Relating Session Types and Behavioural Contracts: The Asynchronous Case

Relating Session Types and Behavioural Contracts: the Asynchronous Case*

Mario Bravetti Gianluigi Zavattaro

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

Abstract. We discuss the relationship between session types and behavioural contracts under the assumption that processes communicate asynchronously. We show the existence of a fully abstract interpretation of session types into a fragment of contracts, that maps session subtyping into binary compliance-preserving contract refinement. In this way, the recent undecidability result for asynchronous session subtyping can be used to obtain an original undecidability result for asynchronous contract refinement.

1 Introduction

Session types are used to specify the structure of communication between the endpoints of a distributed system or the processes of a concurrent program. In recent years, session types have been integrated into several mainstream programming languages (see, e.g., [18,28,29,21,27,1,26]) where they specify the pattern of interactions that each endpoint must follow, i.e., a communication protocol. In this way, once the expected communication protocol at an endpoint has been expressed in terms of a session type, the behavioural correctness of a program at that endpoint can be checked by exploiting syntax-based type checking techniques. The overall correctness of the system is guaranteed when the session types of the interacting endpoints satisfy some deadlock/termination related (see, e.g., [16,13]) compatibility notion. For instance, in case of binary communication, i.e., interaction between two endpoints, *session duality* rules out communication errors like, e.g., deadlocks: by session duality we mean that each send (resp. receive) action in the session type of one endpoint, is matched by a corresponding receive (resp. send) action of the session type at the opposite endpoint. Namely, we have that two endpoints following respectively session types T and $\overline{T}$ ($\overline{T}$ is the dual of T) will communicate correctly.

Duality is a rather restrictive notion of compatibility since it forces endpoints to follow specular protocols. In many cases, endpoints correctly interact even if their corresponding session types are not dual. A typical example is when an endpoint is in receiving state and has the ability to accept more messages than those that could be emitted by the opposite endpoint. These cases are dealt

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

with by considering *session subtyping*: an endpoint with session type T_1 can always be safely replaced by another endpoint with session type T_2, whenever T_2 is a subtype of T_1 (here denoted by $T_2 \leq T_1$). In this way, besides being safe to combine an endpoint with type T_1 with a specular one with type $\overline{T_1}$, it is also safe to combine any such T_2 with $\overline{T_1}$. The typical notion of subtyping for session types is the one by Gay and Hole [17] defined by considering synchronous communication: *synchronous session subtyping* only allows for a subtype to have fewer internal choices (sends), and more external choices (receives), than its supertype. *Asynchronous session subtyping* has been more recently investigated [25,24,15,8,6]: it is more permissive because it widens the synchronous subtyping relation by allowing the subtype to *anticipate* send actions, under the assumption that the subsequent communication protocol is not influenced by the anticipation. Anticipation is admitted because, in the presence of message queues, the effect of anticipating a send is simply that of enqueueing earlier, in the communication channel, the corresponding message. As an example, a session type $\oplus\{l : \&\{l' : \mathbf{end}\}\}$ with a send action on l followed by a receive action on l', is an asynchronous subtype of $\&\{l' : \oplus\{l : \mathbf{end}\}\}$ that performs the same actions, but in reverse order. This admits the safe combination of two endpoints with session types $\oplus\{l : \&\{l' : \mathbf{end}\}\}$ and $\oplus\{l' : \&\{l : \mathbf{end}\}\}$, respectively, because each program has a type which is an asynchronous subtype of the dual type of the partner. Intuitively, the combination is safe in that the initially sent messages are first enqueued in the communication channels, and then consumed.

Behavioural contracts [19,11,10,14,9] (contracts, for short) represent an alternative way for describing the communication behaviour of processes. While session types are defined to be checked against concurrent programs written in some specific programming language, contracts can be considered a language independent approach strongly inspired by automata-based communication models. Contracts follow the tradition of Communicating Finite State Machines (CFSMs) [4], which describe the possible send/receive actions in terms of a labeled-transition system: each transition corresponds with a possible communication action and alternative transitions represent choices that can involve both sends and receives (so called *mixed-choices*, which are usually disregarded in session types). A system is then modeled as the parallel composition of the contracts of its constituting processes. Also in the context of contracts, safe process replacement has been investigated by introducing the notion of *contract refinement*: if a contract C_1 is part of a correct system, then correctness is preserved when C_1 is replaced by one of its subcontracts C_2 (written $C_2 \preceq C_1$ in this paper). Obviously, different notions of contract refinement can be defined, based on possible alternative notions of system correctness. For instance, for binary client/service interaction where correctness is interpreted as the successful completion of the client protocol, the *server pre-order* (see e.g. [3]) has been defined as a refinement of server contracts that preserves client satisfaction. On the other hand, if we move to multi-party systems, and we consider a notion of correctness, called *compliance*, that requires the successful completion of all the partners, an alternative compliance preserving *subcontract relation* [10] is obtained.

Given that both session types and behavioural contracts have been developed for formal reasoning on communication-centered systems, and given that session subtyping and contract refinement have been respectively defined to characterize the notion of safe replacement, it is common understanding that there exists a strong correspondence between these session subtyping and contract refinement. Such a correspondence has been formally investigated for synchronous communication by Bernardi and Hennessy [3]: there exists a natural interpretation of session types into a fragment of contracts where mixed-choice is disallowed, called *session contracts*, such that synchronous subtyping is mapped into a notion of refinement that preserves client satisfaction (but can be applied to both clients and servers; and not only to servers as the server pre-order mentioned above).

The correspondence between session subtyping and contract refinement under asynchronous communication is still an open problem. In this paper we solve such a problem by identifying the fragment of asynchronously communicating contracts for which refinement corresponds to asynchronous session subtyping: besides disallowing mixed-choices as for the synchronous case, we consider a specific form of communication (i.e., FIFO channels for each pair of processes as in the communication model of CFSMs) and restrict to binary systems (i.e., systems composed of two contracts only).

In all, this paper contribution encompasses: (*i*) a new theory of asynchronous behavioural contracts that coincide with CFSMs and includes the notions of contract compliance (correct, i.e. deadlock free, system of CFSMs) and contract refinement (preservation of compliance under any test); and (*ii*) a precise discussion about the notion of *refinement*, showing under which conditions it coincides with asynchronous session subtyping, which is known to be undecidable [7].

