which $\langle P''[f'], \mathcal{S}''[f']\rangle \Rightarrow \langle P'[f], \mathcal{S}'[f]\rangle$ for a renaming f defined as follows. Let a_i be the names to which b_i is mapped by f' (i.e. $f'(b_i) = a_i$). According to the alternative semantics, a_i is the name used in the instance (νa_i) occurring in the initial process P which is α-renamed to (νb_i) in P''. In the alternative semantics, (νa_i) is not α-renamed, but the corresponding name a'_i is used instead of b_i; hence the renaming function f must map b_i to a'_i (i.e. $f(b_i) = a'_i$). Moreover, in $\langle P''[f'], \mathcal{S}''[f']\rangle$, if there are free occurrences of a'_i, these are renamed to $\bullet$ by the transition $\langle P''[f'], \mathcal{S}''[f']\rangle \Rightarrow \langle P'[f], \mathcal{S}'[f]\rangle$. Let b'_i be the corresponding names in $\langle P'', \mathcal{S}''\rangle$ (i.e. $f'(b'_i) = a'_i$); the new renaming function f must map b'_i to $\bullet$ (i.e. $f(b'_i) = \bullet$). This new renaming f continue to be injective on names that occur in input actions. This follows from the fact that the names b'_i which are all mapped to $\bullet$ by f cannot indeed occur in input positions. In fact, these correspond to new names generated by the corresponding operators (νa_i). In the local fragment of the calculus, these names can be used in input actions only inside the scope of the binder (νa_i). But, if this operator generates a new name, this means that it is inside a repetition P^*, and the previous execution of the repetition has been already completed. Hence also the scope of the previous (νa_i) instance has been completed. □

Similarly, we have that all the sequences of transitions in the alternative semantics, have corresponding computations in the standard semantics.

Proposition 2.3. *Let $\langle P, \mathcal{S}\rangle$ be a system with P such that all the names occurring in binders are pairwise distinct and different from the free names in P and from those in $\mathcal{S}$. If $\langle P, \mathcal{S}\rangle \Rightarrow^* \langle P', \mathcal{S}'\rangle$ then there exists $\langle P, \mathcal{S}\rangle \rightarrow^* \langle Q, \mathcal{T}\rangle$ and a renaming f identity for the free names in the initial configuration $\langle P, \mathcal{S}\rangle$ and injective on the free names occurring in input actions in Q such that $Q[f] = P'$ and $\mathcal{T}[f] = \mathcal{S}'$.*

We can finally prove that the alternative semantics preserves the coverability problem for the local fragment.

Theorem 2.5. *Let $\langle P, \mathcal{S}\rangle$ be a system with P such that all the names occurring in bonders are pairwise distinct and different from the free names in P and from those in $\mathcal{S}$. Given $\mathcal{T}$ a multiset over Message containing only names that occur*

free in the process P or in the dataspace $\mathcal{S}$, we have that $\langle P, \mathcal{S}\rangle \Downarrow \mathcal{T}$ if and only if $\langle P, \mathcal{S}\rangle \Rightarrow^* \langle P', \mathcal{S}'\rangle$ with $\mathcal{T} \subseteq \mathcal{S}'$.

Proof. The *only-if* part follows from Proposition 2.2. If $\langle P, \mathcal{S}\rangle \Downarrow \mathcal{T}$ then there exists a system $\langle P'', \mathcal{S}''\rangle$ such that $\langle P, \mathcal{S}\rangle \rightarrow^* \langle P'', \mathcal{S}''\rangle$ with $\mathcal{T} \subseteq \mathcal{S}''$. By Proposition 2.2 there exists $\langle P', \mathcal{S}'\rangle$ such that $\langle P, \mathcal{S}\rangle \Rightarrow^* \langle P', \mathcal{S}'\rangle$ with $\mathcal{S}' = \mathcal{S}''[f]$ with f identity for the free names in the initial configuration. Being $\mathcal{T}$ composed of only names free in the initial configuration, we have also that $\mathcal{T} \subseteq \mathcal{S}'$.

The *if* part follows from Proposition 2.3. Assume $\langle P, \mathcal{S}\rangle \Rightarrow^* \langle P', \mathcal{S}'\rangle$ with $\mathcal{T} \subseteq \mathcal{S}'$. By Proposition 2.3 there exists $\langle P'', \mathcal{S}''\rangle$ such that $\langle P, \mathcal{S}\rangle \rightarrow^* \langle P'', \mathcal{S}''\rangle$ and a renaming f, identity for the free names in the initial configuration $\langle P, \mathcal{S}\rangle$, such that $\mathcal{S}''[f] = \mathcal{S}'$. Being $\mathcal{T}$ composed of only names free in the initial configuration, we have also that $\mathcal{T} \subseteq \mathcal{S}''$, hence also $\langle P, \mathcal{S}\rangle \Downarrow \mathcal{T}$. □

Resorting to the theory of WSTS

We now move to the proof of decidability of coverability for the alternative semantics of the local fragment. The first step is to define a wqo for the alternative transition system: this is now possible because the number of reachable processes is finite, as well as the names used in the dataspace. This finiteness result follows from the re-use, of the predefined names a', when new names are needed.

Proposition 2.4. *Let $\langle P_0, \mathcal{S}_0\rangle$ be a system with P_0 such that all the names occurring in binders are pairwise distinct and different from the free names in P_0 and from those in $\mathcal{S}_0$. Let Der_P and $Der_{\mathcal{S}}$ be the reachable processes and the reachable dataspaces, i.e., $Der_P = \{P | \langle P_0, \mathcal{S}_0\rangle \Rightarrow^* \langle P, \mathcal{S}\rangle\}$ and $Der_{\mathcal{S}} = \{\mathcal{S} | \langle P_0, \mathcal{S}_0\rangle \Rightarrow^* \langle P, \mathcal{S}\rangle\}$. We have that Der_P is finite while $Der_{\mathcal{S}}$ contains multisets of messages taken from a finite domain.*

Proof. We start by showing that $Der_{\mathcal{S}}$ contains messages taken from a finite domain. Messages are tuples of names; the length of the tuples in the reachable dataspaces is bound by *max*, where *max* is the greatest value between the maximal length of the tuples in the initial dataspace $\mathcal{S}_0$ and the maximal number of parameters in an *out* operation. Moreoveor, the names occuring in such tuples are taken from the following finite set: $fn(P) \cup n(\mathcal{S}_0) \cup \{a' \mid (\nu a) \text{ occurs in } P\} \cup \bullet$ where $n(\mathcal{S}_0)$ are the names occurring in $\mathcal{S}_0$. Obviously, the possible tuples of bound length, containing names taken from a finite set, are finite as well.

Concerning Der_P, as in the proof of Lemma 2.1, we prove by induction on the structure of P that a function $size_a(P)$ returns an upper bound to the cardinality of Der_P.

$$
\begin{array}{ll}
size_a(\mathbf{1}) = 2 & size_a(out(\tilde{a})) = 3 \\
size_a(in(\tilde{\eta})P) = 1 + n \times d \times size_a(P) & size_a((\nu a)P) = size_a(P) \\
size_a(P|Q) = size_a(P) \times size_a(Q) & size_a(P^*) = size_a(P) + 2 \\
size_a(P + Q) = size_a(P; Q) = 1 + size_a(P) + size_a(Q) &
\end{array}
$$

where, in this case, n is the maximal number of bound parameters in an *in* operation, and d is the cardinality of the set of possible names $fn(P) \cup n(\mathcal{S}_0) \cup \{a' \mid (\nu a) \text{ occurs in } P\} \cup \bullet$ discussed above. □

We are now ready to define an ordering on systems which is a wqo when applied to the systems that can be reached from an initial one:

$$\langle P, \mathcal{S} \rangle \leq_S \langle P', \mathcal{S}' \rangle \quad \Leftrightarrow \quad P = P' \wedge \mathcal{S} \subseteq \mathcal{S}'$$

Notice that the ordering $\leq_S$ simply combines the identity on processes and multiset inclusion on dataspaces.

Proposition 2.5. *Let $S_0 = \langle P_0, \mathcal{S}_0 \rangle$ be a system with P_0 such that all the names occurring in binders are pairwise distinct and different from the free names in P_0 and from those in $\mathcal{S}_0$. Let Sys be the set of systems that are reachable from $\langle P_0, \mathcal{S}_0 \rangle$ according to the alternative semantics, i.e., $Sys = \{\langle P, \mathcal{S} \rangle \, | \langle P_0, \mathcal{S}_0 \rangle \Rightarrow^* \langle P, \mathcal{S} \rangle\}$. We have that $(Sys, \leq_S)$ is a wqo, and that $(Sys, \Rightarrow)$ is compatible with $(Sys, \leq_S)$.*

Proof. The ordering $(Sys, \leq_S)$ is a wqo as a direct consequence of Proposition 2.4 and of the three well known results about wqo recalled after Definition 2.8: in fact $\leq_S$ simply combines two orderings, the identity on the reachable processes (that are finite) and multiset inclusion on the reachable dataspaces (that are multisets on a finite domain).

We now consider the compatibility of $(Sys, \Rightarrow)$ w.r.t. the ordering $\leq_S$. Consider $\langle P_1, \mathcal{S}_1 \rangle \leq_S \langle P_2, \mathcal{S}_2 \rangle$ and $\langle P_1, \mathcal{S}_1 \rangle \Rightarrow \langle P_1', \mathcal{S}_1' \rangle$. Being $\langle P_1, \mathcal{S}_1 \rangle \leq_S \langle P_2, \mathcal{S}_2 \rangle$, we have that $P_1 = P_2$ and $\mathcal{S}_1 \subseteq \mathcal{S}_2$. It is easy to see that the process transition used to infer $\langle P_1, \mathcal{S}_1 \rangle \Rightarrow \langle P_1', \mathcal{S}_1' \rangle$ can be used to infer also $\langle P_1, \mathcal{S}_2 \rangle \Rightarrow \langle P_1', \mathcal{S}_2' \rangle$, with $\mathcal{S}_1' \subseteq \mathcal{S}_2'$. Hence, we have that $\langle P_1', \mathcal{S}_1' \rangle \leq_S \langle P_1', \mathcal{S}_2' \rangle$. □

We are finally ready to prove the decidability of coverability for the local fragment.

Theorem 2.6. *Let $\langle P, \mathcal{S} \rangle$ be a system of the local fragment and let $\mathcal{T}$ be a target multiset of messages. Then $\langle P, \mathcal{S} \rangle \Downarrow \mathcal{T}$ is decidable.*

Proof. It is not restrictive to assume P such that all the names occurring in (νa) operators are pairwise distinct and different from the free names in P_0 and from those in $\mathcal{S}_0$; in fact, if this is not the case, it is sufficiente to apply appropriate α-conversions.[4]

By Theorem 2.5 we have that $\langle P, \mathcal{S} \rangle \Downarrow \mathcal{T}$ if and only if $\langle P, \mathcal{S} \rangle \Rightarrow^* \langle P', \mathcal{S}' \rangle$ with $\mathcal{T} \subseteq \mathcal{S}'$. We now prove that the latter is decidable by applying the results for WSTS to the transition system $(Sys, \Rightarrow)$ defined as in Proposition 2.5, i.e., $Sys = \{\langle P', \mathcal{S}' \rangle \, | \langle P, \mathcal{S} \rangle \Rightarrow^* \langle P', \mathcal{S}' \rangle\}$.

By Proposition 2.5 we know that $(Sys, \leq_S)$ is a wqo, and that $(Sys, \Rightarrow)$ is compatible with $(Sys, \leq_S)$. We now show that it is possible to algorithmically decide whether $\langle P, \mathcal{S} \rangle \Rightarrow^* \langle P', \mathcal{S}' \rangle$ with $\mathcal{T} \subseteq \mathcal{S}'$, by resorting to the decidability result of WSTS recalled in Theorem 2.4.

[4] It is possible to consider α-conversion because in the statement of the Theorem we consider $\langle P, \mathcal{S} \rangle \Downarrow \mathcal{T}$, which is defined on the standard semantics including α-conversion.

By Proposition 2.4, we have that the possible processes P' in Sys are finite; let Der_P be such finite set of processes. Moreover, $\mathcal{T} \subseteq \mathcal{S}'$ implies that, for every P', $\langle P', \mathcal{T} \rangle \leq_S \langle P', \mathcal{S}' \rangle$. Hence, our problem consists of checking, if there exists at least one P', taken from the finite set Der_P, such that $\langle P, \mathcal{S} \rangle \Rightarrow^* \langle P', \mathcal{S}' \rangle$ with $\langle P', \mathcal{T} \rangle \leq_S \langle P', \mathcal{S}' \rangle$. By Theorem 2.4 we have that the latter is decidable under the assumption that it is possible to compute the so-called *pred_basis* for each state of the transition system $(Sys, \Rightarrow)$. We complete the proof by showing that this actually holds.

