

Contracts in distributed systems

Massimo Bartoletti

Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy

Emilio Tuosto

Department of Computer Science, University of Leicester, UK

Roberto Zunino

DISI-Università degli Studi di Trento and COSBI, Italy

We present a parametric calculus for contract-based computing in distributed systems. By abstracting from the actual contract language, our calculus generalises both the *contracts-as-processes* and *contracts-as-formulae* paradigms. The calculus features primitives for advertising contracts, for reaching agreements, and for querying the fulfilment of contracts. Coordination among principals happens via multi-party sessions, which are created once agreements are reached. We present two instances of our calculus, by modelling contracts as (i) processes in a variant of CCS, and (ii) as formulae in a logic. With the help of a few examples, we discuss the primitives of our calculus, as well as some possible variants.

1 Introduction

What are contracts for distributed services? How should they be used? These questions are intriguing not only researchers but also practitioners and vendors. In fact, contracts are paramount for correctly designing, implementing, and composing distributed software services. In such settings, contracts are used at different levels of abstraction, and with different purposes. Contracts are used to model the possible interaction patterns of services, with the typical goal of composing those services only which guarantee deadlock-free interactions. At a different level of abstraction, contracts are used to model Service Level Agreements (SLAs), specifying what has to be expected from a service, and what from the client. Also in this case, a typical goal is that of matching clients and services, so that they agree on the respective rights and obligations.

Contracts have been investigated from a variety of perspectives and using a variety of different formalisms and analysis techniques, ranging from *c*-semirings [7, 8, 15], to behavioural types [6, 10, 11], to formulae in suitable logics [1, 3, 21], to categories [9], *etc.* This heterogeneous ecosystem of formalisms makes it difficult to understand the essence of those methods, and how they are related.

As a first step towards remedying this situation, we propose a generic calculus for Contract-Oriented Computing (in short, CO₂). By abstracting away from the actual contract language, our calculus can encompass a variety of different contract paradigms. We provide a common set of primitives for computing with contracts: they allow for advertising and querying contracts, for reaching agreements, and for fulfilling them with the needed actions. All these primitives are independent from the chosen language of contracts, and they only pivot on some general requirements fixed in the *contract model* proposed here.

A remarkable feature of our approach is that contracts are not supposed to be always respected after they have been stipulated. Indeed, we can model the quite realistic situation where promises may be possibly reneged. Therefore, in CO₂ contracts are not discharged after they have been used to couple services and put them in a session, as usually done e.g. in the approaches dealing with behavioural types.

In our approach, contracts are also used to drive computations after sessions have been established, e.g. to detect violations and to provide the agreed compensations.

Synopsis. The overall contribution of the paper is a calculus for computing with contracts in distributed systems. The calculus is designed around two main principles.

The first is the separation of concerns between the way contracts are modelled and the way they are used in distributed computations. Indeed, we abstract from the actual contract language by only imposing a few general requirements. In this way, we envisage our calculus as a generic framework which can be tuned by instantiating the contract model to concrete formalisations of contracts. In § 2 we present the abstract contract model, followed by two concretisations: in § 2.1 we adopt the contracts-as-processes paradigm whereby CCS-like processes represent contracts that drive the behaviour of distributed participants; in § 2.2 we embrace instead the contracts-as-formulae paradigm, by instantiating our calculus with contracts expressed in a suitable logic. We relate the two concrete models in § 2.3, first with the help of a few examples, and then by showing that contracts-as-formulae, expressed in a significant fragment of our logic, can be suitably encoded into contracts-as-processes (Theorem 2.7).

The second design principle of our calculus is that its primitives must be reasonably implementable in a distributed setting. To this purpose, we blend in § 3 a few primitives inspired by Concurrent Constraint Programming (CCP [22]) to other primitives inspired by session types [17]. The key notions around which our primitives are conceived are *principals* and *sessions*. The former represent distributed units of computation that can advertise contracts, execute the corresponding operations, and establish/check agreements. Each agreement corresponds to a fresh session, containing rights and obligations of each stipulating party. Principals use sessions to coordinate with each other and fulfil their obligations. Also, sessions enable us to formulate a general notion of “misbehaviour” which paves the way for automatic verification. We finally suggest possible variants of our primitives and of the contract model (§ 3.6).

Related Work. Multi-party session types [18] are integrated in [5] with decidable fragments of first-order logic (e.g., Presburger arithmetic) to transfer the design-by-contract of object-oriented programming to the design of distributed interactions. We follow a methodologically opposite direction. In fact, in [5] one starts from a *global assertion* (i.e., global choreography and contracts) to arrive to a set of *local assertions*; distributed processes abiding with local assertions are guaranteed to have correct interactions (and monitors can be synthesized from local assertions to control execution in untrusted settings). In our framework instead, a principal declares its contract independently of the others and then advertises it; a CO_2 primitive tries then to harmonise contracts by searching for a suitable agreements. In other words, one could think of our approach as based on orchestration rather than choreography. The same considerations above apply to [19] where protocol modelling (state machines with memory) represents global choreographies. There, contracts are represented as parallel state machines (according to a CSP-like semantics). Basically, the contract model of [19] coincides with its choreography model.

In cc-pi [8], CCP is mixed with communication through name fusion. In this model, involved parties establish SLA by merging the constraints representing their requirements. Constraints are values in a c-semiring advertised in a global store. It is not permitted to merge constraints making the global store inconsistent, since an agreement cannot be reached in that case. Conversely, CO_2 envisages contracts as binding promises rather than requirements. Actually, even if a principal *A* tells an absurdum, this will result in a contract like: “*A* is stating a contradiction” added to the environment. When this happens, our approach is not “contracts are inconsistent, do not open a session”, but rather “*A* is promising the impossible, she will not be able to keep her promise, and she will be blamed for that”. The cc-pi calculus

is further developed in [7] to include long running transactions and compensations. There, besides the global constraint store, a local store for each transaction is featured. Local inconsistencies are then used to trigger compensations. In CO_2 , compensations do not represent exceptional behaviour to be automatically triggered by inconsistencies; rather, compensations fall within “normal” behaviour and have to be spelt out inside contracts. Indeed, after a session has been established, each honest principal A either maintains her promises, or she is culpable of a violation; she cannot simply try to execute arbitrary compensations in place of the due actions. Of course, other principals may deem this promise too weak and avoid establishing a session with A .