More precisely, concerning (*ii*), we show asynchronous subtyping over session types to be encodable into refinement over binary and non mixed-choice asynchronous behavioral contracts (CFSMs). This means that, for contracts of this kind, refined contracts can anticipate outputs w.r.t. the original contract as it happens in the context of session subtyping. Moreover we show that it is crucial, for such a correspondence to hold, that, when establishing refinement between two binary and non mixed-choice asynchronous behavioral contracts, *only tests that are actually binary* (a single interacting contract) *and non mixed-choice are considered*: if we also consider tests that are either multiparty (mutliple interacting contracts) or mixed-choice, in general, a binary and non mixed-choice contract C' that anticipates output w.r.t. a binary and non mixed-choice contract C is not a subcontract of it. This observation has deep implications on decidability properties in the context of general asynchronous behavioral contracts (CFSMs): while *compliance*, i.e. (non) reachability of deadlocking global CFSM states over asynchronous behavioral contracts (CFSMs) is known to be undecidable [4], an obvious argument showing undecidability cannot be found for the refinement relation: such a relation can be put in direct correspondence with asynchronous session subtyping *only for the restricted binary and non mixed-choice setting (including also tests)*. Therefore, since in general an asynchronous behavioral contract (CFSMs) C' that anticipates output w.r.t. a contract C is

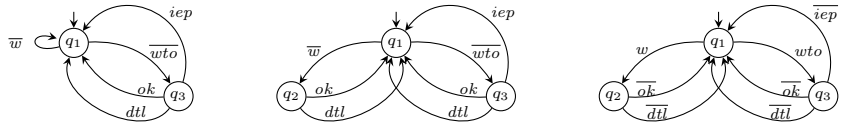

Fig. 1. Fragment of a UDP Server serving Write / WriteTo requests, with specular client and alternative client that records replies only after WriteTo requests.

not a subcontract of it, decidability of refinement over general asynchronous behavioral contracts (CFSMs) remains, quite unexpectedly, an open problem.

Structure of the paper. In Section 2 we define our model of asynchronous behavioural contracts inspired by CFSMs [4]; we define syntax, semantics, correct contract composition, and the notion of contract refinement. In Section 3 we recall session types, focusing on the notion of asynchronous session subtyping [25,7]. In Section 4 we present a fragment of behavioural contracts and we prove that there exists a natural encoding of session types into this fragment of contracts which maps asynchronous session subtyping into contract refinement. Finally, in Section 5 we report some concluding remarks.

2 Behavioural Contracts

In this section we present behavioural contracts (simply contracts for short), in the form of a process algebra (see, e.g. [22,23,2]) based formalization of Communicating Finite State Machines (CFSMs) [4]. CFSMs are used to represent FIFO systems, composed by automata performing send and receive actions having the effect of introducing/retrieving messages to/from FIFO channel. One channel is considered for each pair of sender/receiver automata.

As an example, we can consider a client/service interaction (inspired by the UDP communication protocol) depicted in Figure 1. Communication protocols are denoted by means of automata with transitions representing communication actions: overlined labels denote send actions, while non-overlined labels denote receive actions. The server is always available to serve both Write (w for short) and WriteTo (wto) requests. In the first case, the server replies with OK (ok) or DataTooLarge (dtl), depending on the success of the request or its failure due to an exceeding size of the message. On the other hand, in case of WriteTo, the server has a third possible reply, InvalidEndPoint (iep), in case of wrongly specified destination. We consider two possible clients: a client following a specular protocol, and an alternative client that (given the connectionless nature of UDP) does not synchronize the reception of the server replies with the corresponding requests, but records them asynchronously after WriteTo requests only.

We now present contracts, that can be seen as a syntax for CFSMs. Differently from the examples of communicating automata reported in Figure 1,

$$\frac{j \in I}{\sum_{i \in I} \alpha_i.C_i \xrightarrow{\alpha_j} C_j} \qquad \frac{C\{recX.C/X\} \xrightarrow{\lambda} C'}{recX.C \xrightarrow{\lambda} C'}$$

Table 1. Semantic rules for contracts.

the send (resp. receive) actions will be decorated with a location identifying the expected receiver (resp. sender) contract. This was not considered in the example because, in case of two interacting partners, the sender and receiver of the communication actions can be left implicit.

Definition 1 (Behavioural Contracts). *We consider three denumerable sets: the set $\mathcal{N}$ of message names ranged over by a, b, $\cdots$, the location names Loc, ranged over by l, l', $\cdots$, and the contract variables Var ranged over by X, Y, $\cdots$. The syntax of contracts is defined by the following grammar:*

$$C \ ::= \ \mathbf{1} \ \mid \ \sum_{i \in I} \alpha_i.C_i \ \mid \ X \ \mid \ recX.C \qquad\qquad \alpha \ ::= \ a_l \ \mid \ \overline{a}_l$$

where the set of index I is assumed to be non-empty, and $recX.\_$ is a binder for the process variable X denoting recursive definition of processes: in $recX.C$ a (free) occurrence of X inside C represents a jump going back to the beginning of C. We assume that in a contract C all process variables are bound and all recursive definitions are guarded, i.e. in $recX.C$ all occurrences of X are included in the scope of a prefix operator $\sum_{i \in I} \alpha_i.C_i$. Following CFSMs, we assume contracts to be deterministic, i.e., in $\sum_{i \in I} \alpha_i.C_i$, we have $\alpha_i = \alpha_j$ iff $i = j$. In the following we will omit trailing "$.\mathbf{1}$" when writing contracts.

We use α to range over the actions: $\overline{a}_l$ is a send action, with message a, towards the location l; a_l is the receive of a sent from the location l. The contract $\sum_{i \in I} \alpha_i.C_i$ (also denoted with $\alpha_1.C_1 + \alpha_2.C_2 + \cdots + \alpha_n.C_n$ when $I = \{1, 2, \ldots, n\}$) performs any of the actions α_i and activates the continuation C_i. In case there is only one action, we use the simplified notation $\alpha.C$, where α is such a unique action, and C is its continuation. The contract $\mathbf{1}$ denotes a final successful state.

The operational semantics of contracts C is defined in terms of a transition system labeled over $\{a_l, \overline{a}_l, \mid a \in \mathcal{N}, l \in Loc\}$, ranged over by λ, λ', $\ldots$, obtained by the rules in Table 1. We use $C\{\_/\_\}$ to denote syntactic replacement. The first rule states that contract $\sum_{i \in I} \alpha_i.C_i$ can perform any of the actions α_i and then activate the corresponding continuation C_i. The second rule is the standard one for recursion unfolding (replacing any occurrence of X with the operator $recX.C$ binding it, so to represent the backward jump described above).