Let $\langle Q, \mathcal{U} \rangle \in Sys$; we now show how to compute its corresponding *pred_basis* First of all let $\mathcal{Q} = \{Q' \in Der_P | Q' \stackrel{\alpha}{\Longrightarrow} Q\}$: this set is finite (and computable) as Der_P is finite. For each process $Q' \in \mathcal{Q}$, depending on the kind of process transition $Q' \stackrel{\alpha}{\Longrightarrow} Q$, we add a predecessor system $\langle Q', \mathcal{U}_Q \rangle$ to the *pred_basis*. There are three kinds of transition: those corresponding to an input (i.e., $Q' \stackrel{\tilde{\eta}^{[a_j/b_j]_{j \in J}}}{\Longrightarrow} Q$), to an output (i.e., $Q' \stackrel{\tilde{b}^{a_1 \cdots a_n}}{\Longrightarrow} Q$), and the termination transitions (i.e., $Q' \stackrel{\surd}{\Longrightarrow} Q$). In the first case, the input action consumes from the dataspace the tuple $\tilde{a} = \tilde{\eta}[a_j/b_j]_{j \in J}$; in this case we consider the predecessor system $\langle Q', \mathcal{U} \uplus \{\tilde{a}\} \rangle$. In the second case, the emitted message is $\tilde{b}$; in this case we consider the predecessor system $\langle Q', \mathcal{U} \setminus \{\tilde{b}\} \rangle$. In the third case, we have that the process transition does not modify the dataspace, hence we consider the predecessor system $\langle Q', \mathcal{U} \rangle$. The set containing all such predecessors (computable because finite) is a *pred_basis* for $\langle Q, \mathcal{U} \rangle$. □

3. Behavioural Contracts

Behavioural contracts are used to describe the message-passing behaviour of services. The adoption of process calculi for the specification and analysis of behavioural contracts was initiated by Fournet et al. [27], who proposed to specify contracts with a calculus inspired by CCS [28]. They also defined a notion of conformance between processes and contracts following a substitution principle: a process conforms to a contract if it can replace it in any context without adding additional stuck behaviour. Contract have been subsequently studied in the context of service oriented computing: contracts for client-service interaction have been proposed by Carpineti et al. [10] and then independently extended along different directions by, e.g., Bravetti and Zavattaro (see e.g. [11, 12, 29]) by Laneve and Padovani [30], by Castagna et al. [31], and Barbanera and de'Liguoro [32].

All such theories of contracts introduce, under different assumptions, notions of *contract refinements* that can be seen as generalizations of the notion of conformance initially studied in [27]: a contract refines another one if it can safely replace it in any possible context. To give to the reader an idea of such techniques, here we report the contract theory discussed in [29, 33], for synchronous communication, and [13], for the asynchronous case. In particular, the latter represents the unique contract theory, to the best of our knowledge, specifically tailored to asynchronous communication.

More precisely, the contract theory that we present is based on the following ingredients: the notion of correct contract composition, the definition of contract refinement, and its algorithmic characterization. The theory considers both synchronous and asynchronous communication, excluding the algorithmic characterization which is available only for the synchronous case.

3.1. The Service Calculus

We start by presenting the formal definition of services following the approach of [29, 33]. Services are assumed to reside at a certain location over the network. They are denoted by representing service behaviour with essentially the same syntax as the Linda calculus of the previous Section 2, but considering channel based (instead of tuple based) communication actions/operations as in the context of Service Oriented Computing. We assume a denumerable set of action names $\mathcal{N}$, ranged over by $a, b, c, \ldots$ and a denumerable set *Loc* of location names, ranged over by $l, l', l_1, \cdots$. We use $\tau \notin \mathcal{N}$ to denote an internal (unsynchronizable) service computation. Moreover, we use a and $\overline{a}_l$, with $a \in \mathcal{N}$, to denote service input and output messages, respectively, where a destination location (identifying the service to which the message is sent) is specified for outputs. Finally, we represent inner service communication, modeling, e.g., internal process message exchange: we use a_* and $\overline{a}_*$, with $a \in \mathcal{N}$, to denote internal service input and output messages, where processes executing a_* and $\overline{a}_*$ must synchronize in order to proceed (as in CCS [28] synchronization).

Definition 3.1. (Services) *The syntax of services is defined by the following grammar*

$$\begin{array}{rcl} S & ::= & \mathbf{1} \mid \tau \mid a_* \mid \overline{a}_* \mid a \mid \overline{a}_l \mid \\ & & S{+}S \mid S|S \mid S;S \mid S^* \end{array}$$

In the process algebra we use, as for the Linda calculus of Section 2: the choice $\_ + \_$, parallel $\_|\_$, sequential $\_;\_$, and repetition $\_^*$ operators and "$\mathbf{1}$" denoting (successful) termination.[5] Notice that a term S represents the behavior of a single service: the parallel $\_|\_$ operator is used to represent its internal processes. Subsequently we will introduce systems, where communication among several services (residing at different locations over the network) is modeled, considering both the case of synchronous and asynchronous communication.

The operational semantics of services is defined in terms of a transition system labeled over $\{a, \overline{a}_l, a_*, \overline{a}_*, \tau, \surd \mid a \in \mathcal{N}, l \in Loc\}$, ranged over by λ, λ', $\ldots$, obtained by the rules in Table 3 (plus symmetric rules). As in the previous section, in the operational semantics we use an auxiliary process "$\mathbf{0}$" as the target of the termination transitions $\surd$. Semantic rules are the standard ones: choice $\_ + \_$, parallel $\_|\_$, sequential $\_;\_$, and repetition $\_^*$ operators and "$\mathbf{1}$"

[5] In the following, when writing services, we omit parentheses under the assumption that sequencing $\_;\_$ takes priority over choice $\_ + \_$ and parallel $\_|\_$.

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

$$\frac{S_1 \xrightarrow{\lambda} S_1'}{S_1 + S_2 \xrightarrow{\lambda} S_1'} \qquad \frac{S_1 \xrightarrow{\lambda} S_1' \quad \lambda \neq \surd}{S_1;S_2 \xrightarrow{\lambda} S_1';S_2} \qquad \frac{S_1 \xrightarrow{\surd} \mathbf{0} \quad S_2 \xrightarrow{\lambda} S_2'}{S_1;S_2 \xrightarrow{\lambda} S_2'}$$

$$\frac{S_1 \xrightarrow{a_*} S_1' \quad S_2 \xrightarrow{\overline{a}_*} S_2'}{S_1|S_2 \xrightarrow{\tau} S_1'|S_2'} \qquad \frac{S_1 \xrightarrow{\surd} \mathbf{0} \quad S_2 \xrightarrow{\surd} \mathbf{0}}{S_1|S_2 \xrightarrow{\surd} \mathbf{0}} \qquad \frac{S_1 \xrightarrow{\lambda} S_1' \quad \lambda \neq \surd}{S_1|S_2 \xrightarrow{\lambda} S_1'|S_2}$$

$$S_1^* \xrightarrow{\surd} \mathbf{0} \qquad\qquad \frac{S_1 \xrightarrow{\lambda} S_1' \quad \lambda \neq \surd}{S_1^* \xrightarrow{\lambda} S_1';S_1^*}$$

Table 3: Semantic rules for services (symmetric rules omitted).

are dealt with as for the Linda calculus of Section 2. In particular a_* and $\overline{a}_*$ synchronize over parallel as for CCS [28]. Given a (possibly empty) sequence of labels $w = \lambda_1\lambda_2 \cdots \lambda_{n-1}\lambda_n$, we use $S \xrightarrow{w} S'$ to denote a sequence of transitions $S \xrightarrow{\lambda_1} S_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_{n-1}} S_{n-1} \xrightarrow{\lambda_n} S'$ (in case of $w = \varepsilon$ we have $S' = S$, i.e., $S \xrightarrow{\varepsilon} S$).

As in [29, 33], the semantics of a service S yields a *finite-state* labeled transition system whose states are terms S' reachable from S, i.e. $\exists w : S \xrightarrow{w} S'$.

Example 3.1. (Authentication Server)

We now present a simple example of an authentication server that repeatedly performs two kinds of task: (i) the authentication of clients by receiving their username and password, and (ii) the request to an external account service for update of the list of the registered users.

$$\left(\begin{array}{l} username; password; (\overline{accepted}_{client} + \overline{failed}_{client}) + \\ \overline{updateAccounts}_{accountServer}; newAccounts \end{array}\right)^*$$

The service indicates a repeated choice between the two possible tasks. The first task is activated by the reception of an invocation on username. In this case, a password should subsequently be received and then two possible answers are sent back to the client: either accepted or failed. The second task is activated by sending a request for update to the accountServer. In this case, the newAccounts are subsequently received.

Example 3.2. (Travel Agency)

As a more involved example we consider, in Table 4, a travel agency service. Upon reception of a client's reservation request, the travel agency sends two

$$
\begin{array}{l}
Reservation; (\ \overline{ReserveFlight}_{airRes} \mid \overline{ReserveRoom}_{hotelRes}\ ); \big(\\
\quad (AvailFlights; (\overline{okFlight_*} + \overline{koFlight_*}) + NoFlights; \overline{koFlight_*}) \mid \\
\quad (NoRooms; \overline{ReserveRoom}_{hotelRes})^*; AvailRoom; \\
\qquad (okFlight_*; \overline{TravelPlan}_{client} + koFlight_*; \overline{NoAvail}_{client}) \\
\big)
\end{array}
$$

Table 4: Travel agency service.

parallel requests: to airplane and hotel reservation services. Then it spawns two internal processes that wait in parallel for the replies of each of the two reservation services. The first process, that waits for the reply of the airplain reservation service, internally communicates to the other process, using local names $okFlight_*$ and $koFlight_*$, whether a satisfactory flight is available: the $okFlight_*$ message is sent if a list of available fights is received and one of them is deemed to be satisfactory (based, e.g., on times/prices), otherwise $koFlight_*$ is used to explicitely communicate that no satisfactory flight has been found. The second process, that waits for the reply of the hotel reservation service, keeps actively querying such a service (e.g. trying with different hotels/nearby locations) until an available room is found; then it replies to the initial client's reservation request: based on the local message received from the other process either sends a travel plan or provides a negative answer.

3.2. Contract-based Service Discovery

We now define the notion of behavioural contract, as it appears in [34], which will allow us to reason about service compliance and retrieval *independently of the language* used for implementing service behaviour.

Contracts are defined as labeled transition systems, representing the communicating behavior of a service, standing at a certain location over the network, with respect to its environment (the other services). Thus, formally, as actions in their transition labels we just use τ for internal computations/synchronizations and a and $\overline{a}_l$ for inputs/outputs (where the outputs, as for the service calculus, are directed to a destination location $l \in Loc$). We first define the class of labeled transition systems of interest for defining contracts.

Definition 3.2. (Finite Connected LTS with Termination Transitions) *A finite connected labeled transition system (LTS) with termination transitions is a tuple $\mathcal{T} = (\mathcal{S}, \mathcal{L}, \longrightarrow, s_h, s_0)$ where $\mathcal{S}$ is a finite set of states, $\mathcal{L}$ is a set of labels, the transition relation $\longrightarrow$ is a finite subset of $(\mathcal{S} - \{s_h\}) \times (\mathcal{L} \cup \{\surd\}) \times \mathcal{S}$ such that $(s, \surd, s') \in \longrightarrow$ implies $s' = s_h$, s_h represents a halt state, $s_0 \in S$ represents the initial state, and it holds that every state in $\mathcal{S}$ is reachable (according to $\longrightarrow$) from s_0.*

As in the semantics of services, 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 intend to represent an *internal failure or a deadlock*.

Definition 3.3. (Behavioural Contracts) *A behavioural contract is a finite connected LTS with termination transitions, that is a tuple* $\mathcal{T} = (\mathcal{S}, \mathcal{L}, \longrightarrow, s_h, s_0)$*, where* $\mathcal{L} = \{a, \overline{a}_l, \tau \mid a \in \mathcal{N} \wedge l \in Loc\}$.

Services S *give rise to a behavioral contract as the labeled transition system obtained by their semantics*, where a_* and $\overline{a}_*$ labeled transitions (representing potential internal communications that did not actually take place) are disregarded. More precisely, the obtained behavioural contract is $(\mathcal{S}, \mathcal{L}, \longrightarrow, s_h, s_0)$ where: the initial state s_0 is S, the halt state s_h is $\mathbf{0}$, the set of states $\mathcal{S}$ includes, besides $\mathbf{0}$, the terms S' such that $\exists w : S \stackrel{w}{\longrightarrow} S' \wedge a_*, \overline{a}_* \notin w$ (i.e., both a_* and $\overline{a}_*$ do not belong to the label sequence w) and the transition relation $\longrightarrow$ is the set of trasitions $S' \stackrel{\lambda}{\longrightarrow} S''$ such that $S', S'' \in \mathcal{S}$ and $\lambda \notin \{a_*, \overline{a}_*\}$.

