Choreographies in the Wild

Massimo Bartoletti\textsuperscript{a}, Julien Lange\textsuperscript{b,}\textsuperscript{*}, Alceste Scalas\textsuperscript{a}, Roberto Zunino\textsuperscript{c}

\textsuperscript{a}Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy
\textsuperscript{b}Department of Computing, Imperial College London, UK
\textsuperscript{c}Dipartimento di Matematica, Università degli Studi di Trento, Italy

Abstract

We investigate the use of choreographies in distributed scenarios where, as in the real world, mutually distrusting (and possibly dishonest) participants may be unfaithful to their expected behaviour. In our model, each participant advertises its promised behaviour as a contract. Participants may interact through multiparty sessions, created when their contracts allow to synthesise a choreography. We show that systems of honest participants (which always adhere to their contracts) enjoy progress and session fidelity.

Keywords: contracts, choreographies, session types

1. Introduction

Designing reliable large-scale distributed systems is a difficult challenge. Several intrinsic issues (e.g. concurrency, physical distribution, non-reproducibility of bugs, etc.) make it extremely unlikely that any non-trivial distributed application actually works as its designers intended. To cope with this grand challenge, a lot of foundational and applied research has been carried over in the last 15 years.

One of the most notable approaches is that which studies session types \cite{1, 2, 3} as an abstraction of the communication behaviour of distributed applications. In the top-down approach to session types, the designer of a distributed application specifies its overall communication behaviour through a choreography, which validates some global properties of the application (e.g. safety and deadlock-freedom). However, a distributed application is nothing more than the composition of a number services, running on different nodes in the network. In order to globally enjoy the properties validated by the choreography, all the components forming the application have to be verified; this can be done e.g. by projecting the choreography to end-point types, against which these services are actually type-checked.

The above-mentioned top-down approach is quite suitable in case the designers have complete control over the application, e.g., when they develop all the needed services. However, in many realistic situations this is not the case: for instance, one might need to integrate with already-existing third-party services, e.g., e-commerce facilities, maps, etc. In this case, the top-down approach might not work as expected, because it could be impossible to adapt the end-point type projected from the choreography to the actual interface of the third-party service. In the last few years, approaches mixing top-down and bottom-up composition have been studied to cope with this kind of situation \cite{4, 5, 6}.

Dropping the (unrealistic) hypothesis that a distributed application is built of services coming from a single provider has a further consequence. We can no longer suppose that services live in a “civil society” where they smoothly collaborate with each other; rather, they live in a “wild” environment (like Hobbes’
condition of mere nature \cite{1}) where they must compete for resources, and possibly diverge from the intended behaviour in case they find it convenient to do so. Indeed, this situation is quite reasonable in inter-organisational scenarios, where mistrusted providers compete to achieve conflicting goals. Although some proposals like OASIS \cite{8} try to deal with this issue by prescribing runtime monitoring and logging, no general techniques exist for checking that services actually adhere to their specifications, unless one controls the “bottom layer” of the software stack \cite{9}.

To overcome the state of nature, and move towards the “social contract” (as in Hobbes’ Leviathan), some recent works have proposed to discipline the interaction of services through contracts \cite{10,11}. The idea is a contract-oriented, bottom-up composition, where only those services with compliant contracts can establish sessions through which they interact. To give some intuition about the underlying dynamics, let us assume three participants A, B and K, where the latter plays the role of a contract broker. A possible interaction among these participants could be the following:

1. A advertises its contract \(c_A\) to K;
2. B advertises its contract \(c_B\) to K;
3. K checks if \(c_A\) and \(c_B\) are compliant; if so, it creates a fresh session \(s\) between A and B;
4. at this point, A and B can start interacting through \(s\), by performing the communications prescribed by their contracts. While doing that, they can advertise other contracts, so establishing other sessions (possibly, with different participants).

One key aspect of this contract-oriented approach is that the runtime behaviour of a service may diverge from the advertised one. For instance, participant B above may fail to perform some of the actions in \(c_B\) (either maliciously or accidentally); if A assumes that B respects its contract, then A may get stuck on session \(s\), and consequently it may fail to fulfill its contracts in some other sessions (so possibly damaging itself or other participants). A notable question then arises: how can designers guarantee that their services are honest, in that they always respect the advertised contracts, despite the possible dishonesty of the services they may interact with?

In this paper we address the issue of reconciling the top-down and the bottom-up approaches to service composition in wild environments. To this purpose, we propose a combination of two approaches. Like in the contract-oriented approach \cite{10}, we assume that applications are built bottom-up, by composing services whose advertised contracts admit agreements. However, unlike in previous contract-oriented calculi, service contracts are local session types, in a form similar to those used in \cite{12,13}. A set of contracts admits an agreement whenever it is possible to synthesise from them a choreography — i.e. a global type — whose projections are the contracts themselves \cite{6}. The existence of such a choreography ensures that, if all services behave honestly, the overall application enjoys progress and session fidelity. However, by the wilderness assumption, the actual behaviour of services may deviate from the promised one. In this case, progress no longer holds, but at least we can single out the services which are responsible for contract violations.

 Contributions. We have combined notions and techniques from the session types literature to devise a model of distributed systems where services may be composed bottom-up according to their contracts, and they may deviate from the expected behaviour at runtime. Contracts are expressed as local session types, and multiparty agreements are constructed by adapting the synthesis technique for choreographies of \cite{6}. This allows us to import some results from the session types literature — in particular, the existence of a choreography ensures that contractual agreements enjoy safety and progress (Theorem 3.9). Using choreographies as the basis for agreements is quite flexible: indeed, it allows us to impose further constraints before establishing sessions, e.g. on the number of involved participants, whether or not the session may terminate, etc.

We model services in the \(\text{CO}_2\) calculus \cite{14}, which is adapted here to use (multiparty) local session types as contracts. We define when a service is culpable (i.e., responsible for the next move) in a session, and we show that at least one culpable participant exists in each non-terminated session (Theorem 4.10). If
a system gets stuck, it is always possible to identify which participants violated their contracts. We then adapt the notion of honesty of \cite{11}. We show that an honest service is always able to satisfy its contracts by firing some of its actions, even when the other services do not cooperate (Theorem 6.1). Furthermore, we show that systems of honest services preserve the properties enjoyed by choreographies: in particular, session fidelity (Theorem 6.2) ensures that no extra-contractual interactions are possible in a session, and that honest participants will eventually perform the actions required by their contracts.

We present a case study where the key features of our framework are put into practice. Multiparty sessions, honest participants, and choreography-based contractual agreements are used to model distributed systems where brokers let participants start a new session only if they are advertising contracts such that, as a whole, they implement a class of gossip protocols \cite{15}.

**Synopsis.** The rest of the paper is structured as follows. In Section 2 we sketch an example that we will use through the paper to illustrate our framework. In Section 3 we introduce a contract model based on multiparty session types, and we define contractual agreement as choreography synthesis. In Section 4 we present CO2, suitably adapted to deal with the contracts of Section 3 and we highlight some of its main features. In Section 5 we formalise the notion of honesty. Our main technical results are presented in Section 6 where we show how global properties of choreographies are projected to (honest) CO2 systems \cite{14}. In Section 7 we test the suitability of our framework through a large case study. Finally, in Section 8 we discuss some related work, and in Section 9 we draw some conclusions.

### 2. A Motivating Example

We introduce a running example which we will use through the paper to illustrate our framework. To make symbols lookup easier, we have summarised the syntactic categories and some notation in Table 1 some of the symbols defined therein will only be used in later sections.

An online store \(a\) allows two (still unknown) buyers \(b_1\) and \(b_2\) to make a joint purchase through a simple protocol: after they both request the same item, a quote is sent to \(b_1\), who is then expected to either place an order (order) or end the session (bye); the store also promises to notify \(b_2\) about whether the order was placed (ok) or cancelled (bye). The behaviour promised by \(a\) is described by the following contract:

\[
c_a = b_1?req; b_2?req; b_1!quote; (b_1?order; b_2!ok + b_2?bye; b_2!bye)
\]

What kind of contracts would be compliant with \(c_a\)? One answer consists in the following contracts, advertised by buyers \(B_1\) and \(B_2\):

\[
c_{b_1} = a!req;a?quote; (b'_1!ok;a?order \oplus b'_2!bye; a!bye)
\]

\[
c_{b_2} = a'?req; (b'_1?ok;a'?ok + b'_1?bye; a'?bye)
\]

Here, \(B_1\) promises to send the request to a (still unknown) store \(a\), wait for the quote, and then notify the other buyer \(b'_1\) before accepting or rejecting the store offer; symmetrically, \(B_2\) promises to send the request to the store \(a'\), and then expects to receive the same notification (either ok or bye) from both the other buyer \(b'_1\) and the store itself. Each contract represents the local viewpoint of the participant who advertises.

---

**Table 1: Summary of notation.**

| \(a, b, \ldots\) | Participant identifiers, union of: |
| \(a, b, \ldots \in P\) | Participant names |
| e, e', \ldots | Sorts |
| \(c, c', \ldots\) | Contracts |
| \(T, T', \ldots\) | Systems of contracts |
| \(\gamma, \gamma', \ldots\) | Global types |

| \(u, v, \ldots\) | Session identifiers, union of: |
| \(s, s', t, \ldots \in S\) | Session names |
| \(x, y, \ldots \in \Sigma\) | Session variables |
| \(K, K', \ldots\) | Unsigned contracts |
| \(P, P', \ldots\) | Late contracts |

**Note:** The rest of the paper is structured as follows. In Section 2, we sketch an example that we will use through the paper to illustrate our framework. In Section 3, we introduce a contract model based on multiparty session types, and we define contractual agreement as choreography synthesis. In Section 4, we present CO2, suitably adapted to deal with the contracts of Section 3 and we highlight some of its main features. In Section 5, we formalise the notion of honesty. Our main technical results are presented in Section 6 where we show how global properties of choreographies are projected to (honest) CO2 systems. In Section 7, we test the suitability of our framework through a large case study. Finally, in Section 8, we discuss some related work, and in Section 9, we draw some conclusions.
it: in particular, $c_A$ represents the local viewpoint of the store, and thus it does not (and indeed, it cannot) capture the communications between $B_1$ and $B_2$.

An agreement among $c_A$, $c_{B_1}$, and $c_{B_2}$ may be found after instantiating the participant variables (in each contract) to actual names, through the substitutions \( \{s, a^*, x_1 \} \) and \( \{b_1, b_2, v'_1 \} \). The contracts admit an agreement because a choreography exists which can be projected back to $c_A$, $c_{B_1}$, and $c_{B_2}$. The existence of such a choreography guarantees progress and safety of the contractual agreement (Theorem 3.9). We can exploit the technique in [6] to synthesise a choreography from a set of contracts. In our example, we obtain:

\[
S_{AB_1B_2} = B_1 \rightarrow A : \text{req}; B_2 \rightarrow A : \text{req}; A \rightarrow B_1 : \text{quote} ;
(B_1 \rightarrow B_2 : \text{ok} ; B_1 \rightarrow A : \text{order}; A \rightarrow B_2 : \text{ok} ) + B_1 \rightarrow B_2 : \text{bye}; B_1 \rightarrow A : \text{bye}; A \rightarrow B_2 : \text{bye}
\]

Global type $S_{AB_1B_2}$ says that $B_1$ and $B_2$ send a request to $A$, consecutively; then $A$ replies to $B_1$ with a quote. Subsequently, $B_1$ chooses whether to accept or reject the quote. In the former case, $B_1$ notifies $B_2$ (ok message) then $A$ (order message), who sends a confirmation to $B_2$ (ok message). In the latter case, $B_1$ sends bye to both $B_2$ and $A$, and $A$ confirms the cancellation to $B_2$ by sending bye as well.

However, in realistic distributed scenarios, the existence of a contractual agreement among participants does not guarantee that progress and safety will also hold at runtime: in fact, a participant may advertise a contract promising some behaviour, and then fail to respect it — either maliciously or accidentally. Such failures may then cascade on other participants, e.g., if they remain stuck waiting for a promised message that is never sent.

This sort of situations can be modelled using the CO2 calculus [14]. For instance, a CO2 system for the above scenario will have the following form:

\[
A[(x)(b_1, b_2) \text{tell}_x \downarrow x c_A \cdot \text{fuse} \cdot P_A] \mid B_1[(y)(a, b'_2) \text{tell}_y \downarrow y c_{B_1} \cdot P_{B_1}] \mid B_2[(z)(a, b'_1) \text{tell}_z \downarrow z c_{B_2} \cdot P_{B_2}]
\]

Here, participant $A$ advertises the contract $c_A$ to itself via the primitive $\text{tell}_x \downarrow x c_A$, where $x$ is used as a session handle for interacting with other participants. $B_1$ and $B_2$ advertise their respective contracts to $A$ with a similar invocation.

In this example, $A$ also plays the role of contract broker, which collects the advertised contracts, while looking for an agreement. Once an agreement is found, the $\text{fuse}$ prefix is fired, and a new session is established among all the involved participants. At this point, the participants may realise the agreed contracts — or they may even choose not to. In fact, we will see that when the contracts are violated, the calculus allows for responsible participants to be always ruled out (Theorem 4.10). In systems where participants are honest (i.e. they always fulfil their contracts) the progress and safety of the contractual agreement are also reflected in the runtime behaviour of the CO2 system (Theorem 6.1 and Theorem 6.2).

Other possible agreements. Our framework allows for modelling a wide variety of scenarios. For instance, a participant $B_{12}$ may impersonate both buyers, and promise to always accept the store offer, by advertising the following contract:

\[
c_{B_{12}} = a''\text{req}; a''\text{quote}; a''\text{order}; a''\text{ok}
\]

where the request to the store $a''$ is sent twice (i.e., once for each impersonated customer). In this case, if we instantiate the participant variables in $c_A$ and $c_{B_{12}}$ with substitutions \( \{s, a'' \}, \{b_{12}, b_1, b_2 \} \), we can find an agreement by synthesising the following choreography:

\[
S_{AB_{12}} = B_{12} \rightarrow A : \text{req}; B_{12} \rightarrow A : \text{req}; A \rightarrow B_{12} : \text{quote}; B_{12} \rightarrow A : \text{order}; A \rightarrow B_{12} : \text{ok}
\]

This scenario may be modelled by a CO2 system of the form:

\[
S_2 = A[(x)(b_1, b_2) \text{tell}_x \downarrow x c_A \cdot \text{fuse} \cdot P_A] \mid B_{12}[(w)(a'') \text{tell}_w \downarrow w c_{B_{12}} \cdot P_{B_{12}}]
\]

where the $\text{fuse}$ prefix can now create a session involving $A$ and $B_{12}$.