The semantics of a contract C yields a *finite-state* labeled transition system,[1] whose states are the contracts reachable from C. It is interesting to observe that such a transition system can be interpreted as a communicating automaton of a CFSM, with transitions $\overline{a}_l$ (resp. a_l) denoting send (resp. receive) actions. The final contract $\mathbf{1}$ coincides with states of communicating automata that have

[1] As for basic CCS [22] finite-stateness is an obvious consequence of the fact that the process algebra does not include static operators, like parallel or restriction.

no outgoing transitions. Moreover, we have that each communicating automaton can be expressed as a contract; this is possible by adopting standard techniques [22] to translate finite labeled transition systems into recursively defined process algebraic terms. Hence we can conclude that our contracts coincide with the communicating automata as defined for CFSMs.

Example 1. As an example of contracts used to denote communicating automata, the alternative client and the server in Figure 1 respectively correspond to the following contracts:[2]

$$\begin{aligned} \mathit{Client} &= \mathit{recX}.(\overline{w}.X + \overline{wto}.(\overline{ok}.X + \overline{dtl}.X + \overline{iep}.X)) \\ \mathit{Server} &= \mathit{recX}.(w.(\overline{ok}.X + \overline{dtl}.X) + wto.(\overline{ok}.X + \overline{dtl}.X + \overline{iep}.X)) \end{aligned}$$

Notice that we have not explicitly indicated the locations associated to the send and receive actions; in fact interaction is binary and the sender and receiver of each communication is obviously the partner location, and we leave it implicit.

We now move to the formalization of contract systems. A contract system is the parallel composition of contracts, each one located at a given location, that communicate by means of FIFO channels. More precisely, we use $[C, \mathcal{Q}]_l$ to denote a contract C located at location l with an input queue $\mathcal{Q}$. The queue contains messages denoted with $a_{l'}$, where l' is the location of the sender of such message a. This queue should be considered as the union of many input channels, one for each sender; in fact the FIFO order of reception is guaranteed only among messages coming from the same sender, while two messages coming from different senders can be consumed in any order, independently from the order of introduction in the queue $\mathcal{Q}$. This coincides with the communication model considered in CFSMs.

Definition 2 (FIFO Contract Systems). *The syntax of FIFO contract systems is defined by the following grammar:*

$$P \ ::= \ [C, \mathcal{Q}]_l \quad | \quad P \| P \qquad\qquad \mathcal{Q} \ ::= \ \epsilon \quad | \quad a_l :: \mathcal{Q}$$

We assume that every FIFO contract system P is such that: (i) all locations are different (i.e. every subterm $[C, \mathcal{Q}]_l$ occurs in P with a different location l), (ii) all actions refer to locations present in the system (i.e., for every a_l or $\overline{a}_l$ occurring in P, there exists a subterm $[C, \mathcal{Q}]_l$ of P), (iii) receive and send actions executed by a contract consider a location different from the location of that contract (i.e. every action a_l or $\overline{a}_l$ does not occur inside a subterm $[C, \mathcal{Q}]_l$ of P), and (iv) messages in a queue comes from a location different from the location of the queue (i.e. every message a_l does not occur inside the queue $\mathcal{Q}$ of a subterm $[C, \mathcal{Q}]_l$ of P).

Terms $\mathcal{Q}$ denote message queues: they are sequences of messages $a^1_{l_1} :: a^2_{l_2} :: \cdots :: a^n_{l_n} :: \epsilon$,[3] where "$\epsilon$" denotes the empty message queue. Trailing ϵ are usually

[2] The correspondence is as follows: the labeled transition systems of the indicated contracts and the corresponding automata in Figure 1 are isomorphic.

[3] As usual, we consider :: right associative.

$$\frac{C \xrightarrow{\overline{a}_{l'}} C'}{[C,\mathcal{Q}]_l \xrightarrow{\overline{a}_{l,l'}} [C',\mathcal{Q}]_l} \qquad [C,\mathcal{Q}]_{l'} \xrightarrow{a_{l,l'}} [C,\mathcal{Q} :: a_l]_{l'} \qquad \frac{P \xrightarrow{\overline{a}_{l,l'}} P' \quad Q \xrightarrow{a_{l,l'}} Q'}{P \| Q \xrightarrow{\tau} P' \| Q'}$$

$$\frac{C \xrightarrow{a_l} C' \quad l \notin \mathcal{Q}}{[C,\mathcal{Q} :: a_l :: \mathcal{Q}']_{l'} \xrightarrow{\tau} [C',\mathcal{Q} :: \mathcal{Q}']_{l'}} \qquad \frac{P \xrightarrow{\lambda} P'}{P \| Q \xrightarrow{\lambda} P' \| Q}$$

Table 2. Asynchronous system semantics (symmetric rules for $\|$ omitted).

left implicit (hence the above queue is denoted with $a^1_{l_1} :: a^2_{l_2} :: \cdots :: a^n_{l_n}$). We overload $::$ to denote also queue concatenation, i.e., given $\mathcal{Q} = a^1_{l_1} :: a^2_{l_2} :: \cdots :: a^n_{l_n}$ and $\mathcal{Q}' = b^1_{l'_1} :: b^2_{l'_2} :: \cdots :: b^m_{l'_m}$, then $\mathcal{Q} :: \mathcal{Q}' = a^1_{l_1} :: a^2_{l_2} :: \cdots :: a^n_{l_n} :: b^1_{l'_1} :: b^2_{l'_2} :: \cdots :: b^m_{l'_m}$. In the following, we will use the notation $l \notin \mathcal{Q}$ to state that if $a_{l'}$ is in $\mathcal{Q}$ then $l \neq l'$, moreover we will use the shorthand $[C]$ to stand for $[C, \epsilon]$.

The operational semantics of FIFO contract systems is defined in terms of a transition system labeled over $\{a_{l,l'}, \overline{a}_{l,l'}, \tau \mid l, l' \in Loc, a \in \mathcal{N}\}$, also in this case ranged over by λ, λ', ..., obtained by the rules in Table 2 (plus the symmetric version for the first two rules of parallel composition). The first rule indicates that a send action $\overline{a}_{l'}$ executed by a contract located at location l, becomes an action $\overline{a}_{l,l'}$: the two locations l and l' denote the sender and receiver locations, respectively. The second rule states that, at the receiver location l', it is always possible to execute a complementary action $a_{l,l'}$ (that can synchronize with $\overline{a}_{l,l'}$) whose effect is to enqueue, in the local queue, a_l: notice that only the sender location l remains associated to message a. The third rule is the synchronization rule between the two complementary labels $\overline{a}_{l,l'}$ and $a_{l,l'}$. The fourth rule is for message consumption: a contract can remove a message a_l from its queue, only if a_l is not preceded by messages sent from the same location l. This guarantees that messages from the same location are consumed in FIFO order. The last rule is the usual local rule used to extend to the entire system actions performed by a part of it.