3.3. Behavioural Contracts as Terms

We now introduce a process algebraic representation for behavioural contracts by using basic CCS [35] over $\mathcal{L}$ prefixes extended with successful termination "$\mathbf{1}$", whereas the traditional null process "$\mathbf{0}$" is the halt state and denotes a *failure or a deadlock*, if not reached by a $\surd$ transition (see the discussion above about the halt state). Representing contracts as terms will be useful when developing their theory.

Definition 3.4. (Behavioural Contracts as Terms) *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 *denoting recursive definition of processes. We assume that in a contract* C *all process variables are bound. In the following we will omit trailing "*$.\mathbf{1}$*" when writing contracts.*[6]

Notice that, even if we represent contract LTSes as terms, we preserve their language independent nature (they are just a syntactical representation of LTSes).

The operational semantics of contracts C is defined in terms of a transition system labeled over $\{a, \overline{a}_l, \tau, \surd \mid a \in \mathcal{N}, l \in Loc\}$, ranged over by λ, λ', ..., obtained by the rules in Table 5 (plus a symmetric rule for choice). We use the notation $C\{\_/\_\}$ to denote syntactic replacement. Semantic rules are the standard ones, apart from that of term $\mathbf{1}$, which performs a $\surd$ transition denoting successful termination. We use $C \stackrel{w}{\longrightarrow} C'$ to denote a sequence of transitions

[6] We also omit parentheses under the assumption that prefix $\alpha.\_$ takes priority over choice $\_ + \_$ (similarly as we did for services).

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

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

Table 5: Semantic rules for contracts (symmetric rules omitted).

from C to C' labeled according to the label sequence w (formally the definition is the same as for service terms S).

The semantics of a behavioural contract term C gives rise to a finite[7] connected LTS with termination transitions $(\mathcal{S}, \mathcal{L}, \longrightarrow, \mathbf{0}, C)$, i.e. indeed a contract according to Definition 3.3, where $\mathcal{S}$ includes, besides $\mathbf{0}$, the set of states C' reachable from C, i.e. such that $\exists w : C \xrightarrow{w} C'$, and $\longrightarrow$ includes only transitions between states of $\mathcal{S}$. In [36] we formalize the correspondence between contracts $\mathcal{T} = (\mathcal{S}, \mathcal{L}, \longrightarrow, s_h, s_0)$ and their representation as terms C by showing how to obtain from a contract $\mathcal{T}$ a corresponding C with the same behaviour.

Example 3.3. (Authentication Server Contract)

We now present the contract for authentication service that we introduced in Example 3.1. In this simple case the contract is very similar to the service description itself:

$$recX.(\quad \overline{username.password.}(\overline{accepted}_{client}.X + \overline{failed}_{client}.X) + \overline{updateAccounts}_{accountServer}.newAccounts.X + \mathbf{1} \quad)$$

We just have that sequencing ";" is turned into prefix ".", that repetition via "$*$" is expressed by using recursion "$recX$" and that "1" is used to represent successful termination.

Example 3.4. (Travel Agency Contract)

We also present, in Table 6, the contract for the travel agency service presented in Example 3.2 (see Table 4). This example shows how contracts are just a syntactical representation of LTSes and abstract from the service language; in particular from the internal structure (e.g. internal processes) and internal communications of services: what in the service was internally communicated via $okFlight_*$ and $koFlight_*$ messages is now simply seen as a τ prefix. Notice that in Table 6, for clarity of presentation, parts of the the contract for the travel agency service have been separately defined, i.e. contracts: C_X, representing the status of the service after that the two initial concurrent requests have been sent;

[7] As for basic CCS [28] finite-stateness is an obvious consequence of the fact that the process algebra does not include static operators, like parallel or restriction.

$$
\begin{array}{l}
Reservation.(\\
\quad \overline{ReserveFlight}_{airRes}.\overline{ReserveRoom}_{hotelRes}.C_X + \\
\quad \overline{ReserveRoom}_{hotelRes}.\overline{ReserveFlight}_{airRes}.C_X \\
) \\
\\
C_X = recX.(\\
\quad AvailRoom.(\\
\quad\quad AvailFlights.(\tau.\overline{TravelPlan}_{client} + \tau.\overline{NoAvail}_{client}) + \\
\quad\quad NoFlights.\tau.\overline{NoAvail}_{client} \\
\quad\quad) + \\
\quad NoRooms.(\\
\quad\quad \overline{ReserveRoom}_{hotelRes}.X + \\
\quad\quad AvailFlights.\overline{ReserveRoom}_{hotelRes}.C_Y + \\
\quad\quad NoFlights.\overline{ReserveRoom}_{hotelRes}.C_Z \\
\quad\quad) + \\
\quad AvailFlights.C_Y + \\
\quad NoFlights.C_Z \\
) \\
\\
C_Y = recY.(\\
\quad AvailRoom.(\tau.\overline{TravelPlan}_{client} + \tau.\overline{NoAvail}_{client}) + \\
\quad NoRooms.\overline{ReserveRoom}_{hotelRes}.Y \\
) \\
\\
C_Z = recZ.(\\
\quad AvailRoom.\tau.\overline{NoAvail}_{client} + \\
\quad NoRooms.\overline{ReserveRoom}_{hotelRes}.Z \\
)
\end{array}
$$

Table 6: Travel agency contract (trailing "$\mathbf{1}$" are omitted).

C_Y, representing the status after that an "$AvailFlights$" message has been received, but no room as been found yet; and C_Z, representing the status after that a "$NoFlights$" message has been received, but no room has been found yet.

In order to show how a contract with $\mathbf{0}$ (representing failure or deadlock) is generated, we consider a variant of the travel agency service presented in Table 4, where we take

$$(okFlight_*; \overline{TravelPlan}_{client} + koFlight_*; \overline{NoAvail}_{client})$$

to be replaced by

$$okFlight_*; \overline{TravelPlan}_{client}$$

that is the service programmer "forgot" to consider the case in which the $koFlight_*$ message is internally transmitted. In this case, whenever a "$NoFlights$" message is received, the subsequent internal $\overline{koFlight_*}$ communication cannot be

completed and a deadlock is originated. The obtained contract is as that shown in Table 6, but for the two occurrences of

$$(\tau.\overline{TravelPlan}_{client} + \tau.\overline{NoAvail}_{client})$$

being replaced by

$$\tau.\overline{TravelPlan}_{client}$$

and the two occurrences of "$\tau.\overline{NoAvail}_{client}$*" being replaced by "***0***". The obtained contract, thus, shows that a deadlock is reached whenever the "NoFlights" message is received (from the flight reservation service).*

3.4. Output persistence property

In the following we will study independent contract refinement. As already anticipated in the Introduction under *synchronous* communication a maximal independent contract refinement that preserves compliance does not exist. In [11] we showed that this is a consequence of the symmetry between input and output actions and that a possible solution, for synchronous communication, is to resort to *output persistent* contracts; thus breaking such a symmetry.

Definition 3.5 (Output Persistence). *A contract C is output persistent if, for any C' such that $C \xrightarrow{w} C'$ and $C' \xrightarrow{\overline{a}_l}$, the following holds: $C' \not\xrightarrow{\lambda}$ and, if $C' \xrightarrow{\alpha} C''$ with $\alpha \neq \overline{a}_l$, then also $C'' \xrightarrow{\overline{a}_l}$.*

In the above definition we use $C \xrightarrow{\lambda}$ to mean $\exists C' : C \xrightarrow{\lambda} C'$.

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 holds in languages for the description of service orchestrations (see e.g. WS-BPEL [37]) 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).

Example 3.5. *It is easy to see that, while the travel agency contract of Example 3.4 is output persistent, the authentication server contract of Example 3.3 is not. However, the following modified version of the latter can be considered, to be used in a synchronous setting:*

$$\begin{array}{ll} recX.(& username.password.(\tau.\overline{accepted}_{client}.X + \tau.\overline{failed}_{client}.X) + \\ & \tau.\overline{updateAccounts}_{accountServer}.newAccounts.X + \mathbf{1} \quad) \end{array}$$

Notice that, in this new version of the authentication server, we have simply added an internal action τ in front of outputs occurring in choices. This guarantees that, at the moment the choice is to be resolved, the output action is not yet ready to be executed: it becomes available only after the τ and, then, its eventual execution is mandatory. As a consequence we have that the output persistence property is satisfied.

As observed in [29], a *syntactical condition on the service calculus of Definition 3.1*, which guarantees that output persistent contracts are always obtained, is: each output "$\overline{a}_l$" must always occur in sequence with a preceding "τ", i.e. *syntactically occur as "$\tau;\overline{a}_l$"*. For instance, if in the service of Example 3.1 we replace each output "$\overline{a}_l$" with "$\tau;\overline{a}_l$" we obtain the output persistent contract presented in the Example 3.5 above.[8]

In the remainder, when we consider synchronous communication, we will restrict to output persistent contracts.

3.5. Synchronous Contract Composition

Synchronous systems are formed by the parallel composition of contracts.

Definition 3.6 (Synchronous Systems). *The syntax of synchronous systems is defined by the following grammar*

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

We assume systems to be such that: (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.

A contract located at location l is denoted with $[C]_l$. Located contracts can be combined in parallel with the operator $P \| P$.

Example 3.6. *We consider a synchronous system in which we combine the output persistent contract for the authentication server presented in Example 3.5, here denoted with $C_{AuthServer}$ and located at "authServer" location, with a "client" contract and an "accountServer" contract, which interact, respectively, with the two tasks of the authentication server:*

$$\begin{array}{l}[C_{AuthServer}]_{authServer} \parallel \\ [\overline{username}_{authServer}.\overline{password}_{authServer}.(accepted + failed)]_{client} \parallel \\ [updateAccounts.\overline{newAccounts}_{authServer}]_{accountServer}\end{array}$$

System operational semantics is defined by the rules in Table 7 plus symmetric rules. Transition system labels, still ranged over by λ, λ', $\cdots$, are now taken from the set $\{\overline{a}_{sr}, a_{sr}, \tau, \surd \mid a \in \mathcal{N}; s, r \in Loc\}$, where: $\overline{a}_{sr}$ (a_{sr}, resp.) denotes a potential output (input, resp.) with the sender being at location s and the receiver at location r, τ denotes a synchronization or a move performed internally by one contract in the system and $\surd$ denotes successful termination.

[8] In [33] the same effect of requiring each output "$\overline{a}_l$" to syntactically occur as "$\tau;\overline{a}_l$" is obtained by modifying service operational semantics of Table 3 so that outputs are executed in two steps: the first step is a "τ" transition and the second step the actual "$\overline{a}_l$" transition.

$$\frac{C \xrightarrow{\overline{a}_r} C'}{[C]_s \xrightarrow{\overline{a}_{sr}} [C']_s} \qquad \frac{C \xrightarrow{a} C'}{[C]_r \xrightarrow{a_{sr}} [C']_r} \qquad \frac{C \xrightarrow{\lambda} C'}{[C]_l \xrightarrow{\lambda} [C']_l}\ \lambda \in \{\tau, \surd\}$$

$$\frac{P \xrightarrow{\lambda} P'}{P \| Q \xrightarrow{\lambda} P' \| Q}\ \lambda \neq \surd \qquad \frac{P \xrightarrow{\overline{a}_{sr}} P' \quad Q \xrightarrow{a_{sr}} Q'}{P \| Q \xrightarrow{\tau} P' \| Q'} \qquad \frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P \| Q \xrightarrow{\surd} P' \| Q'}$$

Table 7: Synchronous system semantics (symmetric rules omitted).

3.6. Asynchronous Contract Composition

In asynchronous systems contracts are equipped with an input message queue.

Definition 3.7 (Asynchronous Systems). *The syntax of asynchronous systems is defined by the following grammar*

$$\begin{array}{lcl} P & ::= & [C, \mathcal{Q}]_l \quad | \quad P \| P \\ \mathcal{Q} & ::= & \epsilon \quad | \quad a^l :: \mathcal{Q} \end{array}$$

We assume asynchronous systems to be such that: (i) and (ii) of Definition 3.6 (with $[C, \mathcal{Q}]_l$ replacing $[C]_l$) hold true.

Terms $\mathcal{Q}$ denote message queues. They are sequences of messages, each one denoted with a^l where a is the action name and l is the location of the sender. We use "ϵ" to denote the empty message queue. Trailing ϵ are usually left implicit, and we use "::" also as an operator over the syntax: if $\mathcal{Q}$ and $\mathcal{Q}'$ are ϵ-terminated queues, according to the syntax above, then $\mathcal{Q} :: \mathcal{Q}'$ means appending the two queues into a single ϵ-terminated list. Therefore, if $\mathcal{Q}$ is a queue, then $\epsilon :: \mathcal{Q}$, $\mathcal{Q} :: \epsilon$, and $\mathcal{Q}$ are syntactically equal. In the following, when we talk about asynchronous contract systems, we will use the shorthand $[C]_l$ to stand for $[C, \epsilon]_l$.