The participants in the CO2 systems $S_1$ and $S_2$ may also be combined, so to obtain:

\[
S_{12} = A[(x)(b_1, b_2) \text{tell}_x \downarrow x c_A \cdot \text{fuse} \cdot P_A] \mid B_1[(y)(a, b'_2) \text{tell}_y \downarrow y c_{B_1} \cdot P_{B_1}] \mid B_2[(z)(a, b'_1) \text{tell}_z \downarrow z c_{B_2} \cdot P_{B_2}] \mid B_{12}[(w)(a'') \text{tell}_w \downarrow w c_{B_{12}} \cdot P_{B_{12}}]
\]
In this case, after all contracts have been advertised to A, either a session corresponding to $\mathcal{G}_{AB_1B_2}$, or to $\mathcal{G}_{AB_1B_2}$ may take place, thus involving a different number of participants depending on which contracts are fused. As mentioned above, the honesty of A can not depend on which of the two agreements is chosen: to be honest, A must respect its contracts whatever agreement is found with the other participants.

3. A Choreography-Based Contract Model

In this section we exploit concepts and results from the session types literature to devise a multiparty contract model. In Sections 3.1 and 3.2, we present the syntax and semantics of contracts. In Section 3.3, we introduce a notion of culpability. Theorem 3.12 ensures that, if a system of contracts admits an agreement, then at each step of its execution some participants will be “culpable”, i.e. responsible for making the next step in a non-terminated system of contract.

3.1. Local session types as Contracts

We express contracts using the syntax of local session types, in the style of [12, 13]. Let $\mathcal{P}$ and $\mathcal{P}$ be disjoint sets of, respectively, participant names (ranged over by A, B, . . .) and participant variables (ranged over by $\bar{a}, \bar{b}$, . . .), and let $\bar{a}, \bar{b}$ range over $\mathcal{P} \cup \mathcal{P}$. We abstract from the actual data exchanged between participants by using sorts e, e′, . . .. We denote (individual) contracts with c, c′, . . ., and systems of contracts with $T$, $T′$, . . . We write $\mathcal{P}(T)$ for the set of participant names in $T$.

Definition 3.1 (Contracts). The syntax of contracts is given below (we use the colour blue for contracts and green for systems of contracts).

\[
\begin{align*}
    c & ::= \bigoplus_{i \in I} \bar{a}_i e_i; c_i & \text{ where } \forall i \neq j \in I: (\bar{a}_i, e_i) \neq (\bar{a}_j, e_j) & \text{ (Internal choice)} \\
    & \ | \sum_{i \in I} \bar{a}_i e_i; c_i & \text{ where } \forall i \neq j \in I: e_i \neq e_j & \text{ (External choice)} \\
    & \ | \mu x.c & \text{ (Recursion)} \\
    & \ | x & \text{ (Recursion variable)} \\
    & \ | \_ & \text{ (End)} \\
    T & ::= T | T′ & \text{ where } \mathcal{P}(T) \cap \mathcal{P}(T′) = \emptyset & \text{ (System composition)} \\
    & \ | \langle \mathcal{A} \rangle : \rho & \text{ where } \mathcal{A} \neq \mathcal{B} \text{ and } \rho \text{ is a sequence of sorts} & \text{ (Participant’s contract)} \\
    & \ | 0 & \text{ (End)} \\
\end{align*}
\]

Intuitively, in an internal choice, after sending the message $e_i$ to participant $\bar{a}_i$, behaviour $c_i$ takes place; whereas in an external choice if a message of sort $e_i$ is received from $\bar{a}_i$, then behaviour $c_i$ takes place. We write $\text{fv}(c)$ for the (free) participant variables in c. We identify empty internal and external sums, i.e. $0 = \bigoplus_{i \in \mathcal{I}} \bar{a}_i e_i; c_i = \sum_{i \in \mathcal{I}} \bar{a}_i e_i; c_i$.

A system of contracts $T$ may be either: (i) a parallel composition of systems $T | T′$; or (ii) a named contract $\langle \mathcal{A} \rangle : \rho$, saying that participant $\mathcal{A}$ promises to behave according to $\rho$; (iii) a queue $(\mathcal{A}B) : \rho$ of messages from $\mathcal{A}$ to $\mathcal{B}$; we write $[]$ for the empty sequence of message. In a system $T$, we assume that there are exactly two uni-directional queues per pair of participants; that is, two participants A and B can only communicate via queues named AB (from A to B) and BA (from B to A). Whenever we refer to a system $T$, we assume that it respects the conditions in Definition 3.1 as well as those above.
Hereafter, we will use the following notation. We write:

\[ A(B;e_0 \oplus e_1) \mid (AB) : \rho \xrightarrow{A \rightarrow B,e} A(e_0) \mid (AB) : \rho \cdot e \]

\[ A(B;e_0 + e_1) \mid (BA) : e \cdot \rho \xrightarrow{A \leftarrow B,e} A(e_0) \mid (BA) : \rho \]

\[ \mu \times e \equiv e \{ \mu \times e / \times \} \quad A(\emptyset) \equiv 0 \]

\[
\begin{array}{c}
\frac{T_1 \xrightarrow{\lambda} T'_1}{T_1 | T_2 \xrightarrow{\lambda} T'_1 | T_2}
\end{array}
\]

commutative monoidal laws for \( \cup \) and +

Figure 1: Semantics of systems of contracts.

3.2. Semantics

The semantics of systems of contracts is given in Figure 1 where labels \( \lambda \) have either the form \( A \rightarrow B : e \) or \( A \leftarrow B : e \). The label \( A \rightarrow B : e \) means that a message of sort \( e \) is sent by participant \( A \) to \( B \); instead, the label \( A \leftarrow B : e \) means that a message of sort \( e \) is received by \( A \) from \( B \). The first rule in Figure 1 says that, after an internal choice, \( A \) puts a message \( e \) on its queue for participant \( B \). The second rule says that, in an external choice, \( A \) can receive a message (of the expected sort) from an input queue \( BA \). The semantics is assumed up-to the congruence relation \( \equiv \) defined in Figure 1.

Notation 3.2. Hereafter, we will use the following notation. We write:

- \( T \xrightarrow{\alpha} T' \) when either \( T \xrightarrow{A \rightarrow B,e} T' \) or \( T \xrightarrow{A \leftarrow B,e} T' \).
- \( T \rightarrow T' \) when \( T \xrightarrow{\text{empty}} T' \), and the label is unimportant.
- \( \mathcal{P}(T) \) (resp. \( \mathcal{E}(T) \)) for the set of participant (resp. queue) names in \( T \).
- \( \mathcal{Q}(T) \) for the parallel composition of the empty queues connecting all distinct pairs of participants appearing in \( T \).
- \( T(A) \) for the contract of \( A \) in \( T \), and \( T[AB] \) for the content of the queue \( AB \), i.e.:

\[
T(A) = \begin{cases} 
\emptyset & \text{if } T \equiv A(\emptyset) \mid T' \\
\rho & \text{if } T \equiv (AB) : \rho \mid T'
\end{cases}
\]

\[
T[AB] = \begin{cases} 
\rho & \text{if } T \equiv (AB) : \rho \mid T'
\end{cases}
\]

Example 3.3. From the example in Section 3 consider the instantiated contracts of the store \( A \) and its customer \( B_{12} \). We illustrate the system, and how it progresses (hereafter, for the sake of readability, we often omit empty queues).

\[
T_{AB_{12}Q} = A(c_4 \{ \{/a' \} \{ B_{12}/b_{1,2} \} \}) \mid B_{12}(c_{B_{12}} \{ B_{12}/b_{1,2} \}) \mid (B_{12}A) : []
\]

\[
\xrightarrow{B_{12} \rightarrow A \text{req}} A(B_{12} ? \text{req}; B_{12} ? \text{req}; \ldots) \mid B_{12}(A \text{req}; A \text{req}; \ldots) \mid (B_{12}A) : []
\]

\[
\xrightarrow{A \leftarrow B_{12} \text{req}} A(B_{12} ? \text{req}; \ldots) \mid B_{12}(A \text{req}; \ldots) \mid (B_{12}A) : []
\]

3.3. Choreography Synthesis as Agreement

We introduce in Definition 3.4 below the agreement relation, that tells whether some contracts can be combined to describe a correct interaction. Roughly, a set of contracts admits an agreement if it can be assigned a choreography. To do that, we borrow some results from [6][10]: in particular, we exploit a typing system which assigns a (unique) choreography to a set of contracts.

Definition 3.4. The syntax of choreographies \( \mathcal{G} \) is defined as follows:

\[
\mathcal{G} ::= A \rightarrow B ; e ; \mathcal{G} \mid \mathcal{G} + \mathcal{G}' \mid \mathcal{G} \mid \mathcal{G}' \mid \mu \times \mathcal{G} \mid x \mid 0
\]
The first production means that $\mathcal{A}$ sends a message of sort $e$ to $\mathcal{B}$, then interactions in $\mathcal{G}$ take place; $\mathcal{G} + \mathcal{G}'$ means that either interactions in $\mathcal{G}$ or in $\mathcal{G}'$ take place; $\mathcal{G} | \mathcal{G}'$ means that interactions in $\mathcal{G}$ and $\mathcal{G}'$ are executed concurrently; the rest of the productions are for recursive interactions, and termination ($\mathcal{0}$).

To synthesise a choreography $\mathcal{G}$ from a system of contracts $T$, we analyse what actions can occur at each possible state of $T$. These actions are contained in the set $\text{TRS}(T)$ defined below. The auxiliary predicate $T \downarrow \downarrow$ holds whenever two participants are able to synchronise (i.e., $\mathcal{B}$ may receive the message sent by $\mathcal{A}$ as soon as it appears on the queue).

**Definition 3.5 (System of contracts Ready Set).** We define the ready set of a system as follows:

$$\text{TRS}(T) = \begin{cases} \{A \leftarrow B \} \cup \text{TRS}(T') & \text{if } T \equiv A(\sum_{i \in I} B_i ? e; e_i) | T'' \\ \{A \rightarrow B_i | i \in I \} \cup \text{TRS}(T') & \text{if } T \equiv A(i \oplus B_i ? e; e_i) | T'' \\ \emptyset & \text{if } T \equiv 0 \end{cases}$$

We write $T \downarrow \downarrow$ when $\exists A, B : A \rightarrow B \in \text{TRS}(T) \land B \leftarrow A \in \text{TRS}(T)$. We write $T \uparrow \downarrow$ if $T \downarrow \downarrow$ does not hold.

In Figure 2 we present the synthesis of choreographies from systems of contracts (this is a slight adaptation of the synthesis in [16]). Typing judgements have the form $\Gamma \vdash T \triangleright \mathcal{G}$, where $\Gamma$ is an environment which keeps track of recursion variables. Roughly, $\Gamma$ maps pairs $(\mathcal{A}, x)$ to $x'$, where $x$ is the local recursion variable of participant $\mathcal{A}$ and $x'$ is a global recursion variable. A typing judgement $\Gamma \vdash T \triangleright \mathcal{G}$ says that, under hypothesis $\Gamma$, the system $T$ is assigned to global type $\mathcal{G}$.

We write $\mathcal{P}(\mathcal{G})$ (resp. $\mathcal{C}(\mathcal{G})$) for the set of participant (resp. queue) names in $\mathcal{G}$.

**Definition 3.6 (Agreement).** We say that a system of contracts $T$ admits an agreement when $\vdash T \triangleright \mathcal{G}$ can be inferred from the rules in Figure 2.

Essentially, the synthesis rules allow to execute a set of contracts step-by-step, while keeping track of the structure of the interactions in a global type. The rules are driven by the ready set of $T$ and the structure of its processes. We briefly describe them below.

Sequential interactions are dealt with by rule $[\cdot]$. The rule validates prefixes provided that the continuation is typable and that no other interactions are possible in $T$. For instance, rule $[\cdot]$ does not apply to the system

$$\mathcal{A}_2(\mathcal{B}_1 ? e; c_1) | \mathcal{A}_1(\mathcal{B}_1 ? e; c'_1) | \mathcal{A}_2(\mathcal{B}_2 ? e; c_2) | \mathcal{B}_2(\mathcal{A}_2 ? e; c'_2)$$

because there is no ordering relation between the actions of $\mathcal{A}_1$ and $\mathcal{B}_1$ on one hand, and the actions of $\mathcal{A}_2$ and $\mathcal{B}_2$ on the other hand. In such cases, the rule $[\cdot]$ must be used. In fact, the premise $T \downarrow \downarrow$ guarantees that a synthesised global type is unique (up-to structural congruence), cf. [16, Theorem 3.3].

Concurrent branches are introduced by rule $[\cdot]$. The rule validates concurrent branches when they can be validated using a partition of the system being considered (recall that $\mathcal{P}(\mathcal{G}) \cap \mathcal{P}(\mathcal{T}') = \emptyset$ by assumption). Choice in a global type is dealt with by rules $[\cdot]$ and $[\cdot]$. Rule $[\cdot]$ introduces the global type choice operator, it requires both branches to be typable and that no other interactions are possible in $T$ — for the same reason as in rule $[\cdot]$. Rule $[\cdot]$ discharges a branch of an external choice. This allows, e.g., to assign a type to systems of the form:

$$\mathcal{A}(B ? e \oplus B ? e') | \mathcal{B}(A ? e + A ? e_1 + A ? e_2)$$

Recursion is handled by rules $[\cdot]$ and $[\cdot]$. The former rule “guesses” the participants involved in a recursive behaviour. If two of them interact, $[\cdot]$ validates the recursion provided that the system can be typed when such participants are associated to the global recursion variable $x$ — assuming $x$ is fresh. Rule $[\cdot]$ checks that all the participants in the recursion have reached a local recursion variable corresponding to the global recursion. The termination $\mathcal{0}$ is introduced by rule $[\cdot]$, which only applies when all the participants in $T$ are terminated. Rule $[\cdot]$ validates a system up-to structural congruence, so allowing recursive behaviour to be unfolded.
First define some undesirable agreement, since $\Gamma \vdash G$. The system has an unexpected unspecified reception configuration. An is one participant expecting a message, and all participants are either terminated or waiting for an input. For a system of contracts from himself. Definition 3.8 formalises these concepts. Example 3.7. Recall the buyer-seller scenario from Section 2. By combining the contract of store $A$ with those of customers $B_1$ and $B_2$, we obtain the system:

$$T_{AB:B_2} = A(x_1, x_2) | B_1(c_1) \{ x_1 \} | B_2(c_2) \{ x_2 \}$$

The system $T_{AB:B_2}$ admits an agreement: indeed, we have that $\vdash T_{AB:B_2} \rightarrow \mathcal{G}_{AB:B_2}$ (where the choreography $\mathcal{G}_{AB:B_2}$ is that defined at page 7). Similarly, if we combine the store $A$ with $B_{12}$, we have that $T_{AB:B_{12}}$ admits an agreement, since $\vdash T_{AB:B_{12}} \rightarrow \mathcal{G}_{AB:B_{12}}$ (where $\mathcal{G}_{AB:B_{12}}$ is the choreography at page 7).

Theorem 3.9 below establishes some properties of systems of contracts which do admit an agreement. We first define some undesirable properties of systems. A system is a deadlock if all its queues are empty, there is one participant expecting a message, and all participants are either terminated or waiting for an input. An unspecified reception configuration is a system for which there is a participant $B$ who is permanently unable to read any messages from its queues, i.e., there are unexpected messages in $B$'s buffers. A system has a self-interaction if some participant is expected to send a message to himself (or to receive a message from himself). Definition 3.8 formalises these concepts.