In the following, we call computation step a τ-labeled transition $P \xrightarrow{\tau} P'$; a computation, on the other hand, is a (possibly empty) sequence of τ-labeled transitions $P \xrightarrow{\tau}{}^* P'$, in this case starting from the system P and leading to P'. To simplify the notation, we omit the τ labels, i.e., we use $P \longrightarrow P'$ for computation steps, and $P \longrightarrow^* P'$ for computations.

We now move to the definition of *correct* composition of contracts. We take inspiration from the notion of *compliance* among contracts as defined, e.g., by Bernardi and Hennessy [3]. Informally, we say that a contract system is correct if all its reachable states (via any computation) are such that: the system has successfully completed or it is able to perform computation steps (i.e. τ transitions) and after each step it moves to a system which is, in turn, correct. In other terms, a system is correct if all of its maximal sequences of τ labeled transitions

either lead to a successfully completed system or are infinite (do not terminate). The notion of *successful completion* for a system is formalized by a predicate $P\surd$ defined as follows:
$\big([C_1, \mathcal{Q}_1] \| \ldots \| [C_n, \mathcal{Q}_n]\big)\surd$ *iff* $\forall i \in \{1, \ldots, n\}.\ C_i = recX_1 \ldots recX_{m_i}.\mathbf{1} \wedge \mathcal{Q}_i = \epsilon$
Notice that the predicate checks whether all input queues are empty and all contracts coincide with the terminated contract $\mathbf{1}$ (possibly guarded by some recursive definition).

We are now ready to define our notion of correct contract composition.

Definition 3 (Correct Contract Composition – Compliance). *A system P is a correct contract composition according to compliance, denoted $P\downarrow$, if for every P' such that $P \longrightarrow^* P'$, then either P' is a successfully completed system, i.e. $P'\surd$, or there exists an additional computation step $P' \longrightarrow P''$.*

Example 2. As an example of correct system we can consider $[Client]_c \| [Server]_s$ where *Client* is the contract defined in Example 1 above for the alternative client in Figure 1 in which all actions are decorated with s, while *Server* is the contract for the server in which all actions are decorated with c. In this system successful completion cannot be reached, but the system never stucks, i.e., every system reachable via a computation always has an additional computation step.

Notice that the above *Client*/*Server* system is a correct contract composition even if the considered *Client* does not behave specularly w.r.t. the server. When we replace a contract with another one by preserving system correctness, we say that we refine the initial contract. As an example, consider the correct system $[b_{l'}.\overline{a}_{l'}]_l \parallel [\overline{b}_l.a_l]_{l'}$ composed of two specular contracts. We can replace the contract $b_{l'}.\overline{a}_{l'}$ with $\overline{a}_{l'}.b_{l'}$ by preserving system correctness (i.e. $[\overline{a}_{l'}.b_{l'}]_l \parallel [\overline{b}_l.a_l]_{l'}$ is still correct). The latter differs from the former in that it anticipates the send action $\overline{a}_{l'}$ w.r.t. the receive action $b_{l'}$. This transformation is usually called *output anticipation* (see e.g. [25]). Intuitively, output anticipation is possible because, under asynchronous communication, its effect is simply that of anticipating the introduction of a message in the partner queue. In the context of asynchronous session types, for instance, output anticipation is admitted by the notion of session subtyping [25,15] that, as we will discuss in the following sections, is the counterpart of contract refinement in the context of session types.

We now formally define contract refinement and we observe that, differently from session types, output anticipation is not admitted as a general contract refinement mechanism.

Definition 4 (Contract Refinement). *A contract C' is a refinement of a contract C, denoted $C' \preceq C$, if and only if, for all FIFO contract systems $([C]_l \| P)$ we have that: if $([C]_l \| P)\downarrow$ then $([C']_l \| P)\downarrow$.*

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').

The above definition contains a universal quantification on all possible contract systems P and locations l, hence it cannot be directly used to algorithmically check contract refinement. To the best of our knowledge, there exists

no general algorithmic characterization (or proof of undecidability) for such a relation. Nevertheless, we can use the definition on some examples.

For instance, consider the two contracts $C = b_{l'}.\overline{a}_{l'}$ and $C' = \overline{a}_{l'}.b_{l'}$ discussed above. We have seen that C' is a safe replacement of C in the specific context $[\,]_l \| [\overline{b}_l.a_l]_{l'}$. But we have that $C' \not\preceq C$ because there exists a discriminating context $[\,]_l \| [\overline{b}_l.a_l + a_l]_{l'}$. In fact, when combined with C', the contract in l' can take the alternative branch a_l, leading to an incorrect system where the contract at l blocks waiting for a never incoming message $b_{l'}$.

The above example shows that output anticipation, admitted in the context of asynchronous session types, is not a correct refinement mechanism for contracts. The remainder of the paper is dedicated to the definition of a fragment of contracts in which it is correct to admit output anticipation, but we first recall session types and asynchronous subtyping.

3 Asynchronous Session Types

In this section we recall session types, in particular we discuss binary session types for asynchronous communication. In fact, for this specific class of session types, subtyping admits output anticipation.

We start with the formal syntax of binary session types, adopting a simplified notation (used, e.g., in [7,8]) 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 5 (Session Types). *Given a set of labels L, ranged over by l, the syntax of binary session types is given by the following grammar:*

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

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 results in this paper. 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 closed session, i.e., a session 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 $\overline{\mu \mathbf{t}.T} = \mu \mathbf{t}.\overline{T}$.

We now move to the session subtyping relation, under the assumption that communication is asynchronous. The subtyping relation was initially defined by Gay and Hole [17] for synchronous communication; we adopt a similar co-inductive definition but, to be more consistent with the contract theory that we will discuss in the next sections, we follow a slightly different approach, being process-oriented instead of channel-based oriented.[4] Moreover, following [25], we consider a generalized version of unfolding that allows us to unfold recursions $\mu \mathbf{t}.T$ as many times as needed.

Definition 6 (n-unfolding).

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

Another auxiliary notation that we will use is that of input context which is useful to identify sequences of initial input branchings; this is useful because, as we will discuss in the following, in the definition of asynchronous session subtyping it is important to identify those output selections that are guarded by input branchings.

Definition 7 (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.

As an example of how input contexts are used, consider the session type $\&\{l_1 : \oplus\{l : \mathbf{end}\}, l_2 : \oplus\{l : \mathbf{end}\}\}$. It can be decomposed as the input context $\&\{l_1 : []^1, l_2 : []^2\}$ with two holes that can be both filled with $\oplus\{l : \mathbf{end}\}$.

We are now ready to recall the *asynchronous* subtyping $\leq$ introduced by Mostrous et al. [24] following the simplified formulation in [7].