Example 3.7. *We consider an asynchronous system in which we combine the contract for the travel agency defined in Table 6, here denoted with $C_{TravelAgency}$ and located at "travelAgency" location, with a "client" contract, an airplane reservation contract "airRes" and an hotel reservation contract "hotelRes":*

$$\begin{array}{l} [\overline{Reservation}_{travelAgency}.(TravelPlan + NoAvail)]_{client} \ \| \\ [C_{TravelAgency}]_{travelAgency} \ \| \\ [ReserveFlight.(\overline{AvailFlights}_{travelAgency} + \overline{NoFlights}_{travelAgency})]_{airRes} \ \| \\ [recX.(ReserveRoom.(\overline{NoRooms}_{travelAgency}.X + \overline{AvailRoom}_{travelAgency}.X) \\ \qquad + \mathbf{1})]_{hotelRes} \end{array}$$

$$\frac{C \xrightarrow{\overline{a}_r} C'}{[C, \mathcal{Q}]_s \xrightarrow{\overline{a}_{sr}} [C', \mathcal{Q}]_s} \qquad [C, \mathcal{Q}]_r \xrightarrow{a_{sr}} [C, \mathcal{Q} :: a^s]_r \qquad \frac{C \xrightarrow{\surd} C'}{[C, \epsilon]_l \xrightarrow{\surd} [C', \epsilon]_l}$$

$$\frac{C \xrightarrow{\tau} C'}{[C, \mathcal{Q}]_s \xrightarrow{\tau} [C', \mathcal{Q}]_s} \qquad \frac{C \xrightarrow{a} C' \qquad b^l \in \mathcal{Q} \Rightarrow b \neq a}{[C, \mathcal{Q} :: a^s :: \mathcal{Q}']_r \xrightarrow{\tau} [C', \mathcal{Q} :: \mathcal{Q}']_r}$$

Table 8: Asynchronous system semantics (rules for parallel omitted).

Asynchronous system operational semantics is defined by the rules in Table 8 plus the rules for the parallel operator of Table 7. In Table 8 we assume that $b^l \in \mathcal{Q}$ holds true if and only if b^l syntactically occurs inside $\mathcal{Q}$. This notation is used in the premise of the novel τ synchronization rule that represents the consumption of an a message from the queue by removal of the oldest a one.

As an example consider the system: $[\overline{a}_s.\overline{b}_s]_r \parallel [b.a]_s$. After executing the two outputs, the system evolves to $[\mathbf{1}]_r \parallel [b.a, a^r :: b^r]_s$. The receiver is now ready to consume the two messages stored in the queue, thus reaching $[\mathbf{1}]_r \parallel [\mathbf{1}]_s$. Notice that the two messages are consumed in the opposite order of reception.

This means that the information about the sender attached to queue messages is actually not used by the operational semantic rules in Table 8: even if omitted we would have obtained the same transitions. Nevertheless, we decided to use the same queue syntax as in [13] to be more adherent to reality, where messages can be distinguished e.g. depending on the sender. As a matter of fact, in [13], this information is used to produce, instead of τ actions, more informative labels that include denotation of the sender-receiver (this makes it possible to establish conformance w.r.t. a given choreographical specification).

3.7. Contract Refinement

We now recall the formal definition of independent contract refinement that preserves correct composition of contracts in both the synchronous and asynchronous cases. With $P \xrightarrow{\tau}{}^* P'$ we denote the existence of a (possibly empty) sequence of τ-labeled transitions starting from the system P and leading to P'.

Definition 3.8 (Correct Contract Composition – Compliance). *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 $P' \xrightarrow{\tau}{}^* P''$ and $P'' \xrightarrow{\surd}$.*

Intuitively, a system composed of contracts is correct if any possible computation may guarantee completion, i.e. it can be extended to reach a successfully terminated computation (in the asynchronous case this means that all queues are empty). In this case, such contracts are called *compliant*. An example of contract composition that is correct (both in the synchronous and asynchronous

case) is $[\overline{a}_{l_3}]_{l_1} \parallel [\overline{b}_{l_3}]_{l_2} \parallel [a.b]_{l_3}$. Another example is $[\overline{a}_s.\overline{b}_s]_r \parallel [b.a]_s$ considered above, which is correct in the asychronous case only.

Example 3.8. *As larger examples, consider the sychronous and asychronous systems of Examples 3.6 and 3.7. While the latter is a correct contract composition, the former is surely not correct because the contract of the authServer can internally decide to invoke the update task twice, while the contract of the accountServer is able to reply only once. This problem is solved if we, instead, consider an accountServer with the following recursive contract:*

$$recX.(updateAccounts.\overline{newAccounts}_{authServer}.X + \mathbf{1})$$

After replacement of the accountServer contract with the one above, it is easy to see that the contract composition turns out to be correct: the system can always successfully complete after having performed (in every possible order) one authentication task and an arbitrary number of update tasks. A version in which also the authentication task can be performed arbitrarily many times can be obtained by considering a client with the following contract:

$$recX.(\overline{username}_{authServer}.\overline{password}_{authServer}.(accepted.X + failed.X) + \mathbf{1})$$

Notice that the discussion above is valid also under asychronous communication.

We are now ready to define the notion of *contract refinement*. Given a contract C, we use $oloc(C)$ to denote the set of locations used as destinations in all the output actions occurring inside C.

Definition 3.9 (Independent Refinement). *A pre-order $\leq$ over contracts is an independent refinement if, for any $n \geq 1$, contracts $C_1, \ldots, C_n$ and $C'_1, \ldots, C'_n$ such that $\forall i.\, C'_i \leq C_i$, and distinguished location names $l_1, \ldots, l_n \in Loc$ such that $\forall i.\, oloc(C_i) \cup oloc(C'_i) \subseteq \{l_j \mid 1 \leq j \leq n \wedge j \neq i\}$, we have:*

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

An independent refinement pre-order formalizes the possibility to replace in a correct contract composition every contract with one of its refinements, with the guarantee that the new system is still correct. In [11] it is shown that in the synchronous case, in the absence of the output persistence assumption, it could happen that given two independent refinement pre-orders, their union is no longer an independent refinement pre-order. In other words, there exists no maximal independent refinement pre-order.

On the contrary, if we restrict to output persistent contracts or we consider asynchronous communication, we have that the maximal independent refinement pre-order exists: it can be achieved by considering 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 others unchanged). This form of refinement, that we call *compliance testing* [38], is a form of testing where both the test and the system under test must reach success. Given a system P, we use $loc(P)$ to denote the subset of Loc of the locations of contracts syntactically occurring inside P.

Definition 3.10 (Refinement Relation). *A contract C' is a refinement of a contract C denoted $C' \preceq C$, if and only if for all $l \in Loc$ and system P such that $l \notin loc(P)$ and $l \notin oloc(C) \cup oloc(C') \subseteq loc(P)$, we have:*

$$([C]_l \| P)\downarrow \;\Rightarrow\; ([C']_l \| P)\downarrow$$

In the following, whenever $C' \preceq C$ we will also say that C' is a subcontract of C (or equivalently that C is a supercontract of C').

Theorem 3.1 (Maximal Independent Refinement). *There exists a maximal independent refinement $\leq$ pre-order and it corresponds to the (compliance testing based) refinement relation "$\preceq$".*

3.8. Properties of Contract Refinement

We now discuss some properties of contract refinement and also show a sound characterization that is decidable for the synchronous case. We use $I(C)$ ($O(C)$, resp.) to stand for the set of names a (located names a_l, resp.) of input actions a (output actions $\overline{a}_l$, resp.) syntactically occurring in C. Given $N \subseteq \{a_l \mid a \in \mathcal{N} \wedge l \in Loc\}$, we assume $\overline{N}$ to stand for $\{\overline{a}_l \mid a_l \in N\}$.

We first observe that the refinement relation $\preceq$ allows input on new names (and unreachable outputs on new names) to be added in refined contracts.

Theorem 3.2 (Refinements with Extended Inputs and Outputs). *Let C, C' be contracts. Both of the following hold*

$$\begin{array}{lcl} C'\{\mathbf{0}/\alpha.C'' \mid \alpha \in I(C') - I(C)\} \preceq C & \Leftrightarrow & C' \preceq C \\ C'\{T/\alpha.C'' \mid \alpha \in \overline{O(C') - O(C)}\} \preceq C & \Leftrightarrow & C' \preceq C \end{array}$$

where T is: $\mathbf{0}$ in the synchronous case, $\tau.\mathbf{0}$ in the asynchronous case.

This theorem is a direct consequence of queue based communication (in the asynchronous case) and output persistence (in the synchronous case): concerning its second statement, it follows from the fact that a subcontract C' cannot have reachable outputs that were not included in the potential outputs of the supercontract C; concerning its first statement, it follows from similarly observing that a compliant test P of a contract $[C]_l$ cannot have reachable outputs directed to l that C cannot receive (e.g. in the asynchronous case $[a]_l \| [\overline{a}_l + \overline{b}_l]_{l'}$ is not a correct contract composition). From the first statement of this theorem we can derive a fundamental property of the maximal independent refinement pre-order, which holds both in synchronous and asynchronous cases: external choices on inputs can be extended, e.g. $a + b \preceq a$. Concerning outputs, we instead have that $\overline{a}_l \preceq \overline{a}_l + \overline{b}_l$ in the asynchronous case only and $\tau.\overline{a}_l \preceq \tau.\overline{a}_l + \tau.\overline{b}_l$ in both synchronous and asynchronous cases, i.e. internal choices on outputs can be reduced. This because the lefthand term is more deterministic (typical property in testing).

Example 3.9. *As a larger example, consider the contract $C_{TravelAgency}$ in Table 6 of the travel agency service presented in Example 3.2 (see Table 4). A supercontract of $C_{TravelAgency}$ is obtained by considering the contract of a modified version of the travel agency service where we replace the line*

$$(NoRooms; \overline{ReserveRoom}_{hotelRes})^*; AvailRoom;$$

in Table 4 with just

$$AvailRoom;$$

The obtained contract C is a modification of that presented in Table 6 where in the definition of contracts C_X, C_Y and C_Z we just remove the whole branch of choice (i.e. "$+$") starting with the "$NoRooms$" input (in the case of C_Y and C_Z the choice itself is consequently removed). Since in such a contract C the "$NoRooms$" input no longer occurs, we have $NoRooms \in I(C_{TravelAgency}) - I(C)$, hence, according to Theorem 3.2 (first statement), $C_{TravelAgency} \preceq C$. Formally this derives from the fact that $C \preceq C$ and that the LTS of $C\{\mathbf{0}/\alpha.C' \mid \alpha \in I(C_{TravelAgency}) - I(C)\}$ is the same as that of $C_{TravelAgency}$.

We now focus on determining an algorithmic sound characterization of the synchronous contract refinement relation. This is achieved by resorting to the theory of fair testing, called *should-testing* [39].

Should-testing is a fair variant of must testing [40] where successful tests are defined as follows: process B passes test t iff every finite execution of $B \| t$ (the process subjected to the test, where "$\_\|\_$" denotes parallel execution of two processes, synchronized on all observable actions) has passed through or can be extended to pass through a successful state of t.

As a side result we also have that the refinement relation $\preceq$ is coarser than fair testing preorder. We denote with $\preceq_{test}$ the *should-testing* pre-order defined in [39] where we consider $\surd$ to be included in the set of actions of terms under testing as any other action ($\surd$ is treated as a normal action and not as the special action representing success of tests in [39]). In order to resort to the theory should-testing, we define a normal form for contracts C, denoted with $\mathcal{NF}(C)$, that corresponds to terms of the language in [39] (mainly a matter of replacing $\mathbf{1}$ with a $\surd$ action, see [29] for details).

Theorem 3.3 (Resorting to Fair Testing). *Let C, C' be contracts. We have*

$$\mathcal{NF}(C'\{\mathbf{0}/\alpha.C'' \mid \alpha \in I(C') - I(C)\}) \preceq_{test} \mathcal{NF}(C) \quad \Rightarrow \quad C' \preceq C$$

The opposite implication does not hold in general. This can be easily seen by considering *uncontrollable* contracts, i.e. contracts for which there is no compliant test. For instance the contract $\mathbf{0}$, any other contract $a.b.\mathbf{0}$ or $c.d.\mathbf{0}$ or more complex examples like $a + a.b$. These contracts are all equivalent according to our refinement relation, but of course not according to fair testing. Notice that such uncontrollable contracts have completely different traces: this means that trace pre-order is *not* coarser than our refinement relation.