Definition 3.8 (Deadlock/unspecified reception). For a system of contracts $T$, we say that:

- $T$ is a deadlock if the following conditions hold:

  $\forall AB \in \mathcal{C}(T): T[AB] = []$

  $\exists C \in \mathcal{P}(T): T(C) \equiv A?ec + e'$

  $\forall C \in \mathcal{P}(T): T(C) \equiv A?ec + e' \lor T(C) \equiv 0$

- $T$ is an unspecified reception configuration if $\exists B \in \mathcal{P}(T)$:

  $T(B) \equiv A?ec + c_1 + c_2 \land (\forall e, A \in \mathcal{P}(T): T(B) \equiv A?ec + e' \implies T[AB] \neq [] \land T[AB] \neq e \cdot \rho)$

- $T$ has a self-interaction if there exists $A \in \mathcal{P}(T)$ such that $T'(A) \equiv A?ec + e$ or $T'(A) \equiv A?ec + e'$. 

Figure 2: Inference of global types.
Theorem 3.9 (Agreement). If \( T \) admits an agreement, then for all \( T' \) such that \( T \downarrow \mathcal{Q}(T) \xrightarrow{*} T' \):

1. \( T' \) is not a deadlock.
2. \( T' \) is not an unspecified reception configuration.
3. \( T' \) has no self-interactions.
4. Either there is \( T'' \) such that \( T' \xrightarrow{\rightarrow} T'' \), or
   \[
   \forall AB \in C(T): T'[AB] = [] \quad \text{and} \quad \forall C \in \mathcal{P}(T'): T'(C) \equiv 0
   \]

Proof. Items (1), (2) and (4) follow from Theorem 3.1 in [16]. Item (3) is a direct consequence of Theorem 3.9 and of the assumption that for all \( AB \in C(T) \): \( A \neq B \).

3.4. Culpability

We introduce a notion of culpability in systems of contracts. Intuitively, a participant is culpable in a system if it is able to send or receive some message. This condition is not necessarily permanent: a culpable participant may exculpate itself by sending/receiving the required messages.

Definition 3.10 (Contract culpability). A is culpable in \( T \) when \( T \overset{\mathsf{A}:=B \mathsf{e}}{\longrightarrow} \), for some \( B \) and \( e \).

As expected, a participant \( A \) is not culpable in a system of contracts not including any contracts of \( A \).

Lemma 3.11. For all system \( T \), if \( A \notin \mathcal{P}(T) \), then \( A \) is not culpable in \( T \).

The following theorem ensures that, if a system of contracts admits an agreement, then at each step of its execution (until reaching the final state \( 0 \)) some participant is culpable. This guarantees that, unless some participants want to stay culpable forever, the system of contracts enjoys progress.

Theorem 3.12 (Presence of culpability in contracts). If \( T \) admits an agreement and \( T \downarrow \mathcal{Q}(T) \xrightarrow{*} T' \neq 0 \), then there exists at least one culpable participant in \( T' \).

Proof. Since \( T' \neq 0 \) then, for some \( A, B, e, c, c' \), either (a) \( T' \equiv A(Be; c \oplus c') \downarrow \), or (b) \( T' \equiv A(Be; c + c') \downarrow \).

Case (a) follows directly, since a send action can always be fired. Case (b) follows from Theorem 3.9, i.e., there must be a message for \( A \) in a queue \( BA \), otherwise we would have a deadlock or an unspecified reception configuration.

4. Introducing Choreographies to the Wild

We present a variant of the CO2 calculus (for COntract-Oriented computing) [14] which we adapt to deal with the multiparty contracts and sessions introduced in Section 3. In Sections 4.1 and 4.2 we give its syntax and semantics, while in Section 4.3 we explain in detail how new sessions are established, providing several examples. In Section 4.4 we formalise when participants in CO2 systems are “culpable”, and show that a non-terminated session always implies the (possibly transient) culpability of some participant (Theorem 4.10). Finally, in Section 4.5 we discuss how the session creation mechanics can be adjusted in order to enforce some desired properties on the resulting interactions.
commutative monoidal laws for \( + \) and \( \cdot \), with \( 0 \) as identity element.

\[
\begin{align*}
\mathcal{A}[u]P \equiv (u)\mathcal{A}[P] & \quad \text{if } u \notin \text{fv}(Z) \\
(u)Z \equiv Z & \quad \text{if } u \notin \text{fv}(Z) \\
(u)(v)Z \equiv (v)(u)Z & \quad \text{if } u \notin \text{fv}(Z)
\end{align*}
\]

\[
\mathcal{A}[K] | \mathcal{A}[K'] \equiv \mathcal{A}[K | K']
\]

Figure 3: Structural congruence rules for CO\(_2\).

4.1. Syntax

Let \( S \) and \( S \) be disjoint sets of, respectively, session names (ranged over by \( s, s', t, \ldots \)) and session variables (ranged over by \( x, y, z \ldots \)). Let \( u, v, \ldots \) range over \( S \cup S \). The syntax of CO\(_2\) is given as follows:

\[
\begin{align*}
\text{(Processes)} & \quad P, Q \ ::= \sum_{i \in I} P_i \cdot P_i \mid P \mid Q \mid (\overline{s})P \mid (\overline{a})P \mid X(\overline{a}, \overline{a}) \\
\text{(Prefixes)} & \quad p \ ::= \tau \mid \text{tell}_{\overline{u}, c} \mid \text{fuse} \mid \text{do}_s^e \\
\text{(Latent contracts)} & \quad K \ ::= \downarrow_u \mathcal{A} \text{ says } c \mid K \mid K' \\
\text{(Systems)} & \quad S \ ::= \mathcal{A}[P] \mid \mathcal{A}[K] \mid s[T] \mid S \mid S \mid (\overline{s}, \overline{a})S \mid 0
\end{align*}
\]

CO\(_2\) features CCS-style processes, equipped with branching \( \sum \) (not to be confused with the choice operator used in contracts), parallel composition \( \cdot \), restrictions of session and participant variables, and named process invocation. As usual, we denote the empty sum with \( 0 \), and omit its trailing occurrences. We assume that each process identifier \( X \) has a unique defining equation \( X(y_1, \ldots, y_n, a_1, \ldots, a_m) := P \) such that \( \text{fv}(P) \subseteq \{y_1, \ldots, y_n, a_1, \ldots, a_m\} \) and each occurrence of process identifiers in \( P \) is prefix-guarded. We shall take the liberty of omitting the arguments of \( X(\overline{y}, \overline{a}) \) when they are clear from the context. The prefixes are for internal action (\( \tau \)), contract advertisement (\text{tell})`, session creation upon contractual agreement (\text{fuse}), and execution of contractual actions (\text{do}).

A latent contract of the form \( \downarrow_u \mathcal{A} \text{ says } c \) represents the promise of participant \( \mathcal{A} \) to fulfil \( c \) by executing \text{do}-actions, once \( x \) has been instantiated with a session name.

CO\(_2\) systems may be parallel compositions of participants \( \mathcal{A}[P] \) (where \( P \) is the process being executed by \( \mathcal{A} \)), collections of latent contracts \( \mathcal{A}[K] \) (where \( \mathcal{A} \) is the participant to which the contracts in \( K \) have been advertised), and sessions \( s[T] \) (where \( s \) is a session name, and \( T \) is a system of contracts as in Section 3). We assume well-formed systems where each participant name \( \mathcal{A} \) can appear with at most one process — i.e., we rule out systems like \( \mathcal{A}[P] \mid \mathcal{A}[Q] \).

Note that CO\(_2\) process and system productions allow to delimit both session names/variables (\( \overline{a} \)) and participant variables (\( \overline{a} \)), but not participant names: this is because participant names are used to uniquely (and statically) identify participants in a network, and therefore we do not want these names to be \( \alpha \)-converted.

4.2. Semantics

The semantics of CO\(_2\) (Figure 1) is defined up-to a congruence relation between CO\(_2\) systems (Figure 3). The rules defining \( \equiv \) are mostly standard. In Figure 3, \( Z, Z', Z'' \) range over processes, systems, or latent contracts; all rules involving delimited session names/variables \( u, v \) also hold for delimited participant variables \( a, b \) (e.g., \( \mathcal{A}[a]P \equiv (a)\mathcal{A}[P] \)). Rule \( \mathcal{A}[(u)P] \equiv (u)\mathcal{A}[P] \) allows to move delimitations between CO\(_2\) systems and processes; the last rule \( \mathcal{A}[K] \mid \mathcal{A}[K'] \equiv \mathcal{A}[K | K'] \) will be discussed in Section 4.3 and in Example 4.4.

Definition 4.1 (Semantics of CO\(_2\)). \( \xi \) is the smallest relation between CO\(_2\) systems closed under the rules of Figure 4 which are to be considered up-to the structural congruence relation \( \equiv \) in Figure 3. The labels \( \ell \) have the form \( \mathcal{A}: p \), where \( \mathcal{A} \) is the name of the participant firing prefix \( p \).

We briefly discuss the rules in Figure 4. \( \text{[CO}_2\text{-Tau]} \) allows a participant \( \mathcal{A} \) to fire an internal action. \( \text{[CO}_2\text{-Tell]} \) allows \( \mathcal{A} \) to advertise a contract \( c \) to \( B \); as a result, a new latent contract is created, recording the fact that
Example 4.2. Consider the CO$_2$ label $A$ it was promised by $A$. According to rule $x$ consider the following reduction: $(\tau)$ substitutions $(constructed from T)$ with the communication queues connecting its participants. Rule $[CO_2-Def]$ allows $A$ to perform an input/output action $e$ towards $B$ on session $s$, provided that $T$ (i.e., the current state of the contracts at $s$) allows for it. The other CO$_2$ rules are standard. In rule $[CO_2-Del]$, the prefix $p$ fired in the premise becomes $\tau$ in the conclusions, when $p$ contains a delimited name/variable. For instance, consider the following reduction: $(x) A[tell_{\downarrow x} e . P] \xrightarrow{\text{del}(p)} (x) (A[P] | B[\downarrow x A \ says e])$; one would not use the label $A : \tau : A$ if $x$ is not visible outside the process of $A$.

Example 4.2. Consider the CO$_2$ system:

$$S = A[do^*_\uparrow int + do^*_\uparrow bool] \mid s[A(B!int) \mid B(A?int) \mid (AB) : []] \mid B[do^*_\uparrow int]$$

Participant $A$ is willing to perform an action towards $B$ on session $s$, with either a message of sort $\text{int}$ or $\text{bool}$. However, $A$’s contract in $s$ only specifies that $A$ should send a message of sort $\text{int}$ to $B$: therefore, according to rule $[CO_2-Do]$, only the first branch of $A$ may be chosen, and the system reduces as follows.

$$S \xrightarrow{\text{del}(p)} A[0] \mid s[A(0) \mid B(A?int) \mid (AB) : \text{int}] \mid B[do^*_\uparrow int]$$

$$S \xrightarrow{\text{del}(p)} A[0] \mid s[A(0) \mid B(0) \mid (AB) : []] \mid B[0]$$

4.3. Reaching Agreements

We now discuss the mechanics underlying the $[CO_2-Fuse]$ rule, with a focus on the agreement relation which regulates session establishment.
Definition 4.3 (Agreement relation $K \triangleright^T P$). Let $K \equiv \big|_{i \in I} \downarrow_{x_i} A_i \text{ says } c_i$, and let $\pi : \mathcal{P} \rightarrow \mathcal{P}$ and $\sigma : \mathcal{S} \rightarrow \mathcal{S}$ be two substitutions mapping participant variables to participant names, and session variables to session names, respectively. Also, let $T \equiv \big|_{i \in I} A_i(e_i)\pi$. Then, $K \triangleright^T P$ holds iff:

1. $\text{dom}(\sigma) = \bigcup_{i \in I} \{x_i\}$;
2. $\text{dom}(\pi) = \bigcup_{i \in I} \text{fv}(e_i)$;
3. $T$ admits an agreement — i.e., $\exists \mathcal{G} : T \triangleright \mathcal{G}$.

Conditions 1 and 2 on $\sigma$ and $\pi$, guarantee that all the session and participant variables are indeed instantiated. Condition 3 holds whenever a judgment $T \triangleright \mathcal{G}$ can be inferred, for some $\mathcal{G}$ (see Definition 3.6).

We discuss two safety issues that Definition 4.3 rules out:

1. Our assumptions on a system $T$ imply that each participant may have at most one contract in $K$, cf. Section 3.2. This restriction allows us to avoid problematic scenarios arising during session establishment. For instance, consider a process $A[K|P]$ advertising two contracts $c$ and $c'$ using session variables $x$ and $y$, respectively. Were we not preventing both contracts to appear in the same session $s$, $c$ and $c'$ could be fused together in $s$. Therefore, by rule $[\text{CO-Fuse}]$, both $x$ and $y$ would be replaced by $s$. Hence, each prefixes of the form $do^x$ and $do^y$ in $P$ would be turned into $do^s$, possibly mixing the actions originally intended for one contract with those intended for the other. In such situations, one cannot enforce safety properties such as deadlock freedom, since contracts $c$ and $c'$ are independent in $T$, while there might be causality dependencies in $P$.

2. By item (3) of Theorem 3.9, it cannot be the case that, within a contract $c_i$ belonging to $A_i$, a free participant variable is substituted by $A_i$ itself.

When searching for possible agreements, the congruence rule $A[K] \vdash A[K'] \equiv A[K \mid K']$ in Figure 3 allows for rearranging collections of latent contracts, in order to select the compliant ones that satisfy the premises of $[\text{CO-Fuse}]$ rule, cf. Example 4.4 below.

Example 4.4 (Agreements and structural congruence). Consider the system:

$$S = (x, y, z)(B[\text{fuse} \cdot P \mid Q] \mid B[K] \mid A_1[\ldots] \mid A_2[\ldots] \mid A_3[\ldots])$$

where $K = \downarrow_x A_1 \text{ says } A_2 \downarrow y A_2 \text{ says } A_3 \downarrow y A_3 \text{ says } B[\text{bool}]$.

The $\downarrow$ prefix of participant $B$ cannot be immediately fired: no contract matches $A_3$’s, and thus the three latent contracts cannot be assigned a global type. Therefore, there is no $T, \pi$ such that $K \triangleright^T P$, as required by rule $[\text{CO-Fuse}]$. However, we can rearrange the latent contracts via structural congruence:

$$S \equiv S' = (z)(x, y)(B[\text{fuse} \cdot P \mid Q] \mid B[K'] \mid B[\downarrow_x A_3 \text{ says } B[\text{bool}] \mid A_1[\ldots] \mid A_2[\ldots] \mid A_3[\ldots])$$

where $K' = \downarrow_x A_1 \text{ says } A_2 \downarrow y A_2 \text{ says } A_3 \downarrow y A_3 \text{ says } A_1 \downarrow x.$

If we take $\sigma = \{s/x, y\}$ (with $s$ fresh) and $T = A_1(A_2[\text{int}] \mid A_2[\text{int}])$, we now have $K' \triangleright^T P$. Indeed, $T$ is composed by $A_1$’s and $A_2$’s contracts, and $T \triangleright A_1 \rightarrow A_2 : \text{int}$; furthermore, the domains of $\sigma$ and $\pi = \emptyset$ comply with the conditions in Definition 4.3.