In [13] a calculus is proposed to model SLAs which combines π -calculus communication, concurrent constraints, and sessions. There, the constraint store is global and sessions are established between two processes whenever the stated requirements are consistent. Interaction in sessions happens through communication and label branching/selection. A type system is provided to guarantee safe communication, although not ensuring progress. Essentially, the main role of constraints in this calculus is that of driving session establishment. Instead, in CO_2 the contracts of an agreement leading to a session are still relevant e.g. to detect violations.

A “boolean” notion of compliance between two contracts is introduced in [12]: either the contract of the client and one of the service are compliant, or they are not. In Ex. 3.4 we discuss a “multi-level” notion of compliance encompassing more than two contracts. Also, in [12] not compliant contracts, may become compliant by adjusting the order of asynchronous actions. When this is possible, an orchestrator can be synthesised from the client and service contracts. In some sense, the orchestrator acts as an “adapter” between the client and the service. In our approach, the orchestrator behaves as a “planner” which finds a suitable set of contracts and puts in a session all the principals involved in these contracts.

CO_2 takes inspiration from [3, 4]. There, the contract language is the logic PCL, and contracts are recorded into a global constraint store. CO_2 instead features local environments for principals and sessions to enable possible distributed implementations.

Our approach differs from those discussed above, as well as from all the other approaches we are aware of (e.g. [6, 10, 11]), w.r.t. two general principles. First, we depart from the common principle that contracts are always respected after their stipulation. We represent instead the more realistic situation where promises are not always maintained. As a consequence, in CO_2 we do not discard contracts after they have been used to couple services and put them in a session, as done e.g. in all the approaches dealing with behavioural types. In our approach, contracts are also used to drive computations after sessions have been established (cf. § 3), e.g. to detect violations and to provide the agreed compensations.

The second general difference is that CO_2 smoothly allows for handling contracts-as-processes (cf. § 2.1). To the best of our knowledge, it seems hard to accomodate these contracts within frameworks based on constraint systems ([4, 13]), logics ([3, 5]), or c-semirings (e.g. [8, 7, 15]).

2 An abstract contract model

We now sketch the basic ingredients of a generic contract model, before providing a formal definition.

We start by introducing some preliminary notions and definitions; some of them will only be used later on in § 3. *Principals* are those agents which may advertise contracts, establish agreements, and realise them. *Sessions* are created upon reaching an agreement, and provide the context in which principals can interact to fulfil their contracts. Let \mathcal{N} and \mathcal{V} be countably infinite, disjoint sets of names and variables, respectively. Assume \mathcal{N} partitioned into two infinite sets \mathcal{N}_P and \mathcal{N}_S , for names of principals and of sessions, respectively. Similarly, \mathcal{V} is partitioned into infinite sets \mathcal{V}_P and \mathcal{V}_S for variable identifiers

$n, m, \dots \in \mathcal{N}$	names, union of:	$A, B, \dots \in \mathcal{N}_P \cup \mathcal{V}_P$	principal names/variables
$A, B, \dots \in \mathcal{N}_P$	principal names	$a, b, \dots \in \mathcal{A}$	atoms
$s, t, \dots \in \mathcal{N}_S$	session names	$\langle A_1 \text{ says } a_1, \dots, A_j \text{ says } a_j \rangle$	action
\mathcal{V}	variables, union of:	$c, c', \dots \in \mathcal{C}$	contracts
$a, b, \dots \in \mathcal{V}_P$	principal variables	C	multisets of contracts
$x, y, \dots \in \mathcal{V}_S$	session variables	$\phi, \psi, \dots \in \Phi$	observables
$u, v, \dots \in \mathcal{N} \cup \mathcal{V}$	names or variables	μ, μ', \dots	labels

Table 1: Notation

of principals and sessions. A substitution is a partial map σ from \mathcal{V} to \mathcal{N} ; we write $u \in \text{dom } \sigma$ when σ is defined at u , and require that σ maps $a \in \text{dom } \sigma \cap \mathcal{V}_P$ to \mathcal{N}_P and $s \in \text{dom } \sigma \cap \mathcal{V}_S$ to \mathcal{N}_S .

Our main notational conventions are displayed in Table 1.

The first ingredient of our contract model is a set C of *contracts*. We are quite liberal about it: we only require that $A \text{ says } c \in C$ for all principals A and for all $c \in C$. The contract $A \text{ says } c$ can be thought of as “ c is advertised by A ”. A labelled transition relation $\xrightarrow{\mu}$ on contracts models their evolution under the actions performed by principals.

Two further ingredients are a set Φ of *observables* (properties of contracts) and an entailment relation \vdash between contracts and observables. Note that we keep distinct contracts from observables in our framework. This has the same motivations as the traditional distinction between behaviours (systems) and their properties (formulae predicating on behaviours), which brought in plenty of advantages in the design/implementation of systems.

The last ingredient of our contract model is a relation \odot between contracts and principals. We write $C \odot A$ to mean that, with respect to contracts C , all the obligations of the principal A have been fulfilled.

Def. 2.1 formalises the above concepts.