4. Session Types

In this section we move to session types, in particular we report about our study of asynchronous session subtyping. Session types [41, 14] are types for controlling the communication behaviour of processes over channels. In a very simple but effective way, they express the pattern of sends and receives that a process must perform. They are, therefore, similar to behavioural contracts, but more constrained in the kind of behaviours they can express. Since they can guarantee freedom from some basic programming errors, session types are becoming popular with many main stream language implementations, e.g., Haskell [42], Go [43] or Rust [44]. In [16] session subtyping is introduced for asynchronous communication and it is also stated that it is decidable. Recently it has been proven that, on the contrary, it is undecidable. Here we present such an undecidability result [17] and the decidability result in [19], where the largest known decidable fragment is introduced. In particular, we recall the basic definitions of session types and synchronous and asynchronous session subtyping. We then report the undecidability proof in [17]. Finally, we present the fragment of *single-out* (and *single-in*) session types, for which we show asynchronous subtyping to be decidable [19]. The techniques for these (un)decidability results can be seen as improvements of those developed for Linda process calculi: reduction from Turing complete computational models and exploitation of well quasi orderings.

4.1. Session Subtyping

Session subtyping, which is the counterpart for session types of refinement for behavioural contracts, was first introduced by Gay and Hole [15] for a session-based π-calculus where communication is synchronous. Session subtyping of [15] is endowed with covariant/contravariant properties that correspond to those we observed on behavioural contract refinement: internal choices on outputs can be reduced, while external choices on inputs can be extended. To the best of our knowledge, Mostrous et al. [16] were the first to adapt the notion of session subtyping to an asynchronous setting. Their computation model is a session π-calculus with asynchronous communication that makes use of session queues for maintaining the order in which messages are sent. Based on such a model they introduce the idea of *output anticipation*, which is also relevant for our results in [17, 19] that we present here. Mostrous and Yoshida [45] extended the notion of asynchronous subtyping to session types for the higher-order π-calculus. They also observed that their definition of asynchronous subtyping allows for *orphan messages*, i.e. sent messages which are never consumed from the session queue. Orphan messages are, instead, prohibited with the definition of subtyping given by Chen et al. [46]: they show that such a definition is both sound and complete w.r.t. type safety and orphan message freedom.

We start with the formal syntax of binary session types, adopting a simplified notation (used, e.g., in [17, 19]) without dedicated constructs for sending an output/receiving an input. We instead represent outputs and inputs directly

inside choices. More precisely, we consider output selection $\oplus\{l_i : T_i\}_{i\in I}$, expressing an internal choice among outputs, and input branching $\&\{l_i : T_i\}_{i\in I}$, expressing an external choice among inputs. Each possible choice is labeled by a label l_i, taken from a global set of labels L, followed by a session continuation T_i. Labels in a branching/selection are assumed to be pairwise distinct.

Definition 4.1 (Session Types). *Given a set of labels L, ranged over by l, the syntax of binary session types is given by the following grammar:*

$$T \quad ::= \quad \oplus\{l_i : T_i\}_{i\in I} \quad | \quad \&\{l_i : T_i\}_{i\in I} \quad | \quad \mu\mathbf{t}.T \quad | \quad \mathbf{t} \quad | \quad \mathbf{end}$$

A session type is single-out *if, for all of its subterms $\oplus\{l_i : T_i\}_{i\in I}$, $|I| = 1$; it is* single-in *if, for all of its subterms $\&\{l_i : T_i\}_{i\in I}$, $|I| = 1$.*

In the sequel, we leave implicit the index set $i \in I$ in input branchings and output selections when it is already clear from the denotation of the types. Note also that we abstract from the type of the message that could be sent over the channel, since this is orthogonal to our theory. Types $\mu\mathbf{t}.T$ and $\mathbf{t}$ denote standard tail recursion for recursive types. We assume recursion to be guarded: in $\mu\mathbf{t}.T$, the recursion variable $\mathbf{t}$ occurs within the scope of an output or an input type. In the following, we will consider closed terms only, i.e., types with all recursion variables $\mathbf{t}$ occurring under the scope of a corresponding definition $\mu\mathbf{t}.T$. Type **end** denotes the type of a channel that can no longer be used.

For session types, we define the usual notion of duality: given a session type T, its dual $\overline{T}$ is defined as: $\overline{\oplus\{l_i : T_i\}_{i\in I}} = \&\{l_i : \overline{T}_i\}_{i\in I}$, $\overline{\&\{l_i : T_i\}_{i\in I}} = \oplus\{l_i : \overline{T}_i\}_{i\in I}$, $\overline{\mathbf{end}} = \mathbf{end}$, $\overline{\mathbf{t}} = \mathbf{t}$, and $\overline{\mu\mathbf{t}.T} = \mu\mathbf{t}.\overline{T}$. In the sequel, we say that a relation $\mathcal{R}$ on session types is *dual closed* if $(S,T) \in \mathcal{R}$ implies $(\overline{T},\overline{S}) \in \mathcal{R}$.

We start by considering a *synchronous* subtyping relation, similar to that of Gay and Hole [15] but, to be more consistent with contracts, following a process-oriented instead of a channel-based approach.[9] Moreover, following [16], we consider a generalized version of unfolding that allows us to unfold recursions $\mu\mathbf{t}.T$ as many times as needed.

Definition 4.2 (n-unfolding).

$$\begin{array}{ll}
\mathsf{unfold}^0(T) = T & \mathsf{unfold}^1(\oplus\{l_i : T_i\}_{i\in I}) = \oplus\{l_i : \mathsf{unfold}^1(T_i)\}_{i\in I} \\
\mathsf{unfold}^1(\mu\mathbf{t}.T) = T\{\mu\mathbf{t}.T/\mathbf{t}\} & \mathsf{unfold}^1(\&\{l_i : T_i\}_{i\in I}) = \&\{l_i : \mathsf{unfold}^1(T_i)\}_{i\in I} \\
\mathsf{unfold}^1(\mathbf{end}) = \mathbf{end} & \mathsf{unfold}^n(T) = \mathsf{unfold}^1(\mathsf{unfold}^{n-1}(T))
\end{array}$$

Definition 4.3 (Synchronous Subtyping, $\leq_s$). *$\mathcal{R}$ is a synchronous subtyping relation whenever $(T,S) \in \mathcal{R}$ implies that:*

1. *if $T = \mathbf{end}$ then $\exists n \geq 0$ such that $\mathsf{unfold}^n(S) = \mathbf{end}$;*
2. *if $T = \oplus\{l_i : T_i\}_{i\in I}$ then $\exists n \geq 0$ such that $\mathsf{unfold}^n(S) = \oplus\{l_j : S_j\}_{j\in J}$, $I \subseteq J$ and $\forall i \in I.\,(T_i, S_i) \in \mathcal{R}$;*

[9] Differently from our definitions, in the channel-based approach of Gay and Hole [15] subtyping is covariant on branchings and contra-variant on selections.

3. *if* $T = \&\{l_i : T_i\}_{i\in I}$ *then* $\exists n \geq 0$ *such that* $\mathsf{unfold}^n(S) = \&\{l_j : S_j\}_{j\in J}$, $J \subseteq I$ *and* $\forall j \in J.\, (T_j, S_j) \in \mathcal{R}$;
4. *if* $T = \mu \mathbf{t}.T'$ *then* $(T'\{T/\mathbf{t}\}, S) \in \mathcal{R}$.

T is a synchronous subtype of S, written $T \leq_s S$, if there is a synchronous subtyping relation $\mathcal{R}$ such that $(T, S) \in \mathcal{R}$.

Two types T and S are related by $\leq_s$, whenever S is able to simulate T with output and input types enjoying covariance and contravariance properties, respectively. Notice the asymmetric use of unfolding between the left- and right-hand terms T and S: in T recursion is always unfolded once, while in S many unfoldings can be needed in order to expose the starting operator of T.

As already discussed, subtyping is the counterpart of contract refinement in the context of session types. Consider, for instance,

$$\&\{a : \mathbf{end}, b : \mathbf{end}\} \leq_s \&\{a : \mathbf{end}\} \qquad \oplus\{a : \mathbf{end}\} \leq_s \oplus\{a : \mathbf{end}, b : \mathbf{end}\}$$

that hold for input contravariance and output covariance. These examples of subtypings precisely correspond to those of contract refinements commented in Section 3.8. Note that, while in the case of contracts they were obtained as a consequence of considering the maximal independent refinement, in the theory of session types they are taken by definition.

We now consider the standard notion of *asynchronous* subtyping $\leq$ introduced by Chen et al. [46], which enjoys orphan message freedom; we consider the simple rephrasing based on dual closeness we introduced in [19]. In the definition of $\leq$ we use the following notion of input context.

Definition 4.4 (Input Context). *An input context $\mathcal{A}$ is a session type with multiple holes defined by the syntax:*

$$\mathcal{A} \ ::= \quad [\,]^n \quad | \quad \&\{l_i : \mathcal{A}_i\}_{i\in I}$$

The holes $[\,]^n$, with $n \in \mathbb{N}^+$, of an input context $\mathcal{A}$ are assumed to be consistently enumerated, i.e. there exists $m \geq 1$ such that $\mathcal{A}$ includes one and only one $[\,]^n$ for each $n \leq m$. Given types $T_1, \ldots, T_m$, we use $\mathcal{A}[T_k]^{k\in\{1,\ldots,m\}}$ to denote the type obtained by filling each hole k in $\mathcal{A}$ with the corresponding term T_k.

Definition 4.5 (Asynchronous Subtyping, $\leq$). *$\mathcal{R}$ is an asynchronous subtyping relation whenever it is* dual closed *and $(T, S) \in \mathcal{R}$ implies* 1., 3., *and* 4. *of Definition 4.3, plus the following modified version of* 2.*:*

2. *if* $T = \oplus\{l_i : T_i\}_{i\in I}$ *then* $\exists n \geq 0, \mathcal{A}$ *such that*
   - $\mathsf{unfold}^n(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j\in J_k}]^{k\in\{1,\ldots,m\}}$,
   - $\forall k \in \{1, \ldots, m\}.\, I \subseteq J_k$ *and*
   - $\forall i \in I, (T_i, \mathcal{A}[S_{ki}]^{k\in\{1,\ldots,m\}}) \in \mathcal{R}$

T is an asynchronous subtype of S, written $T \leq S$, if there is an asynchronous subtyping relation $\mathcal{R}$ such that $(T, S) \in \mathcal{R}$.

We now explain the modified version of Rule 2. and its impact on the obtained subtyping relation. Concerning the adopted notation, for each hole k of the input context $\mathcal{A}$ (which is at the beginnig of the righthand term S after any needed unfolding), we take l_j, with $j \in J_k$, to be the labels of the output selection in the hole. Moreover, we use S_{kj} to denote the type reached after output l_j in the hole k. An important characteristic of asynchronous subtyping (formalized by Rule 2. above) is the following one. In a subtype output selections can be anticipated so to bring them before the input branchings that in the supertype occur in front of them. For example

$$\oplus\{l : \&\{l_1 : T_1, l_2 : T_2\}\} \;\leq\; \&\{l_1 : \oplus\{l : T_1\}, l_2 : \oplus\{l : T_2\}\}$$

where the output selection with label l (occurring in the supertype) is anticipated w.r.t. the input branching with labels l_1 and l_2 (such an output selection is present in *all* its input branches). As already discussed in the Introduction, output anticipation reflects the fact that we are considering asynchronous communication protocols in which messages are stored in queues. In this setting, it is safe to replace a peer that follows a given protocol with another one following a modified protocol where outputs are anticipated: in fact, the difference is simply that such outputs will be stored earlier in the communication queue.

As a further example, consider the types $T = \mu\mathbf{t}.\&\{l : \oplus\{l : \mathbf{t}\}\}$ and $S = \mu\mathbf{t}.\&\{l : \&\{l : \oplus\{l : \mathbf{t}\}\}\}$. We have $T \leq S$ by considering an infinite subtyping relation including pairs (T', S'), with S' being $\&\{l : S\}$, $\&\{l : \&\{l : S\}\}$, $\&\{l : \&\{l : \&\{l : S\}\}\}$, ...; that is, the effect of each output anticipation is that a new input $\&\{l : \_\}$ is accumulated in the initial part of the r.h.s. It is worth to observe that every accumulated input $\&\{l : \_\}$ is eventually consumed in the simulation game (orphan message freedom), but the accumulated inputs grows unboundedly.

4.2. Undecidability of Asynchronous Subtyping

Asynchronous session subtyping was considered decidable before Bravetti, Carbone and Zavattaro [17] and Lange and Yoshida [18] independently proved that it was undecidable. Here, we report the proof of undecidability in [17], which is by reduction from the acceptance problem for queue machines. Queue machines are a variant of Pushdown automata that exploits a queue instead of a stack. Differently from a stack, a queue allows the automata to access any symbol in memory without losing other symbols, that can be simply reenqueued. For this reason, Queue machines are Turing powerful while Pushdown automata are not.