Since all the premises of $[\text{CO-Fuse}]$ are now satisfied, it is possible to create a new session $s$, involving $A_1$ and $A_2$, whose initial state is $T \mid Q(T)$, and we obtain the system:

$$S' B[\text{fuse}] (z)(s)(B[\text{fuse} \cdot P \mid Q] \mid s[T \mid Q(T)] \mid B[\downarrow_x A_3 \text{ says } B[\text{bool}] \mid A_1[\ldots] \mid A_2[\ldots] \mid A_3[\ldots] \mid A_3[\ldots] \mid A_3[\ldots])$$

Note that $A_3$’s contract remains latent, and thus it may be fused later on.
Example 4.5. Going back to our running example, take $c_A$, $c_{B_1}$, $c_{B_2}$ from Section 3.4 and $T_{AB:B_2}$, $S_{AB:B_2}$ from Example 3.4. Consider the following CO$_2$ system, where $P_A$, $P_{B_1}$, $P_{B_2}$ are given in Example 3.6.

$$S = A[(x)(b_1, b_2) \text{tell}_A \downarrow_x c_A \cdot \text{fuse} \cdot P_A] | B_1[(y)(a, b'_2) \text{tell}_B \downarrow_y c_{B_1} \cdot P_{B_1}] | B_2[(z)(a, b'_1) \text{tell}_A \downarrow_z c_{B_2} \cdot P_{B_2}]$$

$$\rightarrow \rightarrow$$

$$(x, y, z)(a, a', b_1, b'_1, b_2, b'_2)\{A[\text{fuse} \cdot P_A] | B_1[P_{B_1}] | B_2[P_{B_2}]A[\downarrow_x A \text{ says } c_A | \downarrow_y B_1 \text{ says } c_{B_1} | \downarrow_z B_2 \text{ says } c_{B_2}]\}$$

$$(s)S' = (s)\{A[P_3 \sigma \pi] | s[T_{AB:B_2} | Q(T_{AB:B_2})] | B_1[P_{B_1} \sigma \pi] | B_2[P_{B_2} \sigma \pi]\}
$$

where $\sigma = \{x/y, z\}$ (s fresh) and $\pi = \{a/a', b_1/b_1', b_2/b_2, b'_2\}$

System $S$ is the one already considered in Section 3.4, where all the participants are willing to advertise their respective contracts to the store $A$, by using a $\text{tell}$ primitive. Firing those prefixes (first 3 steps of the computation above) has the effect of creating corresponding latent contracts, collected by $A$. At the 4th step, these contracts are $\text{fused}$. Given $\sigma$ and $\pi$ as above, Definition 4.3 is indeed applicable: the domains of $\sigma$ and $\pi$ comply with the definition premises, and as shown in Example 3.4, a system $T_{AB:B_2}$ consisting of contracts $c_A \pi$, $c_{B_1} \pi$ and $c_{B_2} \pi$ may be assigned a global type $S_{AB:B_2}$. Hence, a new session $s$ is created, based on $T_{AB:B_2}$, plus the queues connecting all pairs of participants. The session variables of the latent contracts being $\text{fused}$ (i.e., $x$ for participant $A$, $y$ for $B_1$, and $z$ for $B_2$) are replaced by the fresh session name $s$ in the processes $P_A$, $P_{B_1}$, and $P_{B_2}$, via $\sigma$. Similarly for participant variables which are replaced by participant names, via $\pi$.

The next example illustrates a case where advertising a contract with both participant names and variables allows for enforcing specific properties on new sessions — in particular, making sure that a certain participant will be involved.

Example 4.6. A seller $S$ wants to sell an item to a specific buyer $B$, via any shipping company that provides a package tracking system. The contract of $S$ is:

$$B_1\text{price;} (B_2?\text{pay;} c_2\text{request;} c_2?\text{tracking;} B_1\text{tracking} + B_2?\text{cancel;} c_2\text{cancel})$$

saying that the seller $S$ promises to send a price to the buyer $B$. Then, $B$ can either pay, or cancel the transaction: in the first case, $S$ must send a shipping request to a (still unknown) shipping company $c$, who in turn must send back a tracking number, which is then forwarded to $B$; otherwise, if $B$ cancels, then $S$ will also cancel the transaction with $c$. This contract admits an agreement only if the role of the buyer is played by $B$; the role of shipping company $c$, instead, may be played by any participant offering a compliant contract (such as $A_1?\text{request;} A_1!\text{tracking} + A_1!\text{cancel}$).

Example 4.7 shows how a participant can use participant variables to enforce some property along different sessions — in particular, to ensure the presence of some participant that was involved in a previous agreement.

Example 4.7. A seller $S$ wants to find a shipping company; then, $S$ wants to use that very same shipping company for every new transaction with some buyer.

The shipping company can be found with a “service discovery” contract such as $c_D = c!\text{find}_\text{ship}$: when a session is established on such a contract, $c$ will be replaced with the name of the participant playing the role of the shipper. For each selling transaction, $S$ advertises a contract of the form $c_S = b_1\text{price;} \ldots c_2\text{address}$, where the seller sends a price to a (still unknown) buyer $b$, and after some interaction (here omitted) it sends a shipping address to the shipping company $c$.

The seller can be implemented with the following CO$_2$ process, where $K$ is a contract broker:

$$P = (x(c) \text{ tell}_K \downarrow_x c_D . \text{do}_x^c \text{find}_\text{ship} . X(c)$$

where $X(c) := (y(b) \text{ tell}_K \downarrow_y c_S \ldots X(c)\} (c_S\text{-related actions omitted})$

Note that $c_S$ may be advertised multiple times, using different session variables, inside the recursive process $X$.

An extended version of this example is available in the appendix [17].
4.4. On Culpability

In this section we formalise the notion of culpability in CO\textsubscript{2} systems. Intuitively, a CO\textsubscript{2} participant is “culpable” when it is expected to send or receive some message in a session — and may exculpate itself by performing the required interaction.

We begin by noticing that, according to the CO\textsubscript{2} semantics, each do prefix exposed by a participant, say \(A[P]\), is driven by the contract that \(A\) promised to abide by. In a sense, CO\textsubscript{2} is “culpability-driven”, according to Definition 4.8 below: when a participant is culpable, it has the duty of making the session progress according to its contract. The notion of culpability in CO\textsubscript{2} systems is an extension of the one on contracts (Definition 3.10), covering established sessions.

**Definition 4.8 (Process culpability).** We say that \(A\) is culpable at \(s\) in \(S\) iff \(S \equiv s[T] \mid S'\) and \(A\) is culpable in \(T\).

We say that \(A\) is culpable in \(S\), for some \(s\) and \(u\), \(S \equiv (u)S_0\) and \(A\) is culpable at \(s\) in \(S_0\).

According to Definition 4.8, when e.g. \(A\) is culpable at \(s\) in \(S\), then necessarily \(s \in \text{fv}(S)\); furthermore, \(A\) is also culpable in \((s)S\) (i.e., culpability is preserved when \(s\) is closed and the corresponding session is hidden).

**Example 4.9.** Consider the system \(S'\) from Example 4.5, having the form \(S' \equiv s[T] \mid \cdots\) where \(T = T_{AB}\). We have that, by Definition 3.10, both \(B_1\) and \(B_2\) are culpable in \(T\): in fact, \(T \ rightsquigarrow (B_1 \equiv \text{req})\) and \(T \ rightsquigarrow (B_2 \equiv \text{req})\). Therefore, by Definition 4.8, we also have that \(B_1\) and \(B_2\) are culpable at \(s\) in \(S'\); also, we have that \(B_1\) and \(B_2\) are culpable both in \(S'\) and in \((s)S'\).

A culpable participant can overcome its culpability status by firing its do prefixes, according to [CO\textsubscript{2}-Do], until another participant becomes culpable or session \(s\) terminates. Hence, as long as a culpable participant \(A\) does not enable a do-prefix matching a contractual obligation, \(A\) will remain culpable. Note that when a participant is involved in multiple sessions, it may result culpable in more than one of them.

**Theorem 4.10 (Presence of culpability in CO\textsubscript{2} systems).** Given a CO\textsubscript{2} system \(S\), whenever \(S \rightsquigarrow^* (s)S'\) such that \(S' \equiv s[T] \mid \cdots\) with \(T \neq 0\), then there exists at least one culpable participant at \(s\) in \(S'\).

**Proof.** By Theorem 3.12, we have that there exists at least one culpable participant in \(T\) — and by Definition 4.8 such a participant is also culpable at \(s\) in \(S'\).

Theorem 4.10 says that in an active session established through a fuse reduction, there is always at least one participant \(A[P]\) who leads the next interaction. Thus, if a corresponding do\textsubscript{e} prefix is not in \(P\), \(S\) may get stuck, and \(A\) is culpable.

4.5. Finer-grained Forms of Agreement

In our framework, the participants firing fuse prefixes are playing the role of contract brokers. In some situations, they may want to guarantee some additional requirements before establishing a new session, in order to obtain finer levels of agreement.

In this section, we discuss some scenarios which can be addressed with simple fuse variations. The first one addresses the (possible) issue of participants playing multiple roles in a contract. The other examples add constraints to the synthesised choreography, in order to enforce global properties on sessions.

**Example 4.11 (Single-role enforcement).** Consider the last example of Section 3. It shows a participant \(B_{12}\) that, by advertising the contract \(\psi_{B_{12}}\), plays the roles of two buyers in the contract advertised by the seller \(A\). This is allowed since Definition 3.3 is quite liberal with respect to the substitution \(\pi\) mapping participant variables to participant names. However, \(A\) may want to ensure that the two buyers are actually distinct participants: i.e., \(\pi\) must not map two participant variables to the same participant name. Since \(A\) is also a contract broker, it could enforce this requirement through its fuse prefixes, e.g., by adding the following condition to Definition 3.3:

\[
\forall i \in I: \forall a, b \in \text{fv}(c_i): a \neq b \implies \pi(a) \neq \pi(b)
\]

With this restriction in place, \(A\) disregards the possible agreement involving \(\psi_{B_{12}}\).
Using the same approach as in Example 4.11 (i.e., extending the requirements of Definition 4.3) we can parameterise fuse with a predicate \( \phi \) on global types. Such a predicate is checked on the global type \( G \) (synthesised from the set of latent contracts): when \( \phi(G) \) holds for all \( G \), we have the standard fuse behaviour described in Section 4.3; in general, fuse\( ^\phi \) can enforce global properties emerging from the composition of individual contracts, as illustrated in Examples 4.12 and 4.13 below.

Example 4.12 (Avoiding infinite sessions). A contract broker wants to start new sessions only if they are guaranteed to terminate within a finite number of interactions. This is useful e.g. when a system has a short life expectancy due to an approaching scheduled maintenance, and the brokers are instructed to put limitations on new sessions.

We obtain such a behaviour by letting \( \phi(G) \iff \text{bv}(G) = \emptyset \), where \( \text{bv}(G) \) is the set of bound variables in \( G \). When \( \phi(G) \) holds, we have that new sessions will be created only if there are no recursion variables in \( G \) — and therefore, the choreography does not admit recursive behaviours.

Note that this property cannot be decided by looking at the individual contracts, only. For instance, consider:

\[
A(\text{Ble}) | B(\mu \mathbf{x}. (A?e + A?e';\mathbf{x}))
\]

Here, B’s contract is recursive; however, A’s contract always chooses its non-recursive branch. Indeed, when such contracts are composed, their global type is \( A \rightarrow B \cdot e \), i.e., a non-recursive choreography.

Example 4.13 (Requiring infinite sessions). A contract broker wants to start only non-terminating sessions, where the participants can perpetually interact. To do that, we define \( \phi(G) \) so that it holds only if \( 0 \) is not a sub-term of \( G \). This ensures that the choreography underlying the session does not admit terminating behaviours.

As in Example 4.12, this property needs to be checked at the global (choreography) level; otherwise, contracts with a terminating branch would always be discarded, even when composed with other contracts that always select recursive branches.

In Section 7 we present a case study which takes advantage of the parameterised fuse primitive.

5. Domesticating Wild Choreographies via Honesty

In this section we discuss the notion of honesty [11], providing several examples.

Intuitively, a participant is honest when it is always able to fulfil its contractual obligations, in any context, no matter what other participants do. This is a desirable property in a distributed contract-oriented scenario: in the simplest case, honesty ensures that a participant will never violate its own contracts due to “trivial” bugs, e.g. missing do prefixes (see Example 5.4). In more complex cases, i.e., when a participant A handles multiple sessions simultaneously, honesty ensures that the erratic behaviour of some participants in a session will not cause A to remain stuck, and unable to fulfill its obligations in other sessions (see Example 6.4).

Honesty is especially important when contract-based interactions have some economic relevance: a non-honest (but not necessarily malicious) participant may be unable to reach its goals (i.e., make its contract progress), or may damage other participants if its session obligations remain unfulfilled. Since the state of CO session allows to single out which participants are responsible for session advancement (by Definition 3.10 and Theorem 4.10), such a state could be used as “proof” for sanctioning contractual violations, possibly in an automatic way. In this scenario, a non-honest participant A may be vulnerable to attacks: other participants may induce a situation in which A cannot fulfil its contract, causing it to be punished. For all these reasons, before deploying a service, its developers might want to ensure that its implementation is honest.

Formally, the definition of honesty relies on that of contract ready sets (Definition 5.1) and process ready sets (Definition 5.3).
Definition 5.1 (Contract Ready Sets). For a contract \( c \), we define the set \( \text{CRS}(c) \) as follows:

\[
\text{CRS}(c) = \begin{cases} 
\text{CRS}(c') & \text{if } c = \mu x.c' \\
\{(A, e_i) \mid i \in I\} & \text{if } c = \sum_{i \in I} A?e_i; c_i \\
\{(A_i, e_i) \mid i \in I\} & \text{if } c = \bigoplus_{i \in I} A_i!e_i; c_i \text{ and } I \neq \emptyset 
\end{cases}
\]

and we call each element of \( \text{CRS}(c) \) a contract ready set of \( c \).

Intuitively, contract ready sets represent the immediate obligations in a contract: if \( A \) advertises \( c \), then \( A \)'s process should also offer the interaction capabilities of one of the ready sets in \( \text{CRS}(c) \). Each interaction is a pair \((A, e)\), where \( A \) is the other party involved in the interaction, and \( e \) is the sort of the exchanged message. An external choice results in a single ready set, meaning that \( A \) is expected to offer all the interactions therein; instead, an internal choice results in one singleton ready set for each branch, meaning that \( A \) should pick one of its choices and offer the corresponding interaction (cf. Example 5.2 below). Note that Definition 5.1 above extends the notion of ready set in [11] (where only bilateral sessions are considered) to suit our multiparty contract model.