Definition 2.1. A contract model is a tuple $\langle C, \mathcal{A}, \rightarrow, \Phi, \vdash, \odot \rangle$ where

- C is a set of contracts, forming a subalgebra of a term-algebra $T_{\mathcal{V} \cup \mathcal{N}}(\Sigma)$ for some signature Σ which includes the operations $u \text{ says } _$ for each $u \in \mathcal{V}_P \cup \mathcal{N}_P$
- \mathcal{A} is a set of atoms (ranged over by a, b, \dots)
- $C \xrightarrow{\mu} C'$ is a labelled transition relation over finite multisets on C . The set of labels comprises actions, i.e. tuples of the form $\langle A_1 \text{ says } a_1, \dots, A_j \text{ says } a_j \rangle$
- Φ is a set of observables, forming a subalgebra of a term-algebra $T_{\mathcal{V} \cup \mathcal{N}}(\Sigma')$ for some signature Σ'
- \vdash is a contract entailment relation between finite multisets of C and Φ
- \odot is a contract fulfilment relation between finite multisets of C and principals.

Example 2.1. We illustrate the contract model with the help of an informal example. A seller A and a buyer B stipulate a contract c_0 , which binds A to ship an item after B has paid. Let pay be the atom which models the action of paying. The transition $c_0 \xrightarrow{B \text{ says pay}} c_1$ models the evolution of c_0 into a contract c_1 where A is obliged to ship, while B has no more duties. Now, let ϕ be the observable “ A must ship”. Then, we would have $c_0 \not\vdash \phi$, because A does not have to ship anything yet, while $c_1 \vdash \phi$, because B has paid and so A must ship. It would not be the case that $c_1 \odot A$, because B has paid, while A has not yet fulfilled her obligation to ship.

$$\mu ::= \mathbf{a} \mid \langle A \text{ says } \mathbf{a}^0 \rangle \mid \langle A \text{ says } \mathbf{a}^-, A \text{ says } \mathbf{a}^+ \rangle$$

$$\frac{\sum_i \mathbf{a}_i.c_i \xrightarrow{\mathbf{a}_i} c_i}{\text{(Sum)}} \quad \frac{c_1 \xrightarrow{\mu} c'_1}{c_1 \mid c_2 \xrightarrow{\mu} c'_1 \mid c_2} \text{(Par)} \quad \frac{X \stackrel{\text{def}}{=} c \quad c \xrightarrow{\mu} c'}{X \xrightarrow{\mu} c'} \text{(Def)}$$

$$\frac{c \xrightarrow{\mathbf{a}^0} c' \quad \mu = \langle A \text{ says } \mathbf{a}^0 \rangle}{A \text{ says } c \xrightarrow{\mu} A \text{ says } c'} \text{(Auto)} \quad \frac{c_1 \xrightarrow{\mathbf{a}^-} c'_1 \quad c_2 \xrightarrow{\mathbf{a}^+} c'_2 \quad \mu = \langle A_1 \text{ says } \mathbf{a}^-, A_2 \text{ says } \mathbf{a}^+ \rangle}{A_1 \text{ says } c_1 \mid A_2 \text{ says } c_2 \xrightarrow{\mu} A_1 \text{ says } c'_1 \mid A_2 \text{ says } c'_2} \text{(Com)}$$

Table 2: Labelled transition relation of contract-as-processes

We remark that the use of term-algebras in Def. 2.1 allows us to smoothly apply variable substitutions to contracts and observables. Accordingly, we assume defined the sets $\text{fv}(c)$ and $\text{fv}(\phi)$ of (free) variables of contracts and observables. Note that actions are not required to be in \mathcal{C} . Depending on the actual instantiation of the contract model, it can be useful to include them in \mathcal{C} , so that contracts can record the history of the past actions.

2.1 Contracts as processes

The first instance of our contract model appeals to the contracts-as-processes paradigm. A contract is represented as a CCS-like process [20], the execution of which dictates obligations to principals.

Definition 2.2. *We define a contracts-as-processes language as follows*

- \mathcal{C} is the set of process terms defined by the following grammar:

$$c ::= \sum_i \mathbf{a}_i.c_i \mid A \text{ says } c \mid c \mid c \mid X$$

and Σ is the signature corresponding to the syntax above; in this section, multisets of contracts are identified with their parallel composition, and accordingly we use the metavariable c to denote them. We assume variables X to be defined through (prefix-guarded recursive) equations.

- \mathcal{A} is the union of three disjoint sets: the “inputs” (ranged over by \mathbf{a}^-), the “outputs” (ranged over by \mathbf{a}^+), and the “autonomous activities” (ranged over by \mathbf{a}^0).
- $\xrightarrow{\mu}$ is the least relation closed under the rules in Table 2 and structural equivalence \equiv (defined with the usual rules and $A \text{ says } \mathbf{0} \equiv \mathbf{0}$, where $\mathbf{0}$ denotes the empty sum and trailing occurrences of $\mathbf{0}$ may be omitted).
- Φ is the set of LTL [14] formulae (on a signature Σ'), where the constants are the atoms in \mathcal{A} .
- $c \vdash \phi$ (for closed c and ϕ) holds when $c \models_{\text{LTL}} \phi$ according to the standard LTL semantics where, given a generic trace η of c , the semantics of atoms is:

$$\begin{aligned} \eta \models \mathbf{a}^0 &\iff \exists A, \eta'. \eta = \langle A \text{ says } \mathbf{a}^0 \rangle \eta' \\ \eta \models \mathbf{a}^+ &\iff \eta \models \mathbf{a}^- \iff \exists A_1, A_2, \eta'. \eta = \langle A_1 \text{ says } \mathbf{a}^-, A_2 \text{ says } \mathbf{a}^+ \rangle \eta' \end{aligned}$$

- $c \odot A$ holds iff for all c', c'' such that $c \equiv (A \text{ says } c') \mid c''$ we have that $c' \equiv \mathbf{0}$.