Definition 8 (Asynchronous Subtyping, $\leq$). *$\mathcal{R}$ is an asynchronous 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 $\exists n \geq 0, \mathcal{A}$ 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*
   - *$\forall i \in I, (T_i, \mathcal{A}[S_{ki}]^{k\in\{1,\ldots,m\}}) \in \mathcal{R}$;*
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 $(T'\{T/\mathbf{t}\}, S) \in \mathcal{R}$.*

[4] Differently from our definitions, in the channel-based approach of Gay and Hole [17] subtyping is covariant on branchings and contra-variant on selections.

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

Intuitively, the above co-inductive definition says that it is possibile to play a simulation game between a subtype T and its supertype S as follows: if T is the **end** type, then also S is ended; if T starts with an output selection, then S can reply by outputting at least all the labels in the selection (output covariance), and the simulation game continues; if T starts with an input branching, then S can reply by inputting at most some of the labels in the branching (input contravariance), and the simulation game continues. The unique non trivial case is the case of output selection; in fact, in this case the supertype could reply with output selections that are guarded by input branchings. As an example of application of this rule, consider the session type $T = \oplus\{l : \&\{l_1 : \mathbf{end}, l_2 : \mathbf{end}\}\}$. We have that T is a subtype of $S = \&\{l_1 : \oplus\{l : \mathbf{end}\}, l_2 : \oplus\{l : \mathbf{end}\}\}$, previously introduced. In fact, we have that the following relation

$$\{\ (T, S)\ ,\ (\&\{l_1 : \mathbf{end}, l_2 : \mathbf{end}\}, \&\{l_1 : \mathbf{end}, l_2 : \mathbf{end}\})\ ,\ (\mathbf{end}, \mathbf{end})\ \}$$

is an asynchronous subtyping relation. Rule *2.* of the definition is applied on the first pair (T, S). The first item of the rule is used to decompose S (as discussed above) as the input context $\&\{l_1 : []^1, l_2 : []^2\}$ with two holes both filled with $\oplus\{l : \mathbf{end}\}$. The second item trivially holds because the output selection at the beginning of T has only one label l, as also the output selections filling the holes in the decomposition of S. Finally, the third item holds because of the pair $(\&\{l_1 : \mathbf{end}, l_2 : \mathbf{end}\}, \&\{l_1 : \mathbf{end}, l_2 : \mathbf{end}\})$ present in the relation. The first element of the pair is obtained by consuming the output selection at the beginning of T, while the second element by consuming the initial output selection of the terms filling the holes of the considered input context.

The rationale behind asynchronous session subtyping is that under asynchronous communication it is unobservable whether an output is anticipated before an input or not. In fact, anticipating an output simply introduces in advance the corresponding message in the communication queue. For this reason, rule *2.* of the asynchronous subtyping definition admits the supertype to have inputs in front of the outputs used in the simulation game.

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, but the accumulated inputs grows unboundedly.

There are, on the contrary, cases in which the accumulated input is not consumed, as in the infinite simulation game between $T = \mu\mathbf{t}. \oplus \{l : \mathbf{t}\}$ and $S = \mu\mathbf{t}.\&\{l : \oplus\{l : \mathbf{t}\}\}$, in which only output selections are present in the subtype, and an instance of the input branching in the supertype is accumulated in each step of the simulation game.

Example 3. As a less trivial example, we can express as session types the two client protocols depicted in Figure 1:

$$\mathit{SpecularClient} = \mu\mathbf{t}.\oplus\{w.\&\{\mathit{ok}.\mathbf{t} + \mathit{dtl}.\mathbf{t}\}, \mathit{wto}.\&\{\mathit{ok}.\mathbf{t} + \mathit{dtl}.\mathbf{t} + \mathit{iep}.\mathbf{t}\}\}$$
$$\mathit{RefinedClient} = \mu\mathbf{t}.\oplus\{w.\mathbf{t}, \mathit{wto}.\&\{\mathit{ok}.\mathbf{t} + \mathit{dtl}.\mathbf{t} + \mathit{iep}.\mathbf{t}\}\}$$

We have that $\mathit{RefinedClient} \leq \mathit{SpecularClient}$ because the subtyping simulation game can go on forever: when *RefinedClient* selects the output w a input branching is accumulated in front of the r.h.s. type (*SpecularClient* and its derived types), while if *wto* is selected there is no new input accumulation as a (contravariant) input branching follows such a selected output.

A final observation is concerned with specific limit cases of application of rule *2.*; as discussed above, such a rule assume the possibility to decompose the candidate supertype into an initial input context, with holes filled by types starting with output selections. We notice that there exist session types that cannot be decomposed in such a way. Consider, for instance, the session type $S = \mu\mathbf{t}.\&\{l_1 : \mathbf{t}, l_2 : \oplus\{l : \mathbf{t}\}\}$. This session type cannot be decomposed as an input context with holes filled by output branchings because, for every n, $\mathsf{unfold}^n(S)$ will contain a sequence of input branchings (labeled with l_1) that terminate in a term starting with the recursive definition $\mu\mathbf{t}.\_$. Our opinion is that the definition of asynchronous subtyping does not manage properly these limit cases. For instance, the above session type S could be reasonably considered a supertype of $\mu\mathbf{t}.\oplus\{l : \&\{l_1 : \mathbf{t}, l_2 : \mathbf{t}\}$, that simply anticipates the output selection with label l. Such a type has runs with more output selections, because S has a loop of the recursive definition that does not include the output selection; but this is not problematic because such outputs could be simply stored in the message queue. Nevertheless, we have that such a session type is not a subtype of S due to the above observation about the inapplicability of rule *2*.

For this reason, in the following, we will restrict to session types that do not contain infinite sequences of receive actions. Formally, given a session type S and a subterm $\mu\mathbf{t}.T$ of S, we assume that all free occurrences of $\mathbf{t}$ occur in T inside an output selection $\oplus\{\_\}$.

We conclude this section by observing that asynchronous session subtyping was considered decidable (see [25]), but recently Bravetti, Carbone and Zavattaro proved that it is undecidable [7].[5]

4 Mapping Session Types into Behavioural Contracts

In the previous sections we have defined a notion of refinement for contracts and we have seen that output anticipation is not admitted as a general refinement mechanism. Then we have recalled session types where, on the contrary, output anticipation is admitted by asynchronous session subtyping. In this section we show that it is possible to define a fragment of contracts for which refinement

[5] Lange and Yoshida [20] independently proved that a slight variant of asynchronous subtyping, called orphan-message-free subtyping was undecidable.

turns out to coincide with asynchronous session subtyping. More precisely, the natural encoding of session types into contracts maps asynchronous session subtyping into refinement, in the sense that two types are in subtyping relation if and only if the corresponding contracts are in refinement relation.