Definition 4.6 (Queue machine). *A queue machine M is defined by a six-tuple $(Q, \Sigma, \Gamma, \$, s, \delta)$ where: Q is a finite set of states; $\Sigma \subset \Gamma$ is a finite set denoting the input alphabet; Γ is a finite set denoting the queue alphabet (ranged over by A, B, C, X); $\$ \in \Gamma - \Sigma$ is the initial queue symbol; $s \in Q$ is the start state; $\delta : Q \times \Gamma \to Q \times \Gamma^*$ is the transition function.*

We now formally define queue machine computations.

Definition 4.7 (Queue machine computation). *A* configuration *of a queue machine is an ordered pair (q, γ) where $q \in Q$ is its* current state *and $\gamma \in \Gamma^*$ is the* queue *(Γ^* is the Kleene closure of Γ). The starting configuration on an input string x is $(s, x\$)$. The transition relation $\rightarrow_M$ over configurations $Q \times \Gamma^*$, leading from a configuration to the next one, is defined as follows. For any $p, q \in Q$, $A \in \Gamma$ and $\alpha, \gamma \in \Gamma^*$ we have $(p, A\alpha) \rightarrow_M (q, \alpha\gamma)$ whenever $\delta(p, A) = (q, \gamma)$. A machine M accepts an input x if it eventually terminates on input x, i.e. it reaches a blocking configuration with the empty queue (notice that, as the transition relation is total, the unique way to terminate is by emptying the queue). Formally, x is accepted by M if $(s, x\$) \rightarrow_M^* (q, \epsilon)$ where ϵ is the empty string and $\rightarrow_M^*$ is the reflexive and transitive closure of $\rightarrow_M$.*

As observed above Queue machines are Turing complete, see [47] (page 354) and [17], hence it is undecidable whether a queue machine M accepts input x.

The undecidability of asynchronous session subtyping is proved as follows: given queue machine M with input x, construct a pair of types, say T and S, such that: $T \leq S$ if and only if x is not accepted by M. Given the undecidability of the acceptance problem for queue machines, we also have the undecidability of session subtyping.

Consider a queue machine M with input x. We will define a type T that encodes the finite control of M, i.e., its transition function δ, starting from its initial state s. We first concentrate on the definition of a type S that encodes the machine queue that initially contains $x\$$, where assume x to be the input string $x = X_1 \cdots X_n$ of length $n \geq 0$. Before presenting the formal definition of this type S, we discuss Figure 1 which contains a graphical representation of such type. Session types can be represented as labeled transition systems (in the form of communicating automata [48]), where an output selection $\oplus\{l_i : T_i\}_{i \in I}$ is represented as a choice among alternative output transitions labeled with "$l_i!$", and an input branching $\&\{l_i : T_i\}_{i \in I}$ is represented as a choice among alternative input transitions labeled with "$l_i?$".

In Figure 1 we report the graphical representation of the session type used to model the initial queue with content $X_1 \cdots X_n\$$. Such session type starts with $n + 1$ inputs, respectively on the labels $X_1 \cdots X_n\$$ (notice that the label alphabet for the session type corresponds with the queue alphabet Γ of the queue machine). After such sequence of inputs, there is a recursive type representing the capability to enqueue new symbols. Such a type repeatedly performs an output selection with one choice for each symbol A_i in the queue alphabet Γ (with k being the cardinality of Γ), followed by an input labeled with the same symbol A_i. In this way, assuming to play the subtyping simulation game, the effect of simulating an output on label A_i is that of adding a new input on label A_i at the end of the sequence of input on labels $X_1 \cdots X_n\$$. In other words, this coincides with enqueueing A_i as the new sequence of inputs $X_1 \cdots X_n\$A_i$ is obtained.

We are now ready to formally define the encoding of the queue of a machine into a session type. Notice that, in order to have a general definition, instead of considering the initial queue $X_1 \cdots X_n\$$ we consider a generic queue content

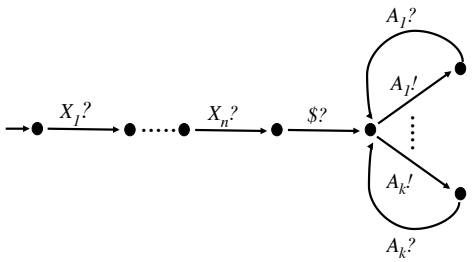

Figure 1: Session type encoding the initial queue $X_1 \cdots X_n\$$

$C_1 \cdots C_m$.

Definition 4.8 (Queue Encoding). *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine and let $C_1 \cdots C_m \in \Gamma^*$, with $m \geq 0$. We define:*

$$[\![C_1 \cdots C_m]\!] = \&\{C_1 : \ldots \&\{C_m : \mu \mathbf{t}. \oplus \{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma}\}\}$$

Given a configuration (q, γ) of M, the encoding of the queue $\gamma = C_1 \cdots C_m$ is thus defined as $[\![C_1 \cdots C_m]\!]$.

Note that whenever $m = 0$, we have $[\![\epsilon]\!] = \mu \mathbf{t}. \oplus \{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma}$. Observe that we are using a slight abuse of notation: in both output selections and input branchings, labels l_A, with $A \in \Gamma$, are simply denoted by A.

We now move to the encoding of the finite control of the queue machine into a session type. In Figure 2, we report a graphical representation of such type. We focus on the transition function δ, and a state $q \in Q$: let us consider $\delta(q, A_i) = (q_i, B_1^i \cdots B_{n_i}^i)$, with $n_i \geq 0$, for all i in $\{1, \ldots, k\}$. This particular state of the queue machine has a corresponding state $[\![q]\!]$ of the graphical representation of the session type. This state $[\![q]\!]$ performs an input branching with a choice for each symbol in the queue alphabet Γ (with k being the cardinality of Γ). Each of these choices represents a possible symbol that can be read from the queue. In fact, the idea is to play a subtyping simulation game where this type is the subtype, and the type encoding the queue machine (discussed above) is the supertype. The latter initially has an input on a label A_j representing the initial symbol of the queue. Hence, the unique branch to be considered in the simulation game is the one having such label A_j. After this initial selection, the continuation is composed of a sequence of outputs labeled with the symbols $B_1^j \cdots B_{n_j}^j$ that are expected to be inserted in the queue as the effect of consuming A_j from the queue. After these output actions have been considered in the simulation game (with the effect of enqueuing the corresponding symbols, see the discussion above about the encoding of the queue) state $[\![q_i]\!]$ of the session type, corresponding to the subsequent state q_i of the queue machine, is reached.

We are now ready to formally define the encoding of the finite control of the queue machine.

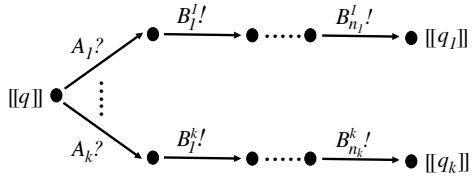

Figure 2: Session type encoding a finite control (for $\Gamma = \{A_i | i \leq k\}$ and $\delta(q, A_i) = (q_i, B_1^i \cdots B_{n_i}^i)$ for every i).

Definition 4.9 (Finite Control Encoding). *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine and let $q \in Q$ and $\mathcal{S} \subseteq Q$. We define:*

$$[\![q]\!]^{\mathcal{S}} = \begin{cases} \mu \mathbf{q}.\&\{A : \oplus\{B_1^A : \cdots \oplus \{B_{n_A}^A : [\![q']\!]^{\mathcal{S} \cup q}\}\}\}_{A \in \Gamma} \\ \qquad \text{if } q \notin \mathcal{S} \text{ and } \delta(q, A) = (q', B_1^A \cdots B_{n_A}^A) \\ \mathbf{q} \qquad \text{if } q \in \mathcal{S} \end{cases}$$

The encoding of the transition function of M is then defined as $[\![s]\!]^{\emptyset}$.

We finally state our undecidability result by means of the Theorem below which was proved in [17].

Theorem 4.1. *Given a queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$, an input string x, and the two types $T = [\![s]\!]^{\emptyset}$ and $S = [\![x\$]\!]$, we have that M accepts x iff $T \not\leq S$.*

4.3. Decidability of Single-Out/Single-In Asynchronous Subtyping

We have seen that the problem of checking asynchronous session subtyping is undecidable. Despite this negative result, several simplified cases have been considered in [17, 18] for which asynchronous session subtyping tunrs out to be decidable. These simplified cases are obtained by imposing limitations to the session type syntax. Here, we report the decidability result for the two fragments of single-out and single-in session types, which was proved in [19]. These fragments are more general than those presented in [17, 18].

We start by recalling a procedure (an algorithm that does not necessarily terminate) for the general subtyping relation, which we previously showed to be undecidable. We first introduce two functions on the syntax of types.

The function $\mathsf{outDepth}$ calculates how many unfoldings are necessary for bringing an output outside a recursion (in every possible input path). If that is not possible, the function is undefined (denoted by $\perp$). As an example consider, for any T_1 and T_2, $\mathsf{outDepth}(\oplus\{l_1 : T_1,\ l_2 : T_2\}) = 0$. On the other hand, consider the type $T_{ex} = \&\Big\{l_1 : \mu \mathbf{t}.\ \oplus\big\{l_2 : T_1\big\},\ l_3 : \mu \mathbf{t}.\&\big\{l_4 : \mu \mathbf{t}'.\ \oplus\{l_5 : T_2\}\big\}\Big\}$: we have, $\mathsf{outDepth}\big(T_{ex}\big) = 2$.

We then define $\mathsf{outUnf}()$, a variant of the unfolding function given in Definition 4.2, which unfolds only where it is necessary, in order to reach an output. The function above differs from unfold^n: for example, $\mathsf{unfold}^2\big(T_{ex}\big)$ would unfold

twice both subterms $\mu\mathbf{t}.\oplus\{l_2 : T_1\}$ and $\mu\mathbf{t}.\&\{l_4 : \mu\mathbf{t}'.\oplus\{l_5 : T_2\}\}$. On the other hand, applying $\mathsf{outDepth}$ to the same term would unfold once the term reached with l_1 and twice the one reached with l_3. In the subtyping procedure defined below we make use of $\mathsf{outUnf}()$ in order to have that recursive definitions under the scope of an output are never unfolded. This guarantees that during the execution of the procedure, even if the set of reached terms could be unbounded, all the subterms starting with an output are taken from a bounded set of terms. This is important to guarantee termination of the algorithm that we are going to define as an extension of the procedure described below.

In the definition of the procedure, we also make use of the notation $\& \in T$ to mean that T eventually reaches an input. Formally, $\& \in T$ if $T = \&\{l_i : T_i\}_{i\in I}$, or $T = \oplus\{l_i : T_i\}_{i\in I}$ with $\& \in T_i$ for all $i \in I$, or $T = \mu\mathbf{t}.T'$ with $\& \in T'$.

Subtyping Procedure. An environment Σ is a set containing pairs (T, S), where T and S are types. Judgements are triples of the form $\Sigma \vdash T \leq_{\mathsf{a}} S$ which intuitively read as "in order to succeed, the procedure must check whether T is a subtype of S, provided that pairs in Σ have already been visited". The *subtyping procedure*, applied to the types T and S, consists of deriving the state space of our judgments using the rules in Figure 3 bottom-up starting from the initial judgement $\emptyset \vdash T \leq_{\mathsf{a}} S$. More precisely, we use the transition relation $\Sigma \vdash T \leq_{\mathsf{a}} S \to \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ to indicate that if $\Sigma \vdash T \leq_{\mathsf{a}} S$ matches the conclusions of one of the rules in Figure 3, then $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is produced by the corresponding premises. The procedure explores the reachable judgements according to this transition relation. We give highest priority to rule Asmp, thus ensuring that at most one rule is applicable.[10] The idea behind Σ is to avoid cycles when dealing with recursive types. Rules RecR_1 and RecR_2 deal with the case in which the type on the right-hand side is a recursion and must be unfolded. If the type on the left-hand side is not an output then the procedure simply adds the current pair to Σ and continues. On the other hand, if an output must be found, we apply RecR_1 which checks whether such output is available. Rule Out allows nested outputs to be anticipated (when not under recursion) and condition $\left(\mathcal{A} \neq [\,]^1\right) \Rightarrow \forall i \in I.\& \in T_i$ (inspired by [46]) makes sure there are no orphan messages. In fact, this condition implies that if there is some output which is anticipated in the subtype w.r.t. some inputs, in every continuation of the subtype there are input actions that will eventually reproduce also the input behaviour of the supertype. The remaining rules are self-explanatory. $\Sigma \vdash T \leq_{\mathsf{a}} S \to^* \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is the reflexive and transitive closure of the transition relation among judgements. We write $\Sigma \vdash T \leq_{\mathsf{a}} S \to_{\mathsf{ok}}$ if the judgement $\Sigma \vdash T \leq_{\mathsf{a}} S$ matches the conclusion of one of the axioms Asmp or End, and $\Sigma \vdash T \leq_{\mathsf{a}} S \to_{\mathsf{err}}$ to mean that no rule can be applied to $\Sigma \vdash T \leq_{\mathsf{a}} S$. Due to input branching and output selection, the rules In and Out could generate branching also in the state space to be explored