Example 5.2. Consider the system of contracts \( T_{AB;B_2} \) from Example 3.7 and in particular, the stipulated contracts therein, using the substitution \( \pi = \{A'/a', B_1'/b_1, B_2'/b_2, B_2''/b_2''\} \) from Example 4.5:

\[
\begin{align*}
\tilde{c}_A &= c_A \pi = B_1'?\text{req}; B_2'?\text{req}; B_1'!\text{quote}; (B_1'?\text{order}; B_2'!\text{ok} + B_1'?\text{bye}; B_2'!\text{bye}) \\
\tilde{c}_{B_1} &= c_{B_1} \pi = A!\text{req}; A'?\text{quote}; (B_2'!\text{ok}; A!\text{order} \oplus B_2'!\text{bye}; A!\text{bye}) \\
\tilde{c}_{B_2} &= c_{B_2} \pi = A!\text{req}; (B_1'?\text{ok}; A'?\text{ok} + B_1'?\text{bye}; A'?\text{bye})
\end{align*}
\]

We have \( \text{CRS}(\tilde{c}_A) = \{|(B_1, \text{req})|\} \): in other words, at this point of the contract, an interaction is expected between \( A \) and \( B_1 \) (since \( A \) is waiting for \( \text{req} \)), while no interaction is expected between \( A \) and \( B_2 \). Furthermore, since we just have a single ready set, \( A \) does not have other interactions to choose from.

Let us now equip \( T_{AB;B_2} \) with one queue between each pair of participants, and let it perform the request exchange between \( B_1 \) and \( A \), with the transitions:

\[
T_{AB;B_2} | Q(T_{AB;B_2}) \xrightarrow{B_1 \rightarrow A!\text{req}} A \leftarrow B_1!\text{req} \xrightarrow{T'_{AB;B_2}} T_{AB;B_2} | Q(T_{AB;B_2})
\]

We have that \( \tilde{c}_A \) in \( T'_{AB;B_2} \) is now reduced to:

\[
\tilde{c}_A' = B_2'?\text{req}; B_1'!\text{quote}; (B_1'?\text{order}; B_2'!\text{ok} + B_1'?\text{bye}; B_2'!\text{bye})
\]

and thus we have \( \text{CRS}(\tilde{c}_A') = \{|(B_2, \text{req})|\} \), i.e., \( A \) is now waiting for a request from \( B_2 \).

If we let the system reduce further, \( \tilde{c}_A' \) reaches its external choice:

\[
\tilde{c}_A'' = B_1'?\text{order}; B_2'!\text{ok} + B_1'?\text{bye}; B_2'!\text{bye}
\]

We now have a single ready set with two elements: \( \text{CRS}(\tilde{c}_A'') = \{|(B_1, \text{order}), (B_1, \text{bye})|\} \), i.e., \( A \) must handle both answers from \( B_2 \). Instead, when \( \tilde{c}_B \) reduces to its internal choice, we have:

\[
\tilde{c}_B'' = B_2'!\text{order} \oplus B_2'!\text{bye}; A!\text{bye}
\]

We now have two ready sets, each one containing one interaction: \( \text{CRS}(\tilde{c}_B'') = \{|(B_2, \text{ok})|, (B_2, \text{bye})|\} \).

Example 5.2 shows that, when a contract \( c \) of a participant \( A \) evolves within a system \( T \), its ready sets change accordingly. Now we need to define the counterpart of contract ready sets for CO2 processes, i.e., the process ready sets.
Definition 5.3 (Process Ready Set). For all CO\textsubscript{2} systems \(S\), all participants \(A\) and sessions \(s\), we define the set of pairs\footnote{The side condition \(\exists s \not\in \vec{v}\) of Definition 5.3 deals with cases like \(S_0 = (s)(A[\text{do}^\ast\text{int}]\) and \(S = S_0 \uparrow (A[\text{Bint}] \uparrow \ldots \uparrow \ldots)\). Without the side condition, \(\text{PRS}_A^\ast(S_0) = \{(B, e)\} \) — hence, by Definition 5.3 \(A\) would result to be ready in \(S\).}:

\[
\text{PRS}_A^\ast(S) = \{(B, e) \mid \exists \vec{v}, \vec{a} : S \equiv (\vec{v}, \vec{a}) (A[\text{do}^\ast\vec{e} \ldots + \ldots | \ldots \ldots)] \land s \not\in \vec{v}\}
\]

Intuitively, Definition 5.3 says that the process ready set of \(A\) over a session \(s\) in a system \(S\) contains the interactions that \(A\) is immediately willing to perform with other participants through its \(\text{do}^\ast\) prefixes. Just like contract ready sets, the interactions are represented by participant/sort pairs. Definition 5.3 is adapted from \cite{IS} to suit our multiparty contract model.

Next, we want to characterise a weaker notion of the process ready set, so that it only takes into account the first actions on a specific session that a participant is willing to make.

Definition 5.4 (Weak Process Ready Set). We write \(S \not\equiv (A : \text{do}^\ast) \rightarrow S'\) iff:

\[
\exists B, p : S \xrightarrow{B, p} S' \land (A \not\equiv B \lor (\forall v \forall C : p = \text{do}^v C \implies s \not\equiv t))
\]

We then define the set of pairs \(\text{WPRS}_A^\ast(S)\) as:

\[
\text{WPRS}_A^\ast(S) = \left\{(B, e) \mid \exists S' : S \not\equiv (A : \text{do}^\ast) \rightarrow S' \land (B, e) \in \text{PRS}_A^\ast(S')\right\}
\]

In Definition 5.4, the transition relation \(\not\equiv (A : \text{do}^\ast) \rightarrow\) abstracts from the reductions of \(S\) which are not due to \(A\)'s actions on session \(s\). In other words, such a relation allows a system \(S\) with a participant \(A[S]\) to reduce freely, until a reducible \(S'\) containing \(A[S]\) is reached, such that a \(\text{do}^\ast\) prefix is at top-level in \(S'\). Note that, since Definition 5.4 builds upon Definition 5.3, if \(s \not\in \text{fv}(S)\), then necessarily \(\text{WPRS}_A^\ast(S) = \emptyset\); i.e., observing the weak process ready set of a session \(s\) is only meaningful after \(s\) has been established, and it is not delimited (and thus it cannot be \(\alpha\)-converted).

We now introduce the final ingredient required to define honesty, that is the \textit{readiness} of a participant.

Definition 5.5 (Readiness). We say that \(A\) is ready in \(S\) iff, whenever \(S \equiv ([\vec{v}](\vec{B})S_0\) for some \(\vec{v}, \vec{B}\) and \(S_0 \equiv s[A(c) \mid \ldots \mid \ldots \ldots]\), the following holds:

\[
\exists \chi \in \text{CRS}(c) : \chi \subseteq \text{WPRS}_A^\ast(S_0)
\]

Definition 5.5 says that a participant \(A\) is \textit{ready} in a system \(S\) when, for all its established sessions \(s\), \(S\) can reduce in such a way that \(A\)'s process will provide at \(s\) all the interactions expected by one of the ready sets of \(A\)'s contract at \(s\). In other words, when \(A\) is "ready", then, for all its contracts \(c\) already involved in a session, \(A\) has a way to fulfil at least its immediate obligations in \(c\).

Example 5.6. Consider the system \(S\) in Example 4.5. We have seen that, after its latent contracts have been fused, we obtain:

\[
(s)S' \equiv (s)(A[P_A\pi] \mid s[T_{AB_1B_2} \mid Q(T_{AB_1B_2})] \mid B_1[P_{B_1}\sigma\pi] \mid B_2[P_{B_2}\sigma\pi])
\]

where the substitutions \(\sigma\) and \(\pi\) are also from Example 4.5. Let us define the processes (after substitutions):

\[
\begin{align*}
P_A\pi &= \text{do}^\ast\pi \text{ req} . \text{do}^\ast\pi \text{ req} . \text{do}^\ast\pi \text{ quote} . (\text{do}^\ast\pi \text{ order} . \text{do}^\ast\pi \text{ ok} + \text{do}^\ast\pi \text{ bye} . \text{do}^\ast\pi \text{ bye}) \\
P_{B_1}\sigma\pi &= \tau . \text{do}^\ast\pi \text{ req} . \text{do}^\ast\pi \text{ quote} . \text{do}^\ast\pi \text{ order} \\
P_{B_2}\sigma\pi &= \text{do}^\ast\pi \text{ req} . (\text{do}^\ast\pi \text{ ok} . \text{do}^\ast\pi \text{ ok} + \text{do}^\ast\pi \text{ bye} . \text{do}^\ast\pi \text{ bye})
\end{align*}
\]
We have:

\[
\begin{align*}
\text{PRS}_A^\tau(S') & = \{(B_1, \text{req})\} = \text{WPRS}_A^\tau(S') \\
\text{PRS}_B^\tau(S') & = \varnothing \neq \{(A, \text{req})\} = \text{WPRS}_B^\tau(S') \\
\text{PRS}_B(S') & = \{(A, \text{req})\} = \text{WPRS}_B(S')
\end{align*}
\]

Note that the \( \tau \) prefix in \( P_B \) prevents \( B_1 \) from interacting immediately with \( A \) on session \( s \), although it is “weakly ready” to do so. Hence, considering that the weak process ready sets of each participant in \( S'_1 \) match their respective contract ready sets in \( T_{AB_1B_2} \) (Example 5.2) according to Definition 5.2, we have that participants \( A, B_1 \) and \( B_2 \) are all ready in \((s)S'_1\).

Before defining honesty formally, we need to characterise the class of systems for which this concept is meaningful, — i.e., those systems where a participant is not (yet) involved in latent contracts nor active sessions, and whose process has no free variables.

**Definition 5.7 (Initial System).** A COI system \( S \) is \( \Lambda \)-initial if \( S \) has no sub-term of the form \( \downarrow A \) says \( c \) or \( \Lambda(c) \) with \( c \neq 0 \), and whenever \( S \) has a subterm of the form \( \Lambda[P] \), then \( P \) is closed. A COI system \( S \) is initial when it is \( \Lambda \)-initial for all participants \( \Lambda \) in \( S \).

**Definition 5.8 (Honesty).** We say that \( \Lambda[P] \) is honest iff, for all \( \Lambda \)-initial \( S \equiv \Lambda[P] \mid S_0 \mid S \rightarrow^* S' \), \( \Lambda \) is ready in \( S' \).

A process \( \Lambda[P] \) is said to be honest when, for all contexts and reductions that \( \Lambda[P] \) may be engaged in, \( \Lambda \) is persistently ready. In other words, for all context \( S_0 \) with which \( \Lambda[P] \) may be composed, and for all systems \( S' \) resulting from the continuation of such a composition, there must be a correspondence between the interactions required by \( \Lambda \)'s stipulated contracts and \( \Lambda \)'s (weak) process ready set. The definition requires an \( \Lambda \)-initial system to rule out contexts with latent/stipulated contracts of \( \Lambda \) — otherwise, \( \Lambda \) could be made trivially dishonest, e.g. by inserting a latent contract \( \downarrow \Lambda \) says \( c \) that \( \Lambda \) cannot fulfil. Notice that \( \Lambda[P] \) is trivially honest when \( P \) advertises no contracts.

**Example 5.9.** Consider the process \( B_1[^{\uparrow} \Lambda \downarrow \text{cy}] \cdot P_{B_1} \) of system \( S \), as defined in Examples 5.3 and 5.6. We show that this process is not honest. In fact, \( S \) can reduce as \( S \rightarrow^* (s)S' \rightarrow^* (s)S'' \), where:

\[
(s)S'' = (s)(A[do^*_B \text{order}.do^*_B \text{ok} + do^*_B \text{bye}.do^*_B \text{bye}]
| s[A(B_1[^{\uparrow} \Lambda \downarrow \text{order}]B_2[^{\uparrow} \Lambda \downarrow \text{ok}] \downarrow \text{cy} B_2[^{\uparrow} \Lambda \downarrow \text{bye}] B_2[^{\uparrow} \Lambda \downarrow \text{bye}] \\
| B_1(B_2[^{\uparrow} \Lambda \downarrow \text{ok}]A[^{\uparrow} \Lambda \downarrow \text{order}]B_2[^{\uparrow} \Lambda \downarrow \text{bye}]A[^{\uparrow} \Lambda \downarrow \text{bye}]) \\
| B_1[^{\uparrow} \Lambda \downarrow \text{order}] B_2[^{\uparrow} \Lambda \downarrow \text{ok}] do^*_B \text{ok} + do^*_B \text{bye}.do^*_B \text{bye}]
\)

At this point, we see that there is a problem in the implementation of \( B_1 \): it does not notify the other buyer before making an order. In fact, \( B_1 \)'s process is trying to perform \( do^*_B \text{order} \), but its contract requires that \( do^*_B \text{ok} \) is performed first (or \( do^*_B \text{bye} \), if the quote is rejected). This is reflected by the mismatch between \( B_1 \)'s process ready set in \( S'' \), and its contract ready sets, in session \( s \):

\[
\begin{align*}
\text{PRS}_B^\tau(S'') & = \{(A, \text{order})\} \\
\text{CRS}(B_2[^{\uparrow} \Lambda \downarrow \text{order}]B_2[^{\uparrow} \Lambda \downarrow \text{bye}]A[^{\uparrow} \Lambda \downarrow \text{bye}]) & = \{(B_2, \text{ok})\}, \{(B_2, \text{bye})\}
\end{align*}
\]

In terms of the definitions above, we can say that there exists a system \( S \) containing \( B_1[^{\uparrow} \Lambda \downarrow \text{cy}] \cdot P_{B_1} \) — that reduces to a \((s)S''\) where \( B_1 \) is not ready (Definition 5.5). Therefore, \( B_1[^{\uparrow} \Lambda \downarrow \text{cy}] \cdot P_{B_1} \) is not honest. In fact, \( B_1 \) is culpable in \((s)S''\), according to Definition 4.8.

As in [18], the definition of honesty subsumes a fair scheduler, eventually allowing participants to fire persistently (weakly) enabled \text{do} actions.
6. Properties of CO₂ Systems

We now give the main properties of our framework. We start by proving that honest participants can always perform their contractual obligations without relying on the context (Theorem 6.1). Then, we formalise a link between the honesty of participants, and two key properties borrowed from the session types literature: Theorem 6.2 introduces session fidelity in CO₂, while Corollary 6.3 (descending from Theorem 6.1) introduces a notion of progress of CO₂ systems.

Theorem 6.1 (Session progress). Let $S_0$ be an $A$-initial CO₂ system with $A[P_0]$ honest. If $S_0 \rightarrow^* (s)S$, where $S \equiv s[T] \mid A[P] \mid \cdots$ and $A$ is culpable at $s$ in $S$, then:

$$S \xrightarrow{A[p_1]} \cdots \xrightarrow{A[p_n]} \xrightarrow{A: \text{do}^s(e)}$$

for some $B$, $e$, and $p_1, \ldots, p_n$ such that $n \geq 0$, and $p_i \neq \text{fuse}$ for all $i \in 1..n$.