We briefly comment on the rules in Table 2. Intuitively, the relation $\xrightarrow{\mu}$ either carries labels of the form $\langle A \text{ says } a^0 \rangle$, which instruct A to fulfil the obligation a^0 , or labels $\langle A_1 \text{ says } a^-, A_2 \text{ says } a^+ \rangle$, which require participants A_1 and A_2 to fulfil the obligations a^- and a^+ , respectively. Rules (SUM), (PAR), and (DEF) are standard. By rule (AUTO), a contract willing to perform an autonomous action a^0 can do so and exhibit the label $\langle A \text{ says } a^0 \rangle$. Rule (COM) is reminiscent of the synchronisation mechanism of CCS; when two complementary actions a^- and a^+ can be fired in parallel, then the parallel composition of contracts emits the tuple $\langle A_1 \text{ says } a^-, A_2 \text{ says } a^+ \rangle$. Note that the rules in Table 2 give semantics to *closed* contracts, i.e. contracts with no occurrences of free variables.

Example 2.2. Recall the buyer-seller scenario from Ex. 2.1. The seller A promises to ship an item if buyer B promises to pay. The buyer B promises to pay. The contracts of A and B are as follows:

$$c_A = A \text{ says } \text{pay}^- . \text{ship}^0 \qquad c_B = B \text{ says } \text{pay}^+$$

A possible computation is then: $c_A \mid c_B \xrightarrow{\langle A \text{ says } \text{pay}^-, B \text{ says } \text{pay}^+ \rangle} A \text{ says } \text{ship}^0 \mid \mathbf{0} \xrightarrow{\langle A \text{ says } \text{ship}^0 \rangle} \mathbf{0}$.

It is evident that the contract of B in Ex. 2.2 is rather naive; the buyer pays without requiring any guarantee to the seller (cf. Ex. 3.6). A possible solution is to use a (trusted) escrow service.

Example 2.3. In the same scenario of Ex. 2.2, consider an escrow service E which mediates between A and B . Basically, A and B trust the escrow service E , and they promise to ship to E and to pay E , respectively. The escrow service promises to ship to B and to pay A only after both the obligations of A and B have been fulfilled. The contracts of A , B , and E are defined as follows:

$$\begin{aligned} c_A &\stackrel{\text{def}}{=} A \text{ says } \text{ship}E^+ . \text{pay}^- & c_E &\stackrel{\text{def}}{=} E \text{ says } \text{ship}E^- . \text{pay}E^- . (\text{pay}^+ \mid \text{ship}^+) + \\ c_B &\stackrel{\text{def}}{=} B \text{ says } \text{pay}E^+ . \text{ship}^- & & \text{pay}E^- . \text{ship}E^- . (\text{pay}^+ \mid \text{ship}^+) \end{aligned}$$

2.2 Contracts as formulae

For the second specialization of our generic model, we choose the contract logic PCL [3]. A comprehensive presentation of PCL is beyond the scope of this paper, so we give here just a brief overview, and we refer the reader to [3, 2] for more details.

PCL extends intuitionistic propositional logic IPC [23] with the connective \twoheadrightarrow , called *contractual implication*. Differently from IPC, a contract $b \twoheadrightarrow a$ implies a not only when b is true, like IPC implication, but also in the case that a “compatible” contract, e.g. $a \twoheadrightarrow b$, holds. So, PCL allows for a sort of “circular” assume-guarantee reasoning, summarized by the theorem $\vdash (b \twoheadrightarrow a) \wedge (a \twoheadrightarrow b) \rightarrow a \wedge b$. Also, PCL is equipped with an indexed lax modality $_ \text{ says } _$, similarly to the one in [16].

The proof system of PCL extends that of IPC with the following axioms, while remaining decidable:

$$\begin{array}{ll} \top \twoheadrightarrow \top & \phi \rightarrow (A \text{ says } \phi) \\ (\phi \twoheadrightarrow \phi) \rightarrow \phi & (A \text{ says } A \text{ says } \phi) \rightarrow A \text{ says } \phi \\ (\phi' \rightarrow \phi) \rightarrow (\phi \twoheadrightarrow \psi) \rightarrow (\psi \rightarrow \psi') \rightarrow (\phi' \twoheadrightarrow \psi') & (\phi \rightarrow \psi) \rightarrow (A \text{ says } \phi) \rightarrow (A \text{ says } \psi) \end{array}$$

Following Def. 2.1, we now define a contract language which builds upon PCL.

Definition 2.3. We define a contracts-as-formulae language as follows:

- C is the set of PCL formulae. Accordingly, Σ comprises all the atoms \mathcal{A} (see below), all the connectives of PCL, and the $_says_$ modality.
- \mathcal{A} is partitioned in promises, written as a , and facts, written as $!a$.
- The labelled relation $\xrightarrow{\mu}$ is defined by the rule: $C \xrightarrow{\langle A \text{ says } a \rangle} C, A \text{ says } a, A \text{ says } !a$
- $\Phi = C$, and $\Sigma' = \Sigma$.
- \vdash is the provability relation of PCL.
- $C \odot A$ holds iff $C \vdash A \text{ says } a$ implies $C \vdash A \text{ says } !a$, for all promises a , i.e. each obligation for A entailed by C has been fulfilled.

Note that the definition of $\xrightarrow{\mu}$ allows principals to perform any actions: the result is that C is augmented with the corresponding fact $!a$. We include the promise a as well, following the intuition that a fact may safely imply the corresponding promise.

Example 2.4. The contracts of seller A and buyer B from Ex. 2.1 can be modelled as follows:

$$c_A = A \text{ says } ((B \text{ says } \text{pay}) \rightarrow \text{ship}) \quad c_B = B \text{ says } \text{pay}$$

By the proof system of PCL, we have that: $c_A \wedge c_B \vdash (A \text{ says } \text{ship}) \wedge (B \text{ says } \text{pay})$.

2.3 On contracts-as-processes vs. contracts-as-formulae

We now compare contracts-as-processes with contracts-as formulae. We start with an empirical argument, by comparing in Table 3 a set of archetypal agreements which use contracts from both paradigms. Our main technical result is Theorem 2.7, where we show that contracts-as-formulae, expressed in a significant fragment of PCL, can be encoded into contracts-as-processes. Finally, we further discuss the differences between the two contract models in some specific examples.