[10] The priority of Asmp is sufficient because all the other rules are alternative, i.e., given a judgement $\Sigma \vdash T \leq_{\mathsf{a}} S$ there are no two rules different from Asmp that can be both applied.

$$\frac{(\mathcal{A} \neq [\,]^1) \Rightarrow \forall i \in I . \& \in T_i \qquad \forall n . I \subseteq J_n \qquad \forall i \in I \,.\, \Sigma \vdash T_i \leq_{\mathsf{a}} \mathcal{A}[S_{ni}]^n}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_{\mathsf{a}} \mathcal{A}[\oplus\{l_j : S_{nj}\}_{j \in J_n}]^n}\ \mathsf{Out}$$

$$\frac{J \subseteq I \qquad \forall j \in J \,.\, \Sigma \vdash T_j \leq_{\mathsf{a}} S_j}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \leq_{\mathsf{a}} \&\{l_j : S_j\}_{j \in J}}\ \mathsf{In} \qquad \frac{}{\Sigma \vdash \mathbf{end} \leq_{\mathsf{a}} \mathbf{end}}\ \mathsf{End}$$

$$\frac{}{\Sigma, (T, S) \vdash T \leq_{\mathsf{a}} S}\ \mathsf{Asmp} \qquad \frac{\Sigma, (\mu \mathbf{t}.T, S) \vdash T\{\mu \mathbf{t}.T/\mathbf{t}\} \leq_{\mathsf{a}} S}{\Sigma \vdash \mu \mathbf{t}.T \leq_{\mathsf{a}} S}\ \mathsf{RecL}$$

$$\frac{T = \mathbf{end} \vee T = \&\{l_i : T_i\}_{i \in I} \qquad \Sigma, (T, \mu \mathbf{t}.S) \vdash T \leq_{\mathsf{a}} S\{\mu \mathbf{t}.S/\mathbf{t}\}}{\Sigma \vdash T \leq_{\mathsf{a}} \mu \mathbf{t}.S}\ \mathsf{RecR}_1$$

$$\frac{\mathsf{outDepth}(S) \geq 1 \qquad \Sigma, (\oplus\{l_i : T_i\}_{i \in I}, S) \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_{\mathsf{a}} \mathsf{outUnf}(S)}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \leq_{\mathsf{a}} S}\ \mathsf{RecR}_2$$

Figure 3: A Procedure for Checking Subtyping

by the procedure. Namely, given a judgement $\Sigma \vdash T \leq_{\mathsf{a}} S$, there are several subsequent judgements $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ such that $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow \Sigma' \vdash T' \leq_{\mathsf{a}} S'$. The procedure could (i) successfully terminate because all the explored branches reach a successful judgement $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{ok}}$, (ii) terminate with an error in case at least one judgement $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$ is reached, or (iii) diverge because no branch terminates with an error and at least one branch never reaches a succesful judgement. As we prove in [19] the procedure is sound with respect to asynchronous subtyping $\leq$ and it can diverge only if the checked types are in the $\leq$ relation.

If we consider types T and S of the example considered after Definition 4.5 the subtyping procedure in Figure 3 applied to $\emptyset \vdash T \leq_{\mathsf{a}} S$ does not terminate. The problem is that the termination rule Asmp cannot be applied because the term on the r.h.s. (i.e. the supertype) generates always new terms in the form $\&\{l : \&\{l : \ldots \&\{l : S\} \ldots\}\}$. Notice that, in this particular example, these infinitely many distinct terms are obtained by adding single inputs (i.e. single-choice input branchings) in front of the term in the r.h.s.: we call this *linear input accumulation*. In general, however, input accumulation takes the *form of a tree* (thus accounting for all possible alternative accumulated input behaviors at the same time).

We now show how to decide asynchronous subtyping over single-out types, i.e. when input accumulation can indeed be in the general form of a tree, but, due to the absence of output selections with multiple choices, it gets accumulated in a deterministic (i.e. unique) way. This will also allow us to deal with single-in types by exploiting duality. As anticipated, it deals with general input accumulation by representing it as a tree. We need to be able to extract the leaves from

these trees: this is done by the *leaf set* function. The leaf set of a session type T is the set of subterms reachable from its root through a path of inputs. For example, the leaf set of the term $\&\{l_1 : \mu\mathbf{t}.\oplus\{l_2 : \mathbf{t}\}, l_3 : \&\{l_4 : \oplus\{l_2 : \mu\mathbf{t}.\oplus\{l_2 : \mathbf{t}\}\}\}\}$ is $\{\mu\mathbf{t}.\oplus\{l_2 : \mathbf{t}\}, \oplus\{l_2 : \mu\mathbf{t}.\oplus\{l_2 : \mathbf{t}\}\}$.

During the check of subtyping, according to Figure 3 (rule Out), when a term in the r.h.s. having input accumulation has to mimic an output in front of the l.h.s., such output must be present in front of all the leaves of the tree. In this case, the checking continues by anticipating the output from all the leaves. We make use of an auxiliary *output anticipation* function, called antOut, that indicates the way a term changes after having anticipated a sequence of outputs. $\mathsf{antOut}(T, \tilde{l})$ yields the term obtained from T by anticipating all outputs occurring in the sequence $\tilde{l}$. For example, the function applied to the type $T = \mu\mathbf{t}.\oplus\left\{l_1 : \&\{l : \oplus\{l_2 : \mathbf{t}\}, l' : \oplus\{l_2 : \mathbf{t}\}\right\}$ and the sequence (l_1, l_2) returns $\&\{l : T, l' : T\}$, while it is undefined with the sequence (l_1, l_1). Moreover, we say that T can infinitely anticipate outputs, written $\mathsf{antOutInf}(T)$, if there exists an infinite sequence of labels $l_{i_1} \cdots l_{i_j} \cdots$ such that $\mathsf{antOut}(T, l_{i_1} \cdots l_{i_n})$ is defined for every n. The definition of $\mathsf{antOutInf}(T)$ is not algorithmic in that it quantifies on every possible natural number n. Nevertheless, it can be decided by checking whether, for every session type obtained from T by means of output anticipations, all the terms populating its leaf set can anticipate the same output label. Although the types that can be obtained from T by means of output anticipations may be infinite, the terms populating the leaf sets are finite and are over-approximated by the function $\mathsf{reach}(T)$ which is defined as the minimal set of (single-out) session types such that:

1. $T \in \mathsf{reach}(T)$;
2. $\&\{l_i : T_i\}_{i\in I} \in \mathsf{reach}(T)$ implies $T_i \in \mathsf{reach}(T)$ for every $i \in I$;
3. $\mu\mathbf{t}.T' \in \mathsf{reach}(T)$ implies $T'\{\mu\mathbf{t}.T'/\mathbf{t}\} \in \mathsf{reach}(T)$;
4. $\oplus\{l : T'\} \in \mathsf{reach}(T)$ implies $T' \in \mathsf{reach}(T)$.

Notice that $\mathsf{reach}(T)$ contains the session types obtained by consuming initial inputs and outputs, and by unfolding recursion when it is at the top level.

We now recall the decidability of $\mathsf{antOutInf}(T)$ and the finiteness of $\mathsf{reach}(T)$ that were proved in [19], and that will be used in the next subsection.

Proposition 4.1. *Given a single-out session type T, $\mathsf{reach}(T)$ is finite and it is decidable whether $\mathsf{antOutInf}(T)$.*

Subtyping algorithm for single-out types. We are now ready to present an additional termination condition that, once included into the subtyping procedure in Figure 3, makes it a valid algorithm for checking subtyping for single-out types. The termination condition is defined as an additional rule, named $\mathsf{Asmp2}$, that complements the already defined Asmp rule by detecting those cases in which the subtyping procedure in Figure 3 does not terminate ($\mathsf{Asmp2}$, presented below, is assumed to have the same priority as rule Asmp: both rules have highest priority). The new rule is defined parametrically on the session type Z, which is the type on the right-hand side of the initial pair of types to be checked (i.e.

the algorithm is intended to check $V \leq Z$, for some type Z). We start from the initial judgement $\emptyset \vdash V \leq_{\mathsf{t}} Z$ and then apply from bottom to top the rules in Figure 3, where $\leq_{\mathsf{a}}$ is replaced by $\leq_{\mathsf{t}}$, plus the following additional rule:

$$\frac{S \in \mathsf{reach}(Z) \qquad \mathsf{antOutInf}(S) \qquad |\gamma| < |\beta| \qquad \mathsf{leafSet}(\mathsf{antOut}(S,\gamma)) = \mathsf{leafSet}(\mathsf{antOut}(S,\beta))}{\Sigma, (T, \mathsf{antOut}(S,\gamma)) \vdash T \leq_{\mathsf{t}} \mathsf{antOut}(S,\beta)} \ \mathsf{Asmp2}$$

Intuitively, we have that this additional termination rule guarantees to catch all those cases where the term on the right grows indefinitely, by anticipating outputs and accumulating inputs. These infinitely many distinct types are anyway obtainable starting from the finite set $\mathsf{reach}(Z)$, by means of output anticipations. Hence there exists $S \in \mathsf{reach}(Z)$ that can generate infinitely many of these types: this guarantees $\mathsf{antOutInf}(S)$ to be true. As observed above, the leaves of such infinitely many terms are themselves taken from the finite set $\mathsf{reach}(Z)$; hence the leaf sets are always taken from the finite set of subsets of $\mathsf{reach}(Z)$ (which is itself finite, see Proposition 4.1). In Section 2 we have recalled the notion of wqo, and that equality on a finite set is a wqo (i.e., in an infinite sequence of elements taken from a finite set, there are at least to equal elements). The termination of our algorithm follows from this wqo: Asmp2, besides checking conditions that are guaranteed to hold if the procedure $\leq_{\mathsf{a}}$ continues indefinitely, checks for the equality between the set of leaves of the r.h.s. term in the current judgement and the r.h.s. in a previously checked one. This is guaranteed to eventually happen as the set of leaves are taken from a finite domain, and equality is a wqo on this finite domain.

We are finally ready to formally state the decidability results for single-out/ single-in session types proved in [19]. We start with the single-out case, for which we have recalled the corresponding deciding algorithm.

Theorem 4.2 (Decidability for Single-out Types). *Asynchronous subtyping $\leq$ over single-out session types is decidable.*

Exploiting dual closeness of $\leq$, the algorithm presented for single-out types can be applied also to single-in types (it is sufficient to check subtyping on the duals, observing that the dual of a single-in type is single-out).

Corollary 4.1 (Decidability for Single-in Types). *Asynchronous subtyping $\leq$ over single-in session types is decidable.*

5. Conclusion

In this paper we have described the contribution of our research group in the context of three areas of interest for the Coordination conference and its research community: shared dataspace coordination languages, behavioural contracts for service composition, and session types. The common denominator of the presented research is the exploitation of techniques borrowed from concurrency theory, in particular, process algebras and Petri nets.

The paper is not intended to be a survey on the three considered areas of research; survey papers already exists for coordination languages [49], for the application of process algebras in the context of shared dataspace coordination [50], and for behavioural types [51, 52], that include both contracts and session types. On the contrary, the objective of the paper is to present, following an original form of presentation, techniques developed during the early years of formal study of coordination languages, that more recently turned out to be successfully applicable also in other contexts.

More precisely, we started by presenting a novel Linda-based process calculus based on the exchange of tuples of names through a shared dataspace. The distinguishing feature of this calculus is that new names can be produced and passed to other processes simply by placing them inside tuples. For this calculus we have considered the *reachability* and *coverability* problems, two properties dealing with the exploration of the reachable states: given a target dataspace, reachability consists of checking whether such dataspace can be reached, while for coverability it is sufficient to reach a state that includes at least the tuples in the target dataspace. By using techniques that we already applied to other shared dataspace process calculi, we have proved that (i) both properties are undecidable in the proposed calculus, (ii) coverability becomes decidable if we assume that processes can use the new names that they receive only inside output operations, while (iii) reachability turns out to be decidable if we remove from the calculus the possibility to generate new names. For undecidability proofs we resort to Random Access Machines, a register based Turing complete formalism, while for the decidability results we exploited Petri nets and Well Structured Transition Systems, that are transition systems equipped with a well quasi ordering on states.

In the second part of the paper we recall the main results in the context of behavioural contracts applied to service oriented computing. We present a language for services having essentially the same syntax of the Linda calculus introduced in the first part of the paper, but with a different semantics: processes (called services in this case) do not share a common message repository, but send messages directly to operations offered by other services. On this language we apply our contract theories by considering both synchronous and asynchronous communication. Contracts describe the input-output communication behaviour of services and can be used to reason about service composition and and service replaceability; formally, we say that a service can be used in place of another one if their behavioural contracts are in *refinement* relation. One of the open problems of behavioural contracts is an algorithmic characterization of refinement in the case of asynchronous communication.