**Proof.** By assumption, $A$ is culpable at $s$ in $S$. By Definition 5.8, $A$ is ready in any reduction of $S_0$ — in particular in $(s)S$. Therefore, by Definition 5.5 there exists $X \in \text{CRS}(e)$ such that $X \subseteq \text{WPRS}_A^1(S)$. From these premises, we have that there must exist some $B, e$ such that $T \xrightarrow{A: \text{do}^s(e)}$ (due to culpability) and $(B, e) \in \text{WPRS}_A^1(S)$ (due to readiness). By Definition 5.4 it means:

$$\exists S': S \not\equiv (A: \text{do}^s(e))^* \cdot S' \land (B, e) \in \text{PRS}_A^1(S')$$

(6.1)

and therefore, by Definition 5.3 we must have that $A[\text{do}^s(e) \cdot P' + P'' \mid Q]$ is a participant of $S'$. Thus, we have to show that $A[P]$ in $S$ may reduce to such a process, with $\text{do}^s(e)$ enabled, by firing its own prefixes $p_1, \ldots, p_n$ only — except fuse.

By contradiction, assume that none of the traces from $S$ to $S'$ (following (6.1)) satisfies the thesis. We show that, in this case, $A[P_0]$ would not be honest: in fact, there would exist a system $\tilde{S}_0 \equiv A[P_0] \mid \cdots$ with a reduct $(s)\tilde{S}$ such that $\tilde{S} \equiv s[T] \mid A[P] \mid \cdots$, where $A$ would not be ready. Our goal is to create such a system $\tilde{S}_0$.

Before proceeding, we define some auxiliary functions.

Given two CO₂ systems $S_A, S_B$ such that there exists a transition $S_A \xrightarrow{\ell} S_B$, $\text{inLab}(S_A, S_B)$ gives the “innermost” label in the derivation inducing such a transition. Intuitively, it pierces through delimitations, and returns the underlying CO₂ prefix before it is turned into $\tau$ by an application of rule [CO₂-Del]:

$$\text{inLab}(S_A, S_B) = \begin{cases} \ell_0 & \text{if} \ \ [\text{R}] \ \ S_A \xrightarrow{\ell_0} S_B \ \ \text{and} \ \ [\text{R}] \notin \{[\text{CO₂-Del}], [\text{CO₂-Par}]\} \\ \text{inLab}(S'_A, S'_B) & \text{if} \ \ [\text{R}] \ \ S_A \xrightarrow{\ell_0} S_B \ \ \text{and} \ \ [\text{R}] \in \{[\text{CO₂-Del}], [\text{CO₂-Par}]\} \end{cases}$$

Given a sequence of CO₂ systems $\tilde{S}_0$ such that each element is a reduct of the preceding one, $\text{inTr}(\tilde{S}_0)$ gives the trace composed by the “innermost” labels of each reduction:

$$\text{inTr}(S_A \cdot \epsilon) = \epsilon \quad \text{(where } \epsilon \text{ is the empty sequence)}$$

$$\text{inTr}(S_A \cdot S_B \cdot \tilde{S}_0) = \text{inLab}(S_A, S_B) \cdot \text{inTr}(S_B \cdot \tilde{S}_0)$$

19
Let us now consider an execution going from the A-initial \( S_0 \) to \((s)S\) in the hypotheses, and let \( \tilde{S} \) be the sequence of CO\(_2\) system configurations composing such an execution; furthermore, let us define the “inner” trace \( \tilde{T} = \text{inTr}(\tilde{S}) \).

With this information at hand, we can build \( \tilde{S}_0 \) so that:

1. \( \tilde{S}_0 \equiv A[P_0] | \tilde{S}_0' \);
2. \( \tilde{S}_0' \) contains:
   
   (a) all the sessions appearing in \( S \);
   
   (b) all the latent contracts appearing in \( S_0 \), except those that are not fused along \( \tilde{S} \) and thus appear in \((s)S\);
   
   (c) participants \( B_1[P_1], \ldots, B_n[P_n] \), where \( B_1, \ldots, B_n \notin A \) also appear in \( S \), and \( P_1, \ldots, P_n \) are written so that:
     
     i. all the “innermost” prefixes in \( \tilde{T} \) fired by \( B_1, \ldots, B_n \) can still be fired in the same order — with the exception of the tell prefixes corresponding to latent contracts in \((s)S\), which are neglected;
     
     ii. after the prefixes in \( \tilde{T} \) are fired, all \( B_1[P_1], \ldots, B_n[P_n] \) reduce to \( B_1[0], \ldots, B_n[0] \).

As a result, we have that \( \tilde{S}_0 \) defined above can reduce to a system \((s)\tilde{S} \) such that \( \tilde{S} \equiv s[T] | A[P] | \cdots \) through a “inner” trace \( \tilde{T} \) whose prefixes match those in \( \tilde{T} \), except for the lack of tell prefixes fired by any \( B \neq A \) (by item 2(c)i above). Furthermore, we have that:

1. no latent contracts appear in \( \tilde{S} \) (unless they have been advertised by \( A \));
2. all established sessions in \( S \) also appear in \( \tilde{S} \), and vice versa;
3. except for \( A[P] \), all participants in \( \tilde{S} \) have the form \( B[0] \).

Hence, no participant in \( \tilde{S} \) can fire a fuse prefix, and no participant \( B \neq A \) in \( \tilde{S} \) has any prefix enabled.

Now, recall that we assumed (by contradiction) that Equation [6.1] cannot be satisfied with a trace consisting only of \( A[P] \)'s prefixes (except fuse). Therefore:

\[
\tilde{A}\tilde{S}' : \tilde{S} \overset{\neq A:.do^*}{\longrightarrow} \tilde{S}' \land (B, e) \in \text{PRS}_A^0(\tilde{S}')
\]

But then, there exists an \( A \)-initial system \( \tilde{S}_0 \equiv A[P] | \cdots \) such that \( \tilde{S}_0 \rightarrow^* (s)\tilde{S} \), and \( A \) is not ready in \((s)\tilde{S} \). Therefore, we must conclude that \( A[P] \) is not honest — thus violating the theorem hypothesis. □

Theorem 6.2 below says that each (honest) participant will strictly adhere to its contracts, once they have been fused in a session \( s \): no extra-contractual interactions are possible on \( s \), and honest participants will eventually perform do* actions, as long as it is required by their contracts.

**Theorem 6.2 (Session fidelity).** Given a CO\(_2\) system \( S_0 \) containing \( A[P_0] \), whenever \( S \rightarrow^* (s)S \) with \( S \equiv s[T] | \cdots \), then:

1. \( S \overset{\neq A:.do^*}{\longrightarrow} A:.do^* e \Rightarrow \exists n \geq 0: \begin{array}{c} T \overset{\lambda_1}{\longrightarrow} \cdots \overset{\lambda_n}{\longrightarrow} A=\lambda=\lambda B=\lambda e \end{array} \) where \( \forall C, e', i = 1..n: \lambda_i \neq A \Rightarrow C : e' \);
2. If \( S \) is \( A \)-initial, \( A[P_0] \) honest, and \( Y = \{ (B, e) | T \overset{\lambda=\lambda B=\lambda e}{\longrightarrow} \} \neq \emptyset \), then

\[
\exists (B, e) \in Y \text{ such that } S \overset{\neq A:.do^*}{\longrightarrow} A:.do^* e
\]

Recall that, as in Definition 5.4, \( \neq A:.do^* \) intuitively stands for any reduction not involving \( A \) on session \( s \).
Proof. Item 1 follows directly from the semantics of CO₂ (in particular, rule \([\text{CO}_2-\text{do}]\)) if \(A\) is able to fire a \(\text{do}^e\) prefix, then its contract must necessarily allow for sending/receiving \(e\) to/from \(B\).

Item 2 follows from the honesty hypothesis for \(A[\text{P}_0]\), and from Definition 5.5 (readiness). Let us fix \(n \geq 1\) enumerating all enabled interactions in \(T\) driven by \(A\) — i.e., let \(n\) be the cardinality of \(Y\) in the theorem statement. Let us consider \(e = T(A)\), i.e., \(A\)'s contract in \(T\). Due to \(A\)'s readiness, \(e\) is such that, either:
1. \(\text{CRS}(e) = \{(B_j,e_j)\}_{j=1..n}\); or,
2. \(\text{CRS}(e) = \{(B_j,e_j)\}_{j=1..n}\),

where \(\{(B_j,e_j)\}_{j=1..n} = Y\) — i.e., \(Y\) contains exactly the \((B,e)\) pairs appearing in \(\text{CRS}(e)\). Recall that, since \(S_0\) is \(A\)-initial and \(A\) is honest, then \(A\) must be \emph{ready} in all possible reducts of \(S_0\) — and, in particular, it must be ready in \((s)(S)\). Therefore, we have that \(\exists X \in \text{CRS}(e)\) such that \(S' \not\equiv \frac{\text{CRS}(e)}{A: \text{do}^e, e} \frac{A: \text{do}^e}{X} \frac{A: \text{do}^e}{e} (S')\), i.e.:
\[
\exists X = \{(B,e), \cdots\} \in \text{CRS}(e) \text{ such that } S' \not\equiv \frac{\text{CRS}(e)}{A: \text{do}^e, e} \frac{A: \text{do}^e}{X} \frac{A: \text{do}^e}{e} (S') \quad (6.2)
\]
Since the \((B,e)\) pair satisfying Equation 6.2 also belongs to \(Y\), the thesis holds. □

Note that, by Theorem 6.1, the conclusion of Item 2 of Theorem 6.2 above may be stricter: i.e., the \(\text{do}^e\) transition may be reached by firing \(A\)'s prefixes only (excluding fuse).

Corollary 6.3 below introduces a notion of “global progress” in CO₂. This notion is only meaningful after a session has been established, and thus a culpable participant exists. In fact, a system without sessions may not progress — either because a set of compliant contracts cannot be found, or because a fuse prefix is not enabled. In both cases, no participant may be deemed culpable, and thus responsible for the next move. However, the system could progress again if it is composed with another system consisting of (honest) participants, allowing for a new session to be established.

\textbf{Corollary 6.3.} \textit{Given an initial \text{CO}_2 system  \(S_0\) with only honest participants, if  \(S_0 \Rightarrow^* S \equiv (s)(S[T] | \cdots)\) with  \(T \not= 0\), then  \(S \Rightarrow\).}

\textit{Proof.} By contradiction. From [6], we know that \(T\) is deadlock free, thus if the \text{CO}_2 system cannot make further reductions, it must be because one of the participant \(A\) cannot meet its obligations. This is in contradiction with Theorem 6.1 since all participants are honest in \(S_0\). □

This result also holds for systems where a process takes part in multiple sessions: the honesty of all participants guarantees that all sessions will (eventually) progress.

\textbf{Example 6.4.} We now give a simple example of a system with multiple sessions. We show how a seemingly honest process \(B\) could be deemed culpable due to the unexpected behaviour of other participants, and how honest participants guarantee progress of the whole system. Consider:
\[
S = (x,y,z,w)(\text{A}[\text{tell}_x \downarrow \text{int} (\text{B} \uparrow \text{int}). \text{fuse}. \text{fuse}] \\
| \text{B}[\text{tell}_y \downarrow \text{int} (\text{A} \uparrow \text{int}). \text{tell}_z \downarrow \text{bool} (\text{C} \uparrow \text{bool}). \text{do}^z \text{int}. \text{do}^z \text{bool}] \\
| \text{C}[\text{tell}_w \downarrow \text{bool} (\text{B} \uparrow \text{bool}). \text{do}^w \text{bool}])
\]

After all four contracts have been advertised to \(A\) and fused, the system reduces to:
\[
S' = (s_1, s_2)(\text{A}[0] | \text{B}[\text{do}^{s_1} \text{int}. \text{do}^{s_2} \text{bool}] | \text{C}[\text{do}^{s_2} \text{bool}] \\
| s_1[\text{A}(\text{B} \uparrow \text{int})] | s_2[\text{B}(\text{C} \uparrow \text{bool})] \\
| s_1[\text{A} | s_2[\text{B}])
\]

Even if both sessions \(s_1\) and \(s_2\) enjoy contractual progress, \(S'\) is stuck: \(A\) does not perform the promised action, thus remaining culpable in \(s_1\); \(B\) is stuck waiting in \(s_1\), thus remaining culpable in \(s_2\). Indeed,

\[\text{In this case, } B \text{ is deemed culpable in } s_2 \text{ because its implementation did not expect } A \text{ to misbehave.}\]
neither A nor B are ready in S′, and thus they are not honest in S. Hence, global progress is not guaranteed. Let us now consider the following variant of S, where all participants are honest:

\[ S = (x,y,z,w) (A[tell_{A \downarrow z}(B!\text{int}).do^y_{A}\text{int}] | \text{fuse} | \text{fuse} | C[tell_{A \downarrow w}(B?\text{bool}).do^y_{B}\text{bool}] | B[tell_{A \downarrow y}(A?\text{int}).tell_{A \downarrow z}(C?\text{bool})(do^y_A\text{int}.do^z_B\text{bool}) + \tau.(do^y_A\text{int}|do^z_B\text{bool}))] \]

In this case, A will respect its contractual duties, while B will be ready to fulfil its contracts on both sessions — even if one is not activated, or remains stuck (here, \( \tau \) represents an internal action, e.g., a timeout: if the first do^y_A\text{int} cannot reduce, B falls back to running the two sessions in parallel). The honesty of all participants in \( S \) guarantees that, once a session is active, it will reach its completion.

7. Case Study: Sharing Knowledge

We illustrate the suitability of our framework to reason about distributed systems where contract brokers are willing to offer specific guarantees to their clients. In particular, we illustrate how three key components of our framework, i.e., (i) multiparty sessions, (ii) the notion of honesty, and (iii) contractual agreements based on choreographies, address complex challenges encountered in the design and specification of distributed systems.

We present a case study where a contract broker wants to provide a specific service to its clients: guaranteeing that participants will interact according to a class of gossip protocols [15]. Gossip protocols are used to spread a piece of information among a group of participants, ensuring that each one of them will eventually acquire such information. In particular, we focus on “knowledge sharing” protocols, where each participant owns some initial knowledge, which must be shared across all participants before the participants terminate, i.e., everyone must eventually receive each other’s initial knowledge.