The first restriction that we discuss is about mixed-choice, i.e., the possibility to perform from the same state both send and receive actions. This is clearly not possible in session types having either output selections or input branchings. But removing mixed-choice from contracts is not sufficient to admit output anticipation. For instance, the system $[b_{l_2}.\overline{c}_{l_2}, \epsilon]_{l_1} \parallel [(a_{l_3}.\overline{b}_{l_1}.c_{l_1}) + c_{l_1}, \epsilon]_{l_2} \parallel [\overline{a}_{l_2}, \epsilon]_{l_3}$ is correct; but if we replace the contract at location l_1 with $\overline{c}_{l_2}.b_{l_2}$, that simply anticipates an output, we obtain $[\overline{c}_{l_2}.b_{l_2}, \epsilon]_{l_1} \parallel [(a_{l_3}.\overline{b}_{l_1}.c_{l_1}) + c_{l_1}, \epsilon]_{l_2} \parallel [\overline{a}_{l_2}, \epsilon]_{l_3}$ which is no longer correct in that the alternative branch c_{l_1} can be taken by the contract in l_2, thus reaching a system in which the contract at l_1 will wait indefinitely for b_{l_2}.

For this reason we need an additional restriction on choices: besides imposing that all the branchings should be guarded by either send or receive actions, we impose all such actions to address the same location l. This is obtained by means of a final restriction about the number of locations: we will consider systems with only two locations, as our objective is to obtain a refinement which is fully abstract w.r.t. subtyping as defined in Section 3, where we considered binary session types (i.e. types for sessions between two endpoints). Given that there are only two locations, each contract can receive only from the location of the partner; hence all receives in a choice address the same location. In general, we will omit the locations associated to send and receive actions: in fact, as already discussed also in Example 1, these can be left implicit because when there are only two locations all actions in one location consider the other location.

A final restriction follows from having restricted our analysis to session types in which there are no infinite sequences of input branchings (see the discussion, at the end of the previous section, about the inapplicability in these cases of rule *2.* of Definition 8). We consider a similar restriction for contracts, by imposing that it is not possible to have infinite sequences of receive actions.

We are now ready to formally define the restricted syntax of contracts considered in this section; it coincides with *session contracts* as defined in [3] plus the restriction on contracts that do not contain infinite sequences of receive actions.

Definition 9 (Session contracts). *Session contracts are behavioural contracts obtained by considering the following restricted syntax:*

$$C \ ::= \ \mathbf{1} \quad | \quad \sum_{i \in I} a^i.C_i \quad | \quad \sum_{i \in I} \overline{a^i}.C_i \quad | \quad X \quad | \quad recX.C$$

where given a session contract $recX.C$, we have that all free occurrences of X occur in C inside a subterm $\sum_{i \in I} \overline{a^i}.C_i$. Notice that we omit the locations l associated to the send and receive actions (which is present in the contract syntax as defined in Definition 1). This simplification is justified because we will consider systems with only two locations, and we implicitly assume all actions of the contract in one location to be decorated with the other location.

In the remainder of this section we will restrict our investigation to FIFO contract systems with only two locations and by considering only session con-

tracts. We will omit the location names also in the denotation of such binary contract systems. Namely, we will use $[C, \mathcal{Q}] \| [C', \mathcal{Q}']$ to denote binary contract systems, thus omitting the names of the two locations as any pair of distinct locations l and l' could be considered.

In the restricted setting of binary session contracts, we can redefine the notion of refinement as follows.

Definition 10 (Binary Session Contract Refinement). *A session contract C' is a binary session contract refinement of a session contract C, denoted with $C' \preceq_s C$, if and only if, for all session contract D, if $([C] \| [D]) \downarrow$ then $([C'] \| [D]) \downarrow$.*

We now define a natural interpretation of session types as session contract; we will subsequently show that this encoding maps asynchronous subtyping into session contract refinement.

Definition 11. *Let T be a session type. We inductively define a function $[\![T]\!]$ from session types to session contracts as follows:*

- $[\![T = \oplus\{l_i : T_i\}_{i \in I}]\!] = \sum_{i \in I} \overline{l_i}.[\![T_i]\!]$; $[\![T = \&\{l_i : T_i\}_{i \in I}]\!] = \sum_{i \in I} l_i.[\![T_i]\!]$;
- $[\![\mu \mathbf{t}.T]\!] = rec\, \mathbf{t}.[\![T]\!]$; $[\![\mathbf{t}]\!] = \mathbf{t}$; $[\![\mathbf{end}]\!] = \mathbf{1}$.

We now move to our main result, i.e., the proof that given two session types T and S we have that $T \leq S$ if and only if $[\![T]\!] \preceq_s [\![S]\!]$. This result has two main consequences. On the one hand, as a positive consequence, we can use the characterization of session subtyping in Definition 8 to prove also session contract refinement. For instance, if we consider the two session subtypes *RefinedClient* and *SpecularClient* of Example 3, we can conclude that

$$recX.(\overline{w}.X + \overline{wto}.(ok.X + dtl.X + iep.X)) \preceq_s$$
$$recX.(\overline{w}.(ok.X + dtl.X) + \overline{wto}.(ok.X + dtl.X + iep.X))$$

because, these two contracts are the encodings of the two above session types according to $[\![\,]\!]$ (notice that these two contracts coincide with the two clients represented in Figure 1). On the other hand, as a negative consequence, we have that session contract refinement $\preceq_s$ is in general undecidable, because asynchronous subtyping $\leq$ is also undecidable as recalled in Section 3.

The first result is about soundness of the mapping of asynchronous session subtyping into session contract refinement, i.e., given two session types T and S, if $T \leq S$ then $[\![T]\!] \preceq_s [\![S]\!]$. In the proof of this result we exploit an intermediary result that simply formalizes the rationale behind asynchronous session subtyping that we have commented after Definition 8: given a correct session contract system, if we anticipate an output w.r.t. a preceding input context, the obtained system is still correct.

Proposition 1. *Consider the two following session contract systems*
$P_1 = [[\![\mathcal{A}[S_k]^{k \in \{1,\ldots,m\}}]\!], \mathcal{Q}] \| [D, \mathcal{Q}' :: l]$ *and*
$P_2 = [[\![\mathcal{A}[\oplus\{l.S_k\}]^{k \in \{1,\ldots,m\}}]\!], \mathcal{Q}] \| [D, \mathcal{Q}']$. *If $P_2 \downarrow$ then also $P_1 \downarrow$.*

Soundness is formalized by the following Theorem.