Sketching a correspondence. We now discuss Table 3. Contracts yielding similar consequences lay on the same row. Each row tells when *interaction* is possible, i.e. when processes will eventually reach $\mathbf{0}$, and when the formulae entail the observable $A \text{ says } a \wedge B \text{ says } b$.

In row 1, B performs b unconditionally. Instead, A specifies in her contracts a causal dependency between b and a — using a prefix in the world of processes, or an implication in the world of formulae. Note that this exchange offers no protection for B , i.e., c_A could be replaced with anything else and B would still be required to provide b .

In row 2, B protects himself by using a causal dependency, using the dual contract of c_A . Now however interaction/entailment is lost: every principal requires the other one to “make the first step”, and circular dependencies forbid any agreement.

In row 3, B makes the first step by inverting the order of the causal dependency in c_B . The outcome is similar to row 1, except that now the contract of B mentions that B expects a to be performed. The meaning of the process is roughly “offer b first, then require a ” which has no analogous contract-as-formula.

In row 4, B is offering b asynchronously, so removing the causal dependency. In the world of processes, this is done using parallel composition instead of a prefix; in the world of (PCL) formulae this

		Contracts-as-processes	Contracts-as-formulae
1.	c_A	$A \text{ says } b^-.a^0$	$A \text{ says } (B \text{ says } b) \rightarrow a$
	c_B	$B \text{ says } b^+$ no protection, interaction	$B \text{ says } b$ no protection, entailment
2.	c_A	$A \text{ says } b^-.a^+$	$A \text{ says } (B \text{ says } b) \rightarrow a$
	c_B	$B \text{ says } a^-.b^+$ no interaction	$B \text{ says } (A \text{ says } a) \rightarrow b$ no entailment
3.	c_A	$A \text{ says } b^-.a^+$	(no equivalent)
	c_B	$B \text{ says } b^+.a^-$ asymmetric, interaction	
4.	c_A	$A \text{ says } b^-.a^+$	$A \text{ says } (B \text{ says } b) \rightarrow a$
	c_B	$B \text{ says } b^+ a^-$ asymmetric, interaction	$B \text{ says } (A \text{ says } a) \rightarrow b$ asymmetric, entailment
5.	c_A	$A \text{ says } b^- a^+$	$A \text{ says } (B \text{ says } b) \rightarrow a$
	c_B	$B \text{ says } b^+ a^-$ symmetric, interaction	$B \text{ says } (A \text{ says } a) \rightarrow b$ symmetric, entailment

Table 3: Contracts-as-processes vs. contracts-as-formulae

is done using \rightarrow instead of \rightarrow . Interaction/entailment is still possible. The contracts c_A and c_B are not symmetric, in which case c_A specifies a causal dependency, while c_B does not.

In row 5, causal dependency is removed from c_A as well. Both A and B are using parallel composition/contractual implication. This results in symmetric contracts which yield interaction/entailment.

A formal correspondence between contract models. We now provide a precise correspondence between two fragments of contracts-as-formulae and contracts-as-processes, building upon the intuition underlying the cases shown in Table 3.

Concretely, we consider a fragment (PCL^-) of PCL contracts comprising atoms, conjunctions, says, and non-nested (contractual) implications. Then, we provide an encoding of PCL^- into contract-as-processes (Def. 2.5) and formally relate them.

Definition 2.4. We define PCL^- as the fragment of PCL where formulae c have the following syntax:

$$\begin{aligned}
c &::= \bigwedge_{i \in I} A_i \text{ says } \alpha_i \\
\alpha &::= \bigwedge_{j \in J} \mathbf{q}_j \mid \left(\bigwedge_{i \in I} B_i \text{ says } \mathbf{p}_i \right) \rightarrow \bigwedge_{j \in J} \mathbf{q}_j \mid \left(\bigwedge_{i \in I} B_i \text{ says } \mathbf{p}_i \right) \rightarrow \bigwedge_{j \in J} \mathbf{q}_j
\end{aligned}$$

Definition 2.5. For all formulae c of PCL^- , we define the contract-as-process $[c]$ as follows, where, for all atoms \mathbf{q} , $\text{OUT}(\mathbf{q})$ is a recursive process defined by the equation $\text{OUT}(\mathbf{q}) = \tau^0.\mathbf{0} + \mathbf{q}^+.\text{OUT}(\mathbf{q})$. Also, we assume an injective function $_/_$ that maps all atoms \mathbf{q} and all principals A into the atom $\mathbf{q}/_A$.

$$\begin{aligned}
[\bigwedge_{i \in I} A_i \text{ says } \alpha_i] &= \parallel_{i \in I} A_i \text{ says } [\alpha_i]_{A_i} \\
[\bigwedge_{j \in J} \mathbf{q}_j]_A &= \parallel_{j \in J} \text{OUT}(\mathbf{q}_j/A) \\
[(\bigwedge_{i \in \{1, \dots, n\}} B_i \text{ says } \mathbf{p}_i) \rightarrow \bigwedge_{j \in J} \mathbf{q}_j]_A &= \mathbf{p}_{1/B_1}^- \cdots \mathbf{p}_{n/B_n}^- \cdot \left(\parallel_{j \in J} \text{OUT}(\mathbf{q}_j/A) \right) \\
[(\bigwedge_{i \in \{1, \dots, n\}} B_i \text{ says } \mathbf{p}_i) \rightarrow \bigwedge_{j \in J} \mathbf{q}_j]_A &= \left(\parallel_{j \in J} \text{OUT}(\mathbf{q}_j/A) \right) \mid \mathbf{p}_{n/B_n}^- \cdots \mathbf{p}_{1/B_1}^- \cdot \mathbf{0}
\end{aligned}$$

In Def. 2.6 below we extract from a PCL^- contract c the associated *latent actions*, i.e. the set of all atoms occurring in c paired with the participant who may have to abide by.