For instance, assuming that \( k \) is a message sort representing the current knowledge of a participant, the system of contracts on the left below implements a sharing knowledge protocol, while the one on the right does not. Indeed, on the right-hand side, A does not acquire B’s knowledge.

\[
A(B!k;B?k) \mid B(A?k;A!k) \quad \checkmark \quad A(B!k) \mid B(A?k) \quad \times
\]

We distinguish three challenges that one needs to address to ensure that a broker provides effectively a knowledge sharing service. (a) Intuitively, for a set of participants to share knowledge, they may interact either via a single (multiparty) session or via several binary sessions. However, guaranteeing safety and liveness properties across several, possibly interleaved, sessions is a notoriously difficult problem (see [19] for instance) and it is even harder to check for semantic properties such as knowledge sharing. Instead, intra-sessions properties are much easier to attain [3]; and in our case a broker is able to take a well-informed decision before starting a session by analysing its choreography. (b) In real world applications, e.g., in cloud computing [20], one cannot assume that each component will behave as expected. In our framework, it is always possible to discover which participants are not fulfilling their contract (culpability); and the honesty of a participant guarantees that it will always fulfil its contract. Therefore, if an honest participant joins a knowledge sharing session, it will always share all its knowledge. (c) Properties of any contract model that offers a good level of expressiveness (e.g., one based on FIFO queues like ours) take the risk of being hard to analyse (or even undecidable [21]). In such a case, one needs to find sound and efficient ways to decide whether a given set of contracts satisfy certain properties. Below, we give an effective decision procedure that guarantees knowledge sharing of a set of contracts by parsing only a global type.

We begin this case study by introducing a formal model for transmission and acquisition of knowledge (Section 7.1); then we show how knowledge is transmitted/acquired in systems of contracts and CO2 systems (Section 7.2). Finally, in Section 7.3 we introduce a variant of the parameterised fuse primitive (cf. Section 4.5) which guarantees that new sessions satisfy the knowledge sharing property. For the sake of readability, some auxiliary technical details are relegated to in the appendix [17].
7.1. Semantic knowledge sharing

We introduce a formalization of knowledge transmission and acquisition among a set of participants through labelled transition systems (LTS), which may be induced by both system of contracts and CO2 systems.

Definition 7.1 (Knowledge of a participant). The knowledge of a participant is a pair \((A, \Omega)\), where \(A\) (acquired knowledge) is a set of participant names, and \(\Omega\) (pending knowledge) is a map from participant names to queues of the form \(A' : A'' : \ldots\), where \(A', A''\) are sets of participant names. We write \(\emptyset\) when \(\Omega\) maps each participant to the empty queue.

If the knowledge of \(B\) is \((\{B\}, \emptyset)\), then \(B\) has only its own initial knowledge. The knowledge of \(B\) may increase as it interacts with other participant. For instance, if the knowledge of \(B\) is \((\{B, E\}, \{A: \{A\}, C: \{C, D\}\})\); then \(B\) has acquired \(E\)'s initial knowledge, while some knowledge is pending, i.e., the initial knowledge of \(A\) (transmitted by \(A\) itself) and the initial knowledge of \(C\), followed by \(C\)'s initial knowledge and \(D\)'s initial knowledge.

We introduce an LTS for knowledge transmission/acquisition in Definition 7.2 below. We use two operators: \(\text{push}_B((A, \Omega), (A', \Omega'))\) which augments the knowledge of a participant with the knowledge transmitted by \(B\), i.e., the knowledge \(A'\) sent by \(B\) is appended to \(\Omega\); and \(\text{pop}_B((A, \Omega))\) moves the first pending knowledge sent by \(B\) to the acquired knowledge \(A\). For instance, we have:

\[
\text{push}_B((\{A\}, \{C: \{C\}, B: \{B\}\}), (\{A, B\}, \{D: \{D\}\})) = (\{A\}, \{C: \{C\}, B: \{B, A\}\})
\]

\[
\text{pop}_B((\{A\}, \{B: \{B\}, C: \{C\}\})) = (\{A, B\}, \{C: \{C\}\})
\]

Observe that push ignores the pending knowledge \(\Omega'\) of the second parameter. The formal definition of these operators is given in the appendix [17].

Definition 7.2 (Knowledge LTS). An LTS \(L = (\Sigma \times \Delta, \Lambda, \rightarrow, k^\delta_0)\) is a knowledge LTS (abbreviated KLTS) iff:

1. \(\Sigma\) is a set of total mappings from participants names to their respective knowledge (i.e., \((A, \Omega)\) pairs by Definition 7.1);

2. \(\Delta\) is any set, called decorating set, with elements ranged over by \(\delta, \delta'\), etc. For brevity, we write \(k^\delta\) instead of \((k, \delta) \in \Sigma \times \Delta\);

3. \(k^\delta_0\) is the initial state, such that for all \(A \in \mathbb{P} : k^\delta_0(A) = (\{A\}, \emptyset);

4. \(\Lambda\) contains labels of the form \(B!A, A?B, \tau\);

5. the labelled transition relation \(\rightarrow \subseteq (\Sigma \times \Delta) \times \Lambda \times (\Sigma \times \Delta)\) has the following properties, for all \(k^\delta_1, k^\delta_2 \in (\Sigma \times \Delta):

   (a) \(k^\delta_1 \xrightarrow{B!A} k^\delta_2 \iff k_2 = k_1[\text{push}^B(k_1(A), k_1(B))/\delta]\);

   (b) \(k^\delta_1 \xrightarrow{A?B} k^\delta_2 \iff k_3 = k_1[\text{pop}^A(k_1(A))/\delta]\);

   (c) \(k^\delta_1 \xrightarrow{\tau} k^\delta_2 \iff k_1 = k_2\).

Item \(\square\) says that each state of a KLTS maps a participant to its current knowledge, e.g., \(k^\delta(A) = (\{A, B\}, \{D: \{C, D\}\})\). The decorations in item \(\square\) allows to distinguish states with a same knowledge mapping. By item \(\square\) each participant has only its own knowledge in the initial state. The transition labels in \(\square\) represent either knowledge transmission \((B!A)\), knowledge acquisition \((A?B)\), or and silent moves \(\tau\). Transitions and knowledge mapping are related as follows

- by \(\square\), when \(B\) sends its knowledge to \(A\), \(A\)'s knowledge is updated by pushing \(B\)'s knowledge to the corresponding (pending) knowledge queue.
• by \( [5b] \), when \( A \) acquires the knowledge from \( B \), \( A \)'s knowledge is updated by dequeuing the first element of the knowledge transmitted by \( B \);

• by \( [5c] \), silent moves have no effects on \( k(\_). \)

Hereafter, we often write \( k, k_1, \ldots \) for KLTS states (omitting decorations \( \delta \) when unimportant).

**Definition 7.3 (Knowledge sharing).** Let \( L = (\Sigma, A, \rightarrow, k_0) \) be a KLTS. We say that a state \( k^\delta \in \Sigma \times \Delta \) shares knowledge among participants \( A \subseteq P \) iff, for all \( B \in A, \exists A' \) s.t. \( k(B) = (A', \Omega) \) and \( A \subseteq A' \).

We say that \( L \) shares knowledge among \( A \) iff, whenever \( k_0^\delta \rightarrow^* k_1^\delta \), then there exists \( k_2^\delta' \) such that \( k_1^\delta' \rightarrow^* k_2^\delta' \) and \( k_2^\delta' \) shares knowledge among \( A \).

A KLTS state \( k^\delta \) shares knowledge among a set of participants \( A \) when, in this state, each participant has acquired the knowledge of everybody else in \( A \). \( L \) shares knowledge among \( A \) iff all traces in a KLTS \( L \) eventually reach a state that shares knowledge among \( A \).

**Example 7.4.** Let \( L_1 \) be a KLTS with a unique trace \( k_0 \xrightarrow{A} k_1 \xrightarrow{B} k_2: L_1 \) does not share knowledge among \( A \) and \( B \). Let \( L_2 \) be a KLTS with a unique trace \( k_0 \xrightarrow{A} k_1 \xrightarrow{B} k_2 \xrightarrow{A} k_3 \xrightarrow{A} k_4: L_2 \) shares knowledge among \( A \) and \( B \).

### 7.2. Knowledge sharing in contracts and CO2

A KLTS \( L^T \) can be easily induced from the semantics of a system of contracts \( T \). Given a system of contracts \( T \), whenever \( A \in P(T) \) receives the special “knowledge message” \( k \) from \( B \in P(T) \), the knowledge of \( A \) is augmented with that of \( B \). Subsequently, if \( A \) sends \( k \) to \( C \in P(T) \), then \( C \) will receive the knowledge of both \( A \) and \( B \). Technically, each time \( k \) is sent (resp. received) by a participant in \( T \), a corresponding transition is induced in \( L^T \), so that knowledge is transmitted (resp. acquired). Instead, if \( T \) reduces by exchanging a message sort \( e \neq k \), then a \( \tau \) transition is induced in \( L^T \). Each configuration of the system of contracts is used as decoration in \( L^T \), so that the LTS of \( T \) and \( L^T \) are isomorphic.

**Example 7.5.** Let \( T_1 = A(B!k) | B(A?k) \). \( L^T_1 \) does not share knowledge among \( A \) and \( B \), and thus \( T_1 \) does not share knowledge among \( A \) and \( B \). Instead, \( T_2 = A(B!k;B?k) | B(A?k;A!k) \) shares knowledge among \( A \) and \( B \).

Analogously, a KLTS \( L^S \) can be induced from the semantics of a CO2 system \( S \). Intuitively, a participant \( A \) in \( S \) transmits (resp. acquires) its knowledge to (resp. from) \( B \) only when \( A \) fires a do\_k prefix on some session \( s[T] \), where \( A \)'s contract in \( T \) sends (resp. receives) \( k \) accordingly. In this case, each state of \( L^S \) is decorated with its corresponding CO2 system so to keep both the KLTS and the semantic LTS of \( S \) isomorphic.

In the example below, we show how two participants \( A \) and \( B \) share knowledge by interacting via two sessions — even though none of systems of contracts in these sessions share knowledge among \( \{A, B\} \).

**Example 7.6 (Sharing knowledge via two sessions).** Consider the CO2 system:

\[
S_1 = A[(x,y)\text{tell}_y \xrightarrow{\downarrow x} c_{A1} \mid \text{tell}_y \xrightarrow{\downarrow y} c_{A2} \mid \text{do}_w^k \cdot \text{do}_w^k \mid \text{B}[(w,z)\text{tell}_w \xrightarrow{\downarrow w} c_{B1} \mid \text{tell}_w \xrightarrow{\downarrow z} c_{B2} \mid \text{do}_w^k \cdot \text{do}_w^k] \mid K[\text{fuse} \cdot \text{fuse}] \]

with contracts \( c_{A1} = B!k, \ c_{A2} = B?k, \ c_{B1} = A?k, \ c_{B2} = A!k \). The system \( S_1 \) reduces as follows:

\[
S_1 \rightarrow^* (x,y,w,z)(A[\text{do}_w^k \cdot \text{do}_w^k] \mid \text{B}[\text{do}_w^k \cdot \text{do}_w^k] \mid K[\downarrow x \ A \text{ says } c_{A1} \mid \downarrow w \ B \text{ says } c_{B1}] \mid K[\downarrow y \ A \text{ says } c_{A2} \mid \downarrow z \ B \text{ says } c_{B2}] \mid K[\text{fuse} \cdot \text{fuse}])
\]

\[
K[\text{fuse}, K[\text{fuse}] \rightarrow S' = (s,t)(A[\text{do}_w^k \cdot \text{do}_w^k] \mid \text{B}[\text{do}_w^k \cdot \text{do}_w^k] \mid K[0] \mid s[A(B!k)] \mid B(A?k)] \mid t[A(B?k)] \mid B(A!k)]
\]

Participants \( A \) and \( B \) share knowledge in \( S' \) since they both exchange two \( k \) messages: in session \( s \), \( A \) sends its knowledge to \( B \), while in session \( t \), \( B \) sends its knowledge to \( A \).
Note however that neither A nor B is honest: they both perform consecutive do actions on different sessions, and the second prefix may become unreachable if the first one does not succeed. For instance, B will become persistently culpable if the process of A in S is replaced by:

\[ A[(y)tell_k ↓ y \ c_{a2} | do^y_k k] \quad \text{or} \quad A[(x,y)tell_k ↓ x \ c_{a1} | tell_k ↓ y \ c_{a2} | do^y_k k] \]

i.e., A does not advertise \( c_{a1} \), or it lacks the expected \( do^x_k k \) prefix. In our terminology, there are \( CO^2 \) systems (those in Section 7.6 above) such that B is not ready once it is composed with them.

The lack of honesty can be addressed by changing A and B’s processes so that they execute their do actions in parallel, e.g., A and B become

\[ A[(x,y)tell_k ↓ x \ c_{a1} | tell_k ↓ y \ c_{a2} | do^x_k k | do^y_k k] \]

\[ B[(w,z)tell_k ↓ w \ c_{b1} | tell_k ↓ z \ c_{b2} | do^w_k k | do^z_k k] \]

Example 7.6 shows that it is possible to achieve knowledge sharing through multiple sessions. If a broker were to guarantee knowledge sharing via, e.g., two sessions as above, it would have to wait for both sessions to be ready before starting them, thus ensuring that both session, as a whole, will implement a sharing knowledge protocol. In fact, this implies that these two sessions are not independent since they can only be started together, and, as discussed above, guaranteeing safety and liveness properties for inter-dependent sessions is not trivial.

Additionally, for a set of \( n \) honest participants to implement a sharing knowledge protocol via binary sessions, \( n \times (n - 1) \) sessions would be necessary. Multiparty sessions offer a more elegant solution here, as we show in Example 7.7 below.

Example 7.7. Consider the \( CO^2 \) system:

\[ S_3 = \ A[(x,y)tell_k ↓ x \ c_{a} | do^x_k k, do^y_k k] \quad \text{or} \quad \ B[(w,z)tell_k ↓ w \ c_{b} | do^w_k k, do^z_k k] \quad \text{K[fuse]} \]

with contracts \( c_{a} = B?k; A!k \) and \( c_{b} = A!k; B?k \). In this case, both A and B are honest and share knowledge in a single session.

7.3. Knowledge sharing choreographies

Our contract model allows for system of contracts that have infinite traces (due to the use of FIFO queues, notably). It is therefore crucial to have an effective and sound procedure to decide whether a given system \( T \) shares knowledge among \( \mathcal{P}(T) \). Definition 7.8 below gives such a procedure, by analysing a global type \( \mathcal{S} \).

We first introduce a map \( K : \mathcal{P} \rightarrow 2^\mathcal{P} \) that, given a participant name \( C \), returns the set of names representing the participants from whom \( C \) acquired knowledge. Initially, we have that \( K(C) = \{ C \} \) for each participant \( C \), i.e., a participant has only access to its own knowledge. Given a global type \( \mathcal{S} \), we want that

\[ \forall C \in \mathcal{P}(\mathcal{S}): K(C) = \mathcal{P}(\mathcal{S}) \]

holds at the end of the session embodied by \( \mathcal{S} \).

We define the boolean function \( \hat{K}(\mathcal{S},N,K) \) which returns true if all the participants in \( N \supseteq \mathcal{P}(\mathcal{S}) \) share the same knowledge (with respect to \( K \)) at the end of the session.