We have presented behavioural contracts because the interest in trying to close the above open problem on contract refinement for asynchronous communication was the main justification for moving to the study of session types, discussed in the third part of this paper. Session types can be seen as a simplification of contracts that already had, besides a rich and well-established theory, a wide application on concurrent programming languages such as, e.g., Haskell [42], Go [43] and Rust [44]. We have specifically focused on session *sub-*

typing which is, for session types, a notion equivalent to the notion of refinement in the context of contracts. We have recalled the well-known algorithm by Gay and Hole [15] for synchronous session subtyping. For asynchronous communication, on the other hand, algorithms that were presented, like that in [16], revealed wrong: in fact, it was proved that asynchronous session subtyping. We have reported here our undecidability proof that, like those for Linda-like process calculi, is by reduction from the halting problem in Turing complete formalism. In this case we used Queue Machines instead of Random Access Machines. Another recent contribution that we have reported here is an algorithm for checking asynchronous session subtyping under the assumption that output (or input) choices have only one branch. Our algorithm is inspired by the one by Gay and Hole, but in order to deal with asynchronous communication we need a more sophisticated termination condition that must check also messages that have been buffered in the communication channels. The termination of this algorithm is proved by exploiting techniques like those used for the decidability proof in Linda-like calculi, namely, the application of well quasi orderings.

We expect two possible lines for future work: on the one hand, analyse the impact on the theory of contracts of our results for session types (in fact, very few results are present in the literature about contracts for asynchronous communication) and, on the other hand, continue in the context of session types by investigating novel techniques for sound algorithmic characterizations of asynchronous session subtyping.

References

References

[1] D. Gelernter, Generative communication in linda, ACM Trans. Program. Lang. Syst. 7 (1) (1985) 80–112.

[2] N. Carriero, D. Gelernter, The s/net's linda kernel (extended abstract), in: Proc. of 10th ACM Symposium on Operating System Principles, SOSP'85, ACM, 1985, p. 160.

[3] R. De Nicola, R. Pugliese, A process algebra based on LINDA, in: Proc. of 1st Int. Conference on Coordination Languages and Models, COORDINATION'96, Vol. 1061 of Lecture Notes in Computer Science, Springer, 1996, pp. 160–178.

[4] N. Busi, R. Gorrieri, G. Zavattaro, Three semantics of the output operation for generative communication, in: Proc. of 2nd Int. Conference on Coordination Languages and Models, COORDINATION'97, Vol. 1282 of Lecture Notes in Computer Science, Springer, 1997, pp. 205–219.

[5] N. Busi, R. Gorrieri, G. Zavattaro, On the Turing equivalence of Linda coordination primitives, in: Proc. of 4th Workshop on Expressiveness in Concurrency, EXPRESS'97, Vol. 7 of Electr. Notes Theor. Comput. Sci., Elsevier, 1997.

[6] C. Dufourd, A. Finkel, P. Schnoebelen, Reset nets between decidability and undecidability, in: Proc. of 25th Int. Colloquium on Automata, Languages and Programming, ICALP'98, Vol. 1443 of Lecture Notes in Computer Science, Springer, 1998, pp. 103–115.

[7] N. Busi, G. Zavattaro, On the expressiveness of event notification in data-driven coordination languages, in: Proc. of 9th European Symposium on Programming, ESOP'00, Vol. 1782 of Lecture Notes in Computer Science, Springer, 2000, pp. 41–55.

[8] N. Busi, R. Gorrieri, G. Zavattaro, Temporary data in shared dataspace coordination languages, in: Proc. of 4th Int. Conference on Foundations of Software Science and Computation Structures, FOSSACS'01, Vol. 2030 of Lecture Notes in Computer Science, Springer, 2001, pp. 121–136.

[9] M. Bravetti, S. Gilmore, C. Guidi, M. Tribastone, Replicating web services for scalability, in: TGC, Vol. 4912 of LNCS, Springer, 2008, pp. 204–221.

[10] S. Carpineti, G. Castagna, C. Laneve, L. Padovani, A formal account of contracts for web services, in: Proc. of 3rd Int. Workshop on Web Services and Formal Methods, WS-FM'06, Vol. 4184 of Lecture Notes in Computer Science, Springer, 2006, pp. 148–162.

[11] M. Bravetti, G. Zavattaro, Contract based multi-party service composition, in: Proc. of Int. Symposium on Fundamentals of Software Engineering, FSEN'07, Vol. 4767 of Lecture Notes in Computer Science, Springer, 2007, pp. 207–222.

[12] M. Bravetti, G. Zavattaro, A theory for strong service compliance, in: Proc. of 9th Int. Conference on Coordination Models and Languages, COORDINATION'07, Vol. 4467 of Lecture Notes in Computer Science, Springer, 2007, pp. 96–112.

[13] M. Bravetti, G. Zavattaro, Contract compliance and choreography conformance in the presence of message queues, in: Proc. of 5th Int. Workshop on Web Services and Formal Methods, WS-FM'08, Vol. 5387 of Lecture Notes in Computer Science, Springer, 2008, pp. 37–54.

[14] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: Proc. of 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'08, ACM, 2008, pp. 273–284. doi:10.1145/1328438.1328472.

[15] S. J. Gay, M. Hole, Subtyping for session types in the pi calculus, Acta Inf. 42 (2-3) (2005) 191–225. doi:10.1007/s00236-005-0177-z.

[16] D. Mostrous, N. Yoshida, K. Honda, Global principal typing in partially commutative asynchronous sessions, in: Proc. of 18th European Symposium on Programming, ESOP'09, Vol. 5502 of Lecture Notes in Computer

Science, Springer, 2009, pp. 316–332. doi:10.1007/978-3-642-00590-9_23.

[17] M. Bravetti, M. Carbone, G. Zavattaro, Undecidability of asynchronous session subtyping, Inf. Comput. 256 (2017) 300–320.

[18] J. Lange, N. Yoshida, On the undecidability of asynchronous session subtyping, in: Proc. of 20th Int. Conference on Foundations of Software Science and Computation Structures, FOSSACS'17, Vol. 10203 of Lecture Notes in Computer Science, 2017, pp. 441–457.

[19] M. Bravetti, M. Carbone, G. Zavattaro, On the boundary between decidability and undecidability of asynchronous session subtyping, Theor. Comput. Sci. 722 (2018) 19–51.

[20] R. Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes, I/II, Inf. Comput. 100 (1) (1992).

[21] E. W. Mayr, An algorithm for the general petri net reachability problem, in: Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA, ACM, 1981, pp. 238–246.

[22] C. Rackoff, The covering and boundedness problems for vector addition systems, Theor. Comput. Sci. 6 (1978) 223–231. doi:10.1016/0304-3975(78)90036-1.
URL http://dx.doi.org/10.1016/0304-3975(78)90036-1

[23] J. C. Shepherdson, H. E. Sturgis, Computability of recursive functions, J. ACM 10 (2) (1963) 217–255.

[24] M. L. Minsky, Computation: finite and infinite machines, Prentice-Hall, Inc., 1967.

[25] A. Finkel, P. Schnoebelen, Well-structured transition systems everywhere!, Theor. Comput. Sci. 256 (1-2) (2001) 63–92.

[26] N. Busi, G. Zavattaro, On the expressiveness of movement in pure mobile ambients, Electr. Notes Theor. Comput. Sci. 66 (3) (2002) 22–36. doi:10.1016/S1571-0661(04)80414-6.

[27] C. Fournet, C. A. R. Hoare, S. K. Rajamani, J. Rehof, Stuck-free conformance, in: Proc. of 16th International Conference on Computer Aided Verification, CAV'04, Vol. 3114 of Lecture Notes in Computer Science, Springer, 2004, pp. 242–254. doi:10.1007/978-3-540-27813-9_19.

[28] R. Milner, Communication and concurrency, Prentice Hall, 1989.

[29] M. Bravetti, G. Zavattaro, Towards a unifying theory for choreography conformance and contract compliance, in: Proc. of 6th Int. Symposium Software Composition, SC'07, Vol. 4829 of Lecture Notes in Computer Science, Springer, 2007, pp. 34–50.

[30] C. Laneve, L. Padovani, The *Must* preorder revisited, in: Proc. of 18th Int. Conference Concurrency Theory, CONCUR'07, Vol. 4703 of Lecture Notes in Computer Science, Springer, 2007, pp. 212–225. doi:10.1007/978-3-540-74407-8_15.

[31] G. Castagna, N. Gesbert, L. Padovani, A theory of contracts for web services, in: Proc. of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'08, ACM, 2008, pp. 261–272. doi:10.1145/1328438.1328471.

[32] F. Barbanera, U. de'Liguoro, Two notions of sub-behaviour for session-based client/server systems, in: Proc. of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'10, ACM, 2010, pp. 155–164.

[33] M. Bravetti, G. Zavattaro, A Foundational Theory of Contracts for Multi-party Service Composition, Fundamenta Informaticae 89 (4) (2008) 451–478.

[34] M. Boreale, M. Bravetti, Advanced mechanisms for service composition, query and discovery, in: Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing, Vol. 6582 of Lecture Notes in Computer Science, Springer, 2011, pp. 282–301.

[35] R. Milner, A complete axiomatisation for observational congruence of finite-state behaviors, Inf. Comput. 81 (2) (1989) 227–247. doi:10.1016/0890-5401(89)90070-9.
URL https://doi.org/10.1016/0890-5401(89)90070-9

[36] M. Bravetti, G. Zavattaro, Contract-based discovery and composition of web services, in: Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Vol. 5569, Springer, 2009, pp. 261–295. doi:10.1007/978-3-642-01918-0_7.

[37] OASIS, Web Services Business Process Execution Language Version 2.0 OASIS Standard, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

[38] M. Bravetti, G. Zavattaro, Choreographies and behavioural contracts on the way to dynamic updates, in: Proc. 1st Workshop on Logics and Model-checking for Self-* Systems, MOD*'14, Vol. 168 of EPTCS, 2014, pp. 12–31.

[39] A. Rensink, W. Vogler, Fair testing, Inf. Comput. 205 (2) (2007) 125–198. doi:10.1016/j.ic.2006.06.002.

[40] R. De Nicola, M. Hennessy, Testing equivalences for processes, Theor. Comput. Sci. 34 (1984) 83–133. doi:10.1016/0304-3975(84)90113-0.

[41] K. Honda, V. T. Vasconcelos, M. Kubo, Language primitives and type discipline for structured communication-based programming, in: Proc. of 7th European Symposium on Programming, ESOP'98, Vol. 1381 of Lecture Notes in Computer Science, Springer, 1998, pp. 122–138. doi:10.1007/BFb0053567.

[42] S. Lindley, J. G. Morris, Embedding session types in haskell, in: Proc. of 9th International Symposium on Haskell, Haskell'16, 2016, pp. 133–145. doi:10.1145/2976002.2976018.

[43] N. Ng, N. Yoshida, Static deadlock detection for concurrent go by global session graph synthesis, in: Proc. of 25th International Conference on Compiler Construction, CC'16, 2016, pp. 174–184. doi:10.1145/2892208.2892232.

[44] T. B. L. Jespersen, P. Munksgaard, K. F. Larsen, Session types for Rust, in: Proc. of 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP'15, 2015, pp. 13–22. doi:10.1145/2808098.2808100.

[45] D. Mostrous, N. Yoshida, Session typing and asynchronous subtyping for the higher-order π-calculus, Inf. Comput. 241 (2015) 227–263. doi:10.1016/j.ic.2015.02.002.

[46] T. Chen, M. Dezani-Ciancaglini, A. Scalas, N. Yoshida, On the preciseness of subtyping in session types, Logical Methods in Computer Science 13 (2) (2017).

[47] D. Kozen, Automata and computability, Springer, New York, 1997.

[48] D. Brand, P. Zafiropulo, On communicating finite-state machines, J. ACM 30 (2) (1983) 323–342.

[49] G. A. Papadopoulos, F. Arbab, Coordination models and languages, Advances in Computers 46 (1998) 329–400. doi:10.1016/S0065-2458(08)60208-9.
URL https://doi.org/10.1016/S0065-2458(08)60208-9

[50] N. Busi, G. Zavattaro, A process algebraic view of shared dataspace coordination, J. Log. Algebr. Program. 75 (1) (2008) 52–85. doi:10.1016/j.jlap.2007.06.003.
URL https://doi.org/10.1016/j.jlap.2007.06.003

[51] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, N. Yoshida, Behavioral types in programming languages, Foundations and Trends in Programming Languages 3 (2-3) (2016) 95–230. doi:10.1561/2500000031.
URL https://doi.org/10.1561/2500000031

[52] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, G. Zavattaro, Foundations of session types and behavioural contracts, ACM Comput. Surv. 49 (1) (2016) 3:1–3:36. doi:10.1145/2873052.
URL https://doi.org/10.1145/2873052