Theorem 1. *Given two session types T and S, if $T \leq S$ then $[\![T]\!] \preceq_s [\![S]\!]$.*

Proof. (Sketch) This theorem is proved by showing that the following relation

$$\mathcal{S} \;=\; \{\;\; \big(\; [[\![S]\!], \mathcal{Q}] \| [D, \mathcal{Q}'] \;,\; [[\![T]\!], \mathcal{Q}] \| [D, \mathcal{Q}'] \;\big) \;\;\mid\; T \leq S \;\}$$

is such that if $(P_1, P_2) \in \mathcal{S}$ and $P_1\downarrow$, then also $P_2\downarrow$.

To prove this result it is sufficient to consider all possible computation steps $[[\![T]\!], \mathcal{Q}] \| [D, \mathcal{Q}'] \longrightarrow P_2'$ and show that there exists P_1' such that $P_1' \downarrow$ and $(P_1', P_2') \in \mathcal{S}$. For all possible computation steps but one the proof of the above result is easy because, thanks to the subtyping simulation game, the existence of P_1' is guaranteed by a corresponding computation step $P_1 \longrightarrow P_1'$. The unique non trivial case is for send actions executed by the contract $[\![T]\!]$. In this case the existence of P_1' is guaranteed by Proposition 1 applied to $[[\![S]\!], \mathcal{Q}]$: in fact, $[\![S]\!]$ can have the corresponding output after some initial inputs, and P_1' is obtained by removing the output selections from $[\![S]\!]$ and introducing the selected label directly in the partner's queue. This term P_1' is such that $P_1\downarrow$ thanks to Proposition 1.

Given the above relation $\mathcal{S}$, as a consequence of its properties we have that if $T \leq S$ then $[\![T]\!]$ is always a safe replacement for $[\![S]\!]$, in every context, hence $[\![T]\!] \preceq_s [\![S]\!]$. □

A second theorem states completeness, i.e., given two session types T and S, if $[\![T]\!] \preceq_s [\![S]\!]$ then $T \leq S$. Actually, we prove the contrapositive statement.

Theorem 2. *Given two session types T and S, if $T \not\leq S$ then $[\![T]\!] \not\preceq_s [\![S]\!]$.*

Proof. (Sketch) The proof of this theorem is based on the identification of a context that discriminates, in case $T \not\leq S$, the two contracts $[\![T]\!]$ and $[\![S]\!]$. Such a context exists under the assumption that $T \not\leq S$. The context is obtained by considering the encoding of the dual of S, i.e., the specular session type $\overline{S}$. In fact, we have that $[[\![S]\!]] \| [[\![\overline{S}]\!]]\downarrow$ because the two contracts follow specular protocols, while $[[\![T]\!]] \| [[\![\overline{S}]\!]]\downarrow$ does not hold. This last result follows from $T \not\leq S$; we consider a run of the subtyping simulation game between T and S that fails (such a run exists because $T \not\leq S$). If the computation corresponding to this run is executed by $[[\![T]\!]] \| [[\![\overline{S}]\!]]$, we have that a stuck system is reached, hence $[[\![T]\!]] \| [[\![\overline{S}]\!]]\downarrow$ does not hold. □

As a direct corollary of the two previous Theorems we have the following full abstraction result.

Corollary 1. *Given two session types T and S, $T \leq S$ if and only if $[\![T]\!] \preceq_s [\![S]\!]$.*

We conclude by discussing the "fragility" of this full-abstraction result; small variations in the contract language, or in the notion of compliance, break such a result. For instance, consider a communication model (similar to actor-based communication) in which each location has only one input FIFO channel, instead of many (one for each potential sender as for CFSMs). In this model, input actions can be expressed simply with a instead of a_l, indicating that a is expected to be consumed from the unique local input queue. Under this variant output anticipation is no longer admitted. Consider, e.g., $[a.\overline{b}_{l_2}]_{l_1} \parallel [c.\overline{a}_{l_1}.b]_{l_2} \parallel [\overline{c}_{l_2}]_{l_3}$,

which is a correct system. If we replace the contract at location l_1 with $\overline{b}_{l_2}.a$, that simply anticipates an output, we obtain $[\overline{b}_{l_2}.a]_{l_1} \parallel [c.\overline{a}_{l_1}.b]_{l_2} \parallel [\overline{c}_{l_2}]_{l_3}$, which is no longer correct, because in case message b (sent from l_1) is enqueued at l_2 before message c (sent from l_3), the entire system is stuck.

Consider another communication model in which there are many input queues, but instead of naming them implicitly with the sender location, we consider explicit channel names like in CCS [22] or π-calculus [23,5]. In this case, a send actions can be written $\overline{a}_{l,\pi}$, indicating that the message a should be inserted in the input queue π at location l. A receive action can be written a_π, indicating that the message a is expected to be consumed from the input queue π. Also in this model output anticipation is not admitted. In fact, we can rephrase the above counter-example as follows: $[a_{\pi_1}.\overline{b}_{l_2,\pi_2}]_{l_1} \parallel [c_{\pi_2}.\overline{a}_{l_1,\pi_1}.b_{\pi_2}]_{l_2} \parallel [\overline{c}_{l_2,\pi_2}]_{l_3}$.

Another interesting observation is concerned with the notion of compliance. In other papers about asynchronous behavioural contracts [12], compliance is more restrictive, in that it requires that, under fair exit from loops, the computation eventually successfully terminates. Consider, for instance, the binary system $[recX.(\overline{a} + \overline{b}.X)] \parallel [recX.(a + b.X)]$. It satisfies the condition above because, if we consider only fair computations, the send action $\overline{a}$ will be eventually executed thus guaranteeing successful termination. In this case, output covariance, admitted by synchronous session subtyping, is not correct. If we consider the contract $recX.(\overline{b}.X)$ having less output branches (hence following the output covariance principle), and we use it as a replacement for the first contract above, we obtain the system $[recX.(\overline{b}.X)] \parallel [recX.(a + b.X)]$ that does not satisfy the above definition of compliance because it cannot reach successful termination.