Definition 7.8. Let \( \hat{K}(\mathcal{S}) \triangleq \hat{K}(\mathcal{S},\mathcal{P}(\mathcal{S}),K_0) \), where \( K_0 \) is the initial knowledge map, i.e. \( \forall C \in \mathcal{P}(\mathcal{S}): K_0(C) = \{ C \} \) and

\[ \hat{K}(\mathcal{S},N,K) \triangleq \begin{cases} \hat{K}(\mathcal{S}',N,K') & \text{if } \mathcal{S} = A \rightarrow B: k; \mathcal{S}' \land K' = K[K(A) \cup K(B)/k] \\ \hat{K}(\mathcal{S}',N,K) \land \hat{K}(\mathcal{S}_2,N,K) & \text{if } \mathcal{S} = \mathcal{S}_1 \lor \mathcal{S}_2 \quad \text{and } \mathcal{P}(\mathcal{S}_1) \subseteq \text{dom}(K_i) \text{ with } i \in \{1,2\} \\ \text{false} & \text{otherwise} \end{cases} \]

We write \( \cup \) for the disjoint union of maps.
Let $T$ be equivalent in the contract model based on binary session types considered in [18]. In the present paper, we need to partition the knowledge map in two — since the sets of participants in the concurrent branches of $\mathcal{G}$ are disjoint. In the case of $\mathcal{G} = \emptyset$, we check that all participants share the same knowledge. For the sake of simplicity, the function returns $\text{false}$ if the global type is recursive.

We write $\text{fuse}_K$ for the variant of $\text{fuse}^\phi$ (as introduced in Section 4.5) where $\phi(\mathcal{G}) \iff \hat{K}(\mathcal{G})$. We illustrate Definition 7.8 in Example 7.10 below.

The proposition below (whose proof is in the appendix [17]) formalises the relationship between local and global knowledge sharing.

**Proposition 7.9.** Let $T$ be a system of contracts such that $\vdash T \triangleright \mathcal{G}$. If $\hat{K}(\mathcal{G})$, then $T$ shares knowledge among $\mathcal{P}(T)$.

We conclude this case study with an example that combine: (i) multiparty sessions, (ii) honest participants, and (iii) contractual agreements based on $\text{fuse}_K$.

**Example 7.10.** Consider the following contracts

\[
\begin{align*}
    c_A &= B_1^i k; B_2^i k; B_3^i k; B_1^! k; B_2^! k; B_3^! k \\
    c_B &\quad = A^! k : A^? k & i \in \{1, 2, 3\}
\end{align*}
\]

Let $T = A(c_A) \mid B_1(c_{B_1}) \mid B_2(c_{B_2}) \mid B_3(c_{B_3})$, we have $T \vdash \mathcal{G}$, with

\[
\mathcal{G} = B_1 \rightarrow A : k ; B_2 \rightarrow A : k ; B_3 \rightarrow A : k ; A \rightarrow B_1 : k ; B_2 \rightarrow B_1 : k ; A \rightarrow B_3 : k ; 0
\]

$\hat{K}(\mathcal{G})$ holds since at the end of the first line of $\mathcal{G}$, we have $K(A) = \{A, B_1, B_2, B_3\}$ and, in the second line, $A$ shares its acquired knowledge with all the other participants.

Consider the CO2 system below which uses the contracts above, where $K$ is a “knowledge sharing” broker using primitive $\text{fuse}_K$:

\[
S = A[P_k] \mid B_1[P_{B_1}] \mid B_2[P_{B_2}] \mid B_3[P_{B_3}] \mid K[\text{fuse}_K]
\]

with

\[
\begin{align*}
    P_k &= (w)(\text{tell}_k c_A \cdot \text{do}_{k}^w k \cdot \text{do}_{k}^w k \cdot \text{do}_{k}^w k \cdot \text{do}_{k}^w k \cdot \text{do}_{k}^w k) \\
    P_{B_i} &= (x)(\text{tell}_k c_{B_i} \cdot \text{do}_{k}^x k \cdot \text{do}_{k}^x k) & i \in \{1, 2, 3\}
\end{align*}
\]

Each of the participants above is honest and $K$ will only start knowledge sharing sessions.

8. Related Work

8.1. CO2 and Verification of Honesty

The origin of CO2 goes back to [10], where a first calculus of contracting processes was introduced. The calculus was then generalised in [22, 14] to suit different contract models (e.g., contracts as processes or logic formulae). In [11], it has been instantiated to a theory of bilateral contracts inspired by [23]. The main differences between our adaptation of CO2 and the original presentation in [22] lie in the specification of session establishment and the introduction of participant variables.

The notion of honesty has been introduced in [11], where a participant is considered honest when, in all possible contexts, it can always exculpate itself by a sequence of its own moves. Here, instead, we require that the participant is ready (Definition 5.3) in all possible contexts, similarly to [13]. The two notions are equivalent in the contract model based on binary session types considered in [18]. In the present paper, a similar correspondence is highlighted by Theorem 6.11 which states that an honest participant $A$ who is
culpable at some session \( s \) can always fire some \( \text{do}_s \) after a sequence of its own moves. Note however that \( A \) may still be culpable after such \( \text{do}_s \), e.g. when its contract requires to perform more outputs.

A negative result of [11] is that honesty is not decidable: hence, verification techniques can only aim at safely approximating it. Two different approaches have been proposed in [18, 24], where \( \text{CO}_2 \) is instantiated with binary session types. In [18], a type system for \( \text{CO}_2 \) processes is presented, associating behavioural types to all session variables used in a process; honesty is then verified by model checking these behavioural types. The technique introduced in [24] is rather different, and more closely based on the semantic definition of honesty: it first devises an abstraction of the contexts wherein \( A[B] \) may run, and then safely approximates the honesty of \( A \) by model checking such an abstraction. Basically, the abstraction only maintains the process of \( A \) and its contracts; the rest of the system is discarded, by over-approximating its moves; then, model checking is used to search the abstracted state space of \( A[B] \) for states where \( A \) is not ready. If the search fails, then \( A \) is honest. Such a search for non-ready states is decidable for finite state processes, i.e. those without delimitation/parallel under process definitions.

We expect that both techniques above can be adapted to cope with the contracts and the calculus studied in this paper. In particular, we now sketch one way to adapt the model checking technique of [24] to the present work.

- First, we develop a safe approximation of honesty, called 1-bounded honesty. We begin by observing that for all synthesizable \( T \), the culpability states of participants along the reductions of \( T \) can be faithfully captured by only considering “1-bounded” configurations (i.e., where each queue contains at most one message). Furthermore, the local and global types in the present work are a subset of those in [4]. Therefore, they enjoy a key property from [4]: for all synthesizable \( T \), each reachable 1-bounded configurations is reachable via a “1-bounded” computation (i.e., an execution such that each step has 1-bounded queues). We can then define “1-bounded” honesty as in Definition 5.8, but using the 1-bounded contract semantics, instead of the unbounded one; the observations above can be used to prove that 1-bounded honesty implies honesty. Note that the 1-bounded contract semantics is finite-state.

- Second, we devise an abstraction of 1-bounded systems of contracts wrt. a participant \( A \), whose honesty we want to verify. This abstraction must preserve all the moves of \( A \), while over-approximating the moves of other possible participants. This is done similarly to [24]. Such an abstraction also has a finite-state semantics.

- Finally, we design an abstraction for \( \text{CO}_2 \) systems which preserves the moves of a participant \( A \) while over-approximating those of the context (similarly to the abstraction for contracts). Again, this abstraction can be obtained by slightly adapting the one in [24], using the abstract contracts from the previous item. Given any \( A[B] \) with no delimitation/parallel under process definitions, such an abstraction has a finite-state semantics.

Summing up, we have a finite-state abstraction of \( A[B] \) that, as in [24], can be searched for states where \( A \) is not ready. If the search fails, we say that \( A[B] \) is abstractly honest. From the observations above, abstract honesty implies 1-bounded honesty, which in turn implies honesty.

8.2. Choreographies and Compliance among Contracts

Several papers have addressed the problem of defining notions of compliance (also called agreement, or compatibility) among contracts, both in the biparty and in the multiparty case. Very roughly, compliance relates contracts that, when composed, give rise to “correct” interactions.

Our notion of compliance (Definition 3.6) exploits the approach introduced in [6, 16], which gives a typing system allowing to synthesise a choreography (i.e., a global type) from a set of local types, and gives a characterisation of the global types that can be synthesised. To the best of our knowledge, our approach is the first formal model of distributed systems where contractual agreement is based on the existence of a choreography. Deniélou and Yoshida [4] have given a characterisation of finite-state communicating machines from which it is possible to synthesise a global type, as well as an algorithm to build global types.
from such machines. Their results could also be used as a foundation for an agreement relation similar the one presented here. However, we remark that the global type obtained from the algorithm in [4] is not unique in general, notably due to the lack of special treatment of independent interactions (i.e., $\mathcal{G} \mid \mathcal{G}'$, in our notation). Practically, this means that specific fuse relations (such as the ones discussed in Section 4.3) based on this algorithm may become non-deterministic.

In [25] contracts are considered compliant when their composition may always reach success (so guaranteeing progress until success). In our work, progress is provided by the synthesis of a global type. In [26], a contract language has been proposed for processes, where, as in the $\pi$-calculus, communications are synchronous and channel names can be passed around. The notion of compliance in [27] is based on a testing approach: contracts are compliant if, essentially, their composition has progress. In the same line of work, [28] proposes to “project” processes into session types (akin to the contracts of [27] but without name passing capabilities). The author gives an adapted notion of compliance, called completeness, which permits to characterise when a set of session types is able to reduce further. The semantic correspondence between global specifications (choreographies) and local specification (local types) has been investigated in [29, 12]. In the context of binary session types, several notions of duality (i.e., compliance between two complementary session types) are studied and compared in [30], including the higher-order case. A syntax-independent notion of compliance, defined over generic LTSs with input/output actions, and encompassing first-order binary session types with either synchronous or asynchronous semantics, is introduced in [31].

Choreographies have been studied with different types of syntax and several well-formedness conditions (ensuring properties of their projections). The syntax of our choreographies is close to the ones used in [12, 3, 32] and satisfy similar properties such as knowledge of choice (or local choice); other formalisms include automata-based syntax (e.g., [33]), Petri nets [34] and graphical representation such as BPMN [35]. Each formalism generally comes with a set of conditions ensuring good properties on the system specified by the choreography — such as realisability [35] or connectedness [29] which ensure a correspondence between the order of messages exchanged by the projections and the order specified by the choreography. In the present work, we only require that a choreography specifies a safe system of contracts (which is guaranteed by the synthesis).

8.3. Conformance between Processes and Contracts

The seminal top-down approach of multiparty session types was first presented in [3]. The methodology it advocates is as follows: a team of programmers designs a global view of the interactions to be implemented (i.e., a choreography or global type), then the choreography is projected onto each role, or participant, so to obtain local types; finally each program implementing one or more roles in the choreography is type-checked against its corresponding projection(s). The approach permits, if the choreography is “well-formed”, to guarantee safety properties such as deadlock freedom, and to ensure that an implementation abides by the specified protocol (session fidelity).

As in most works on session types (e.g., [3, 32, 12]), our contract model does not feature concurrency at the local level, i.e., it does not allow for a multi-threaded single contract. However, our results could be easily extended to support local concurrency, by using a choreography synthesis framework which does cater for it: for instance, the one in [37], which offers the same safety guarantees as the one in this work.

Honesty ensures that a CO2 process will interact in a session according to some specific contract. A similar problem is tackled in the session types literature. However, session type based systems do not cater for a notion of “culpability”, and are generally unable to guarantee progress for processes interacting on multiple interleaved sessions without strong assumptions on their execution context. The problem of guaranteeing progress in a composition of processes interacting through several interleaved sessions is tackled in [19, 32] via a complex typing system. Their results include that a well-typed compositions of processes can interact on multiple interleaved sessions without deadlocking — and thus, in our setting, without remaining persistently culpable. In the present paper, a similar progress result is provided for systems of honest participants (Corollary 6.3): a main difference wrt. [19, 32] is that the honesty of each participant is checked “in isolation”, without assumptions on the runtime context.
Another related approach in the context of multiparty session types is that of run-time monitoring \[38, 39\]. Basically, each monitor is constructed from a local type, and mediates the inputs/outputs of a process in a multiparty session; run-time safety and session fidelity properties are enforced by forbidding interactions not specified in the originating local type. This approach is similar to that of CO\(_2\) in the sense that local types are used to “guard” the processes implementing the protocols, as required by the [CO\(_2\)-Do] rule premises (Figure 4). The framework in \[38\], however, requires a global type to be available beforehand in order to project process monitors — while in our case, global types are synthesised dynamically from the local types advertised by each participant.

9. Concluding Remarks

In this work, we investigated the combination of the contract-oriented calculus CO\(_2\) with a contract model that fulfils two basic design requirements: (i) it supports multiparty agreements, and (ii) it provides an explicit description of the choreography that embodies each agreement. These requirements prompted us towards the well-established results from the session types setting — in particular, regarding the interplay between a well-formed global type and its corresponding local behaviours. We introduced the concepts of global progress and session fidelity in CO\(_2\), also inspired from the analogous concepts in theories based on session types. We built our framework upon a simple version of session types, and yet it turns out to be quite flexible, allowing for sessions where the number of participants is not known beforehand.

This article is the full version of the extended abstract published in \[40\]. Here we streamlined several aspects of the theory, introduced additional results, and extended definitions, explanations and proofs. In addition, we present a large case study whereby our approach is put into practice.

Future work. We plan to extend our work so to offer even more flexibility — for example, by analysing further parameterised fuse\(^5\) variants implementing different session establishment criteria. We also plan to study the possibility for a participant to be involved in a session under multiple contracts, e.g., a bank advertising two services, and a customer publishing a contract which uses both of them in a well-formed choreography.

Another research direction is the concept of “group honesty”. In fact, the current definition of honesty is quite strict: it basically verifies each participant in isolation, thus providing a sufficient (but not necessary) condition for progress. Consider, for example, a CO\(_2\) system like:

$$S = (x, y)\left( A\left[tell\downarrow_x (B\left[tell\downarrow_y (A\left\{ \text{int} \boxplus B\left\{ \text{bool} \} \left. \text{do} x B \left\{ \text{int} \right. \right) \mid B\left[tell\downarrow_y (A\left\{ \text{int} + A\left\{ \text{bool} \} \left. \text{do} y A \left\{ \text{int} \right. \right) \right)\right. \right)\right. \right)\right. \right)\right. \right)$$

\(B\) is not honest, since it is not ready for the \text{bool} branch of its contract. However, the system \(S\) has progress: when \(B\) establishes a session with \(A\), the latter will never take the \text{bool} branch; hence, \(B\) will not remain culpable. This kind of “group honesty” may be used to validate (sub-)systems of participants developed by the same organization: it would ensure that they never “cheat each other”, and are collectively honest when deployed in any context. Furthermore, the group honesty of all participants in a system \(S\) may turn out to be a necessary condition for the global progress of \(S\).

Acknowledgements. We would like to thank Emilio Tuosto for his (in)valuable advice, and the anonymous reviewers of ICE 2013 for their comments and suggestions.

References

[38] G. Castagna, N. Gesbert, L. Padovani, A theory of contracts for web services, ACM Trans. on Prog. Lang. and Sys. 31 (5).