Definition 2.6. The function λ from PCL^- formulae to sets of actions is defined as follows:

$$\begin{aligned}\lambda(\bigwedge_{i \in I} A_i \text{ says } \alpha_i) &= \bigcup_{i \in I} \lambda_{A_i}(\alpha_i) \\ \lambda_A(\bigwedge_{j \in \mathcal{J}} \mathbf{q}_j) &= \{A \text{ says } \mathbf{q}_j \mid j \in \mathcal{J}\} \\ \lambda_A((\bigwedge_{i \in I} B_i \text{ says } \rho_i) \rightarrow \bigwedge_{j \in \mathcal{J}} \mathbf{q}_j) &= \{A \text{ says } \mathbf{q}_j \mid j \in \mathcal{J}\} \cup \{B_i \text{ says } \rho_i \mid i \in I\} \\ \lambda_A((\bigwedge_{i \in I} B_i \text{ says } \rho_i) \twoheadrightarrow \bigwedge_{j \in \mathcal{J}} \mathbf{q}_j) &= \{A \text{ says } \mathbf{q}_j \mid j \in \mathcal{J}\} \cup \{B_i \text{ says } \rho_i \mid i \in I\}\end{aligned}$$

The following result establishes a correspondence between our contract models. A PCL^- contract c requires to perform all the latent actions iff its encoding $[c]$ can be fulfilled.

Theorem 2.7. For all PCL^- formulae c involving principals $\{A_i\}_{i \in I}$, the following are equivalent:

- (contracts-as-formulae) $\forall c', \mu_1, \dots, \mu_n. ((c \xrightarrow{\mu_1} \dots \xrightarrow{\mu_n} c' \wedge \forall i \in I. c' \odot A_i) \implies \lambda(c) \subseteq \{\mu_1, \dots, \mu_n\})$
- (contracts-as-processes) $\exists c'. ([c] \rightarrow^* c' \wedge \forall i \in I. c' \odot A_i)$

The above statement can in fact be reduced to the result below by considering the definition of \odot in both contract models.

Theorem 2.8. For all PCL^- formulae c , $c \vdash \lambda(c)$ if and only if $[c] \rightarrow^* \mathbf{0}$.

The “if” direction mainly follows from the fact that, unless a τ^0 is fired, the residuals of $[c]$ are of the form $[c']$ where c' is a formula equivalent to c . The “only if” direction is proved by considering the order of applications of modus ponens in a proof of $c \vdash \lambda(c)$ and consequently firing the inputs corresponding to premises of implications. This makes their consequences/outputs available. The case of contractual implications is simpler because outputs are immediately available, hence their generated inputs can be fired last.

3 A basic calculus for contract-oriented computing

We now introduce the syntax and semantics of CO_2 . It generalises the contract calculus in [3] by making it independent of the actual contract language. In fact, CO_2 assumes the abstract model of contracts introduced in § 2, which can be instantiated to a variety of actual contract languages. While taking inspiration from Concurrent Constraint Programming, CO_2 makes use of more concrete communication primitives which do not assume a global constraint store, so reducing the gap towards a possible distributed implementation. The main differences between CO_2 and CCP are: (i) in CO_2 constraints are multisets of contracts, (ii) in CO_2 there is no global store of constraints: all the prefixes act on sessions, (iii) the prefix ask of CO_2 also instantiates variables to names, and (iv) the prefixes do (which makes a multiset of constraints evolve) and fuse (which establishes a new session) have no counterpart in CCP.

3.1 Syntax

First, let us define the syntax of CO_2 .

Definition 3.1. The abstract syntax of CO_2 is given by the following productions:

$$\begin{array}{lclclclcl} \text{Systems } S & ::= & \mathbf{0} & \mid & A[P] & \mid & s[C] & \mid & S \mid S & \mid & (u)S \\ \text{Processes } P & ::= & \downarrow_u c & \mid & \sum_i \pi_i.P_i & \mid & P \mid P & \mid & (u)P & \mid & X(\vec{u}) \\ \text{Prefixes } \pi & ::= & \tau & \mid & \text{do}_u a & \mid & \text{tell}_{u \downarrow_v} c & \mid & \text{ask}_{u, \vec{v}} \phi & \mid & \text{fuse}_u \phi \end{array}$$

We stipulate that the following conditions always hold: in a system $\|_{i \in I} n_i[P_i]$, $n_i \neq n_j$ for each $i \neq j \in I$; in a process $(u)P$ and a system $(u)S$, $u \notin \mathcal{N}_P$.

commutative monoidal laws for $ $ on processes and systems				
$u[(v)P] \equiv (v)u[P]$ if $u \neq v$	$Z (u)Z' \equiv (u)(Z Z')$ if $u \notin \text{fv}(Z) \cup \text{fn}(Z')$			
$(u)(v)Z \equiv (v)(u)Z$	$(u)Z \equiv Z$ if $u \notin \text{fv}(Z) \cup \text{fn}(Z)$			
$\downarrow_n c \equiv \mathbf{0}$	$\text{tell}_u \downarrow_n c.P \equiv \mathbf{0}$	$\text{ask}_{u,\bar{v}} \phi.P \equiv \mathbf{0}$ if $\bar{v} \cap \mathcal{N} \neq \emptyset$	$\text{fuse}_n \phi.P \equiv \mathbf{0}$	

Table 4: Structural equivalence (Z, Z' range over systems or processes)