5 Related Work and Conclusion

In this paper we introduced a behavioural contract theory based on a definition of *compliance* (correctness of composition of a set of interacting contracts) and *refinement* (preservation of compliance under any test, i.e. set of interacting contracts): the two basic notions on which behavioural contract theories are usually based [19,11,10,14]. In particular, the definitions of behavioural contracts and compliance considered in this paper have been devised so to formally represent Communicating Finite State Machines (CFSMs) [4], i.e. systems composed by automata performing send and receive actions (the interacting contracts) that communicate by means of FIFO channels. Behavioural contracts with asynchronous communication have been previously considered, see e.g. [12]; however, to the best of our knowledge, this is the first paper defining contracts that formally represent CFSMs. Concerning [12], where at each location an independent FIFO queue of received messages is considered for each channel name "a" (enqueuing only messages of type "a" coming from any location "l_1", "l_2",...), here, instead, we consider contracts that represent CFSMs, i.e. such that, at each location, an independent FIFO queue of received messages is considered for each sender location "l" (enqueuing only messages coming from "l" and having any type "a" , "b", ...). Moreover, while in this paper we make use of a notion of

compliance that corresponds to absence of deadlocking global CFSM states [4] (globally the system of interacting contracts either reaches successful completion or loops forever), in [12] a totally different definition of compliance is considered, which requires global looping behaviours to be eventually terminated under a fairness assumption.

Concerning previous work on (variants of) CFSMs, our approach has some commonalities with [20]. In [20] a restricted version of CFSMs is considered w.r.t. [4], by constraining them to be binary (a system is always composed of two CFSMs only) and not to use mixed choice (i.e. choices involving both inputs and outputs). A specific notion of compliance is considered which, besides absence of deadlocking global CFSM states [4] (i.e. compliance as in this paper) also requires each sent message to be eventually received. Thanks to a mapping from the CFSMs of [20] to session types, compliance of a CFSM A with a CFSM B is shown to correspond to subtyping, as defined in [15], between the mapped session type $T(A)$ and the dual of the mapped session type $T(B)$, i.e. $\overline{T(B)}$. With respect to the subtyping definition used in this paper, [15] adds a requirement that corresponds to the eventual reception of sent messages considered in the definition of compliance by [20]: the whole approach of [20] is critically based on the dual closeness property, i.e. $T' \leq T \Leftrightarrow \overline{T} \leq \overline{T'}$, enjoyed (only) by such a variant of subtyping. Notice that, while [20] makes use of a notion of compliance, it does not consider, as in this paper, a notion of refinement defined in terms of compliance preserving testing (as usual in behavioural contract theories where communicating entities have a syntax).

Concerning previous work on session types, our approach has some commonalities with the above mentioned [15]. The above discussed subtyping variant considered in [15] is shown to correspond to substitutability, in the context of concurrent programs written in a variant of the π-calculus, of a piece of code with session type T with a piece of code with session type T', while preserving error-freedom. A specific error-freedom notion is formalized for such a language, that corresponds to absence of communication error (similar to our notion of compliance) plus the guaranteed eventual reception of all emitted messages (an orphan-message-free property that we do not consider). While the program (context) in which the piece of code is substituted can be seen as corresponding to a test in contract refinement, the subtyping characterization in [15] is based on a specific programming language, while in this paper we consider as tests a generic, language independent, set of CFSMs and we discuss the conditions on tests under which we can characterize asynchronous session subtyping.

In this paper we, thus, discussed the notion of *refinement* over asynchronous behavioural contracts that formalize CFSMs, showing precisely under which conditions it coincides with asynchronous session subtyping, which is known to be undecidable. Under different conditions, e.g., not restricting to binary and non mixed-choice contracts only, alternative notions of refinements are obtained on which the already known undecidability results are not directly applicable. This opens a new problem, concerned with the identification of possibly decidable refinement notions for contracts/CFSMs.

References

1. 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, and N. Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.
2. J. C. M. Baeten and M. Bravetti. A ground-complete axiomatisation of finite-state processes in a generic process algebra. *Mathematical Structures in Computer Science*, 18(6):1057–1089, 2008.
3. G. T. Bernardi and M. Hennessy. Modelling session types using contracts. *Mathematical Structures in Computer Science*, 26(3):510–560, 2016.
4. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
5. M. Bravetti. Reduction semantics in markovian process algebra. *J. Log. Algebr. Meth. Program.*, 96:41–64, 2018.
6. M. Bravetti, M. Carbone, J. Lange, N. Yoshida, and G. Zavattaro. A sound algorithm for asynchronous session subtyping. In *Proc. of 30th Int. Conference Concurrency Theory, CONCUR'19*, Leibniz International Proceedings in Informatics. Schloss Dagstuhl, 2019. To appear.
7. M. Bravetti, M. Carbone, and G. Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017.
8. M. Bravetti, M. Carbone, and G. Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018.
9. M. Bravetti, I. Lanese, and G. Zavattaro. Contract-driven implementation of choreographies. In C. Kaklamanis and F. Nielson, editors, *Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, volume 5474 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.
10. M. Bravetti and G. Zavattaro. Contract based multi-party service composition. In *Proc. of Int. Symposium on Fundamentals of Software Engineering, FSEN'07*, volume 4767 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2007.
11. M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Proc. of 6th Int. Symposium Software Composition, SC'07*, volume 4829 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2007.
12. M. Bravetti and 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*, volume 5387 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2008.
13. M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Mathematical Structures in Computer Science*, 19(3):565–599, 2009.
14. G. Castagna, N. Gesbert, and 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*, pages 261–272. ACM, 2008.
15. T. Chen, M. Dezani-Ciancaglini, A. Scalas, and N. Yoshida. On the preciseness of subtyping in session types. *Logical Methods in Computer Science*, 13(2), 2017.

16. F. S. de Boer, M. Bravetti, M. D. Lee, and G. Zavattaro. A petri net based modeling of active objects and futures. *Fundam. Inform.*, 159(3):197–256, 2018.
17. S. J. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.
18. R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *FASE 2016*, pages 401–418, 2016.
19. C. Laneve and L. Padovani. The *Must* preorder revisited. In *Proc. of 18th Int. Conference Concurrency Theory, CONCUR'07*, volume 4703 of *Lecture Notes in Computer Science*, pages 212–225. Springer, 2007.
20. J. Lange and 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*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017.
21. S. Lindley and J. G. Morris. Embedding session types in Haskell. In *Haskell 2016*, pages 133–145, 2016.
22. R. Milner. *Communication and concurrency*. Prentice Hall, 1989.
23. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I/II. *Inf. Comput.*, 100(1), 1992.
24. D. Mostrous and N. Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015.
25. D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *Proc. of 18th European Symposium on Programming, ESOP'09*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009.
26. R. Neykova, R. Hu, N. Yoshida, and F. Abdeljallal. A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F♯. In *CC 2018*. ACM, 2018.
27. D. A. Orchard and N. Yoshida. Effects as sessions, sessions as effects. In *POPL 2016*, pages 568–581, 2016.
28. L. Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.
29. A. Scalas and N. Yoshida. Lightweight session programming in scala. In *ECOOP 2016*, pages 21:1–21:28, 2016.