We distinguish between *processes* and *systems*. Systems S consist of a set of *agents* $A[P]$ and of *sessions* $s[C]$, composed in parallel. Processes P comprise latent contracts, guarded summation, parallel composition, scope delimitation, and process identifiers. A *latent contract* $\downarrow_x c$ represents a contract c which has not been stipulated yet; upon stipulation, the variable x will be bound to a fresh session name. We allow finite prefix-guarded sums of processes; as usual, we write $\pi_1.P_1 + \pi_2.P_2$ for $\sum_{i=1,2} \pi_i.P_i$. Processes can be composed in parallel, and can be put under the scope of binders $(u)_-$. We use process identifiers X to express recursive processes, and we assume that each identifier has a corresponding defining equation $X(u_1, \dots, u_j) \stackrel{\text{def}}{=} P$ such that $\text{fv}(P) \subseteq \{u_1, \dots, u_j\} \subseteq \mathcal{V}$ and each occurrence of process identifiers Y in P is prefix-guarded. Note that principal names cannot be bound, but can be communicated if permitted by the contract language. The empty system and the empty sum are both denoted by $\mathbf{0}$. As usual, we may omit trailing occurrences of $\mathbf{0}$ in processes. Free/bound occurrences of variables/names are defined as expected. Variables and session names can be bound; in $(u)S$ all free occurrences of u in S are bound; S is *closed* when it has no free variables. Prefixes include the silent prefix τ , action execution $\text{do}_u a$, contract advertisement $\text{tell}_u \downarrow_v c$, contract query $\text{ask}_{u,\bar{v}} \phi$, and contract stipulation $\text{fuse}_u \phi$. The index u indicates the target agent/session where the prefix will be fired or the session where stipulations are kept, in the case of fuse . We shall refer to the set of latent contracts within an agent $A[P]$ as its *environment*; similarly, in a session $s[C]$ we shall refer to C as the environment of s . Note that the environment of agents can only contain latent contracts, advertised through the primitive tell , while sessions can only contain contracts, obtained either upon reaching an agreement with fuse , or possibly upon principals performing some do prefixes.

3.2 Semantics

The semantics of CO_2 is formalised by a reduction relation on systems which relies on the structural congruence laws in Table 4. Only the last row in Table 4 contains non-standard laws: they allow for collecting garbage terms which may possibly arise after variable substitutions.

Definition 3.2. *The binary relation \rightarrow on closed systems is the smallest relation closed under structural congruence and under the rules in Table 5, where the relation $K \triangleright_x^\sigma \phi$ in (FUSE) is introduced in Def. 3.3, and $\uparrow K$ is obtained by removing all the \downarrow_- from K , i.e. if $K = \parallel_{i \in I} \downarrow_{x_i} c_i$, then $\uparrow K = \parallel_{i \in I} c_i$. Also, we identify the parallel composition of contracts $C = c_1 | \dots | c_j$ with the multiset $\{c_1, \dots, c_j\}$ (similarly for latent contracts).*

Axiom (TAU) and rules (PAR), (DEL), and (DEF) are standard. Axioms (TELL₁) and (TELL₂) state that a principal A can advertise a latent contract $\downarrow_x c$ either in her own environment, or in a remote one. In

$A[\tau.P + P' \mid Q] \rightarrow A[P \mid Q]$	(TAU)
$A[\text{tell}_A \downarrow_x c.P + P' \mid Q] \rightarrow A[\downarrow_x A \text{ says } c \mid P \mid Q]$	(TELL ₁)
$A[\text{tell}_B \downarrow_x c.P + P' \mid Q] \mid B[R] \rightarrow A[P \mid Q] \mid B[R \mid \downarrow_x A \text{ says } c]$	(TELL ₂)
$\frac{C \xrightarrow{\langle A_1 \text{ says } a_1, \dots, A_j \text{ says } a_j \rangle} C' \quad j \geq 1}{s[C] \mid \prod_{1 \leq i \leq j} A_i[\text{do}_s a_i.P_i + P'_i \mid Q_i] \rightarrow s[C'] \mid \prod_{1 \leq i \leq j} A_i[P_i \mid Q_i]}$	(DO)
$\frac{\text{dom } \sigma = \vec{u} \subseteq \mathcal{V} \quad C\sigma \vdash \phi\sigma}{(\vec{u})(A[\text{ask}_{s,\vec{u}} \phi.P + P' \mid Q] \mid s[C] \mid S) \rightarrow A[P \mid Q]\sigma \mid s[C]\sigma \mid S\sigma}$	(ASK)
$\frac{K \triangleright_x^\sigma \phi \quad \vec{u} = \text{dom } \sigma \subseteq \mathcal{V} \quad s = \sigma(x) \text{ fresh}}{(\vec{u})(A[\text{fuse}_x \phi.P + P' \mid K \mid Q] \mid S) \rightarrow (s)(A[P \mid Q]\sigma \mid s[\uparrow K]\sigma \mid S\sigma)}$	(FUSE)
$\frac{X(\vec{u}) \stackrel{\text{def}}{=} P \quad P\{\vec{v}/\vec{u}\} \rightarrow P'}{X(\vec{v}) \rightarrow P'}$	(DEF)
$\frac{S \rightarrow S'}{S \mid S'' \rightarrow S' \mid S''}$	(PAR)
$\frac{S \rightarrow S'}{(u)S \rightarrow (u)S'}$	(DEL)

Table 5: Reduction semantics of CO₂

(DO), the principals A_i are participating in the contracts C stipulated on session s . Basically, the system evolves when all the involved principals are ready to fire the required action. Rule (ASK) allows A to check if an observable ϕ is entailed by the contracts C in session s ; notice that the entailment is subject to an instantiation σ of the variables mentioned in the ask prefix. Rule (FUSE) establishes a multi-party session s among all the parties that reach an agreement on the latent contracts K . Roughly, the *agreement* relation $K \triangleright_x^\sigma \phi$ (Def. 3.3) holds when, upon some substitution σ , the latent contracts K entail ϕ , and the fresh session name s substitutes for the variable x .

The simplest typical usage of these primitives is as follows. First, a group of principals exchanges latent contracts using `tell`, hence sharing their intentions. Then, one of them opens a new session using the `fuse` primitive. Once this happens, each involved principal A can inspect the session using `ask`, hence discovering her actual duties within that session: in general, these depend not only on the contract of A , but also on those of the other principals (see e.g. Ex.3.4). Finally, primitive `do` is used to actually perform the duties.

Example 3.1. *The sale scenario between seller A and buyer B from Ex. 2.4 can be formalized as follows.*

$$S = \begin{array}{l} A[(x,b) \text{tell}_A \downarrow_x ((b \text{ says pay}) \rightarrow \text{ship}). \text{fuse}_x (A \text{ says ship}). \text{do}_x \text{ship}] \\ | \quad B[(y) \text{tell}_A \downarrow_y \text{pay}. \text{ask}_y (B \text{ says pay}). \text{do}_y \text{pay}] \end{array}$$

The buyer tells A a contract which binds B to pay, then waits until discovering that he actually has to pay, and eventually does that. A session between the buyer and the seller is created and proceeds smoothly as expected:

$$\begin{aligned} S &\longrightarrow (x,b,y) (A[\downarrow_x A \text{ says } ((b \text{ says pay}) \rightarrow \text{ship}) | \text{fuse}_x (A \text{ says ship}). \text{do}_x \text{ship}] \\ &\quad | B[\text{tell}_A \downarrow_y \text{pay}. \text{ask}_y (B \text{ says pay}). \text{do}_y \text{pay}]) \\ &\longrightarrow (x,b,y) (A[\downarrow_y B \text{ says pay} | \downarrow_x A \text{ says } ((b \text{ says pay}) \rightarrow \text{ship}) | \text{fuse}_x (A \text{ says ship}). \text{do}_x \text{ship}] \\ &\quad | B[\text{ask}_y (B \text{ says pay}). \text{do}_y \text{pay}]) \\ &\longrightarrow (s) (A[\text{do}_s \text{ship}] | B[\text{ask}_s (B \text{ says pay}). \text{do}_s \text{pay}] | s[A \text{ says } ((B \text{ says pay}) \rightarrow \text{ship}), B \text{ says pay}]) \\ &\longrightarrow (s) (A[\text{do}_s \text{ship}] | B[\text{do}_s \text{pay}] | s[A \text{ says } ((B \text{ says pay}) \rightarrow \text{ship}), B \text{ says pay}]) \\ &\longrightarrow (s) (A[0] | B[\text{do}_s \text{pay}] | s[A \text{ says } ((B \text{ says pay}) \rightarrow \text{ship}), B \text{ says pay}, A \text{ says !ship}, \dots]) \\ &\longrightarrow (s) (A[0] | B[0] | s[A \text{ says } ((B \text{ says pay}) \rightarrow \text{ship}), B \text{ says pay}, A \text{ says !ship}, B \text{ says !pay}, \dots]) \end{aligned}$$

In the previous example, we have modelled the system outlined in Ex. 2.1 using a contracts-as-formulae approach. In the following example, we adopt instead the contracts-as-processes paradigms. In the meanwhile, we introduce a further participant to our system: a broker C which collects the contracts from A and B , and then uses a `fuse` to find when an agreement is possible.

Example 3.2. *Recall the contract of Ex. 2.2. We specify the behaviour of the system as:*

$$S \stackrel{\text{def}}{=} \begin{array}{l} A[(x)(\text{tell}_C (\downarrow_x \text{pay}^- . \text{ship}^0). \text{ask}_x \diamond \text{pay}^+. \text{do}_x \text{pay}^- . \text{do}_x \text{ship}^0)] \\ | \quad B[(y)(\text{tell}_C \downarrow_y \text{pay}^+ . \text{do}_y \text{pay}^+) \\ | \quad C[(z)(\text{fuse}_z \diamond (\text{pay}^+ \wedge \diamond \text{ship}^0))] \end{array}$$

The principals A and B advertise their contracts to the broker C , which opens a session for their interaction. As expected, as soon as the payment is received, the goods are shipped. After advertising such contract in her environment, the seller waits until finding that she has promised to ship; after that, she actually ships the item.

$$S \longrightarrow^* (s) (A[0] | B[0] | s[0])$$

3.3 On agreements

To find agreements ($K \triangleright_x^\sigma \phi$), we use the relation \vdash of the contract model.

Definition 3.3. For all multisets K of latent contracts, for all substitutions $\sigma : \mathcal{V} \rightarrow \mathcal{N}$, for all $x \in \mathcal{V}$, for all observables ϕ , we write $K \triangleright_x^\sigma \phi$ iff the following conditions hold:

- $x \in \text{dom } \sigma$
- $\text{fv}(K\sigma) = \text{fv}(\phi\sigma) = \emptyset$
- $\exists s \in \mathcal{N}_S : \forall y \in \text{dom } \sigma \cap \mathcal{V}'_S : \sigma(y) = s$
- $(\uparrow K)\sigma \vdash \phi\sigma$
- no $\sigma' \subset \sigma$ satisfies the conditions above, i.e. σ is minimal.

Basically, Def. 3.3 states that an agreement is reached when the latent contracts in K entail, under a suitable substitution, an observable ϕ . Recall that ϕ is the condition used by the principal acting as broker when searching for an agreement (cf. rule (FUSE) in Table 5). A valid agreement has to instantiate with a minimal substitution σ all the variables appearing in the latent contracts in K as well as the session variable x ; the latter, together with any other session variables in the domain of the substitution σ , is mapped to a (fresh) session name s .

The minimality condition on σ forces brokers to include in agreements only “relevant” participants. For instance, let $\downarrow_{x_1} c_1$, $\downarrow_{x_2} c_2$, and $\downarrow_{x_3} c_3$ be the latent contracts advertised to a broker; if there is an agreement on $\downarrow_{x_1} c_1$ and $\downarrow_{x_2} c_2$, the latent contract $\downarrow_{x_3} c_3$ would not be included. Basically, the minimality condition allows brokers to tell apart unrelated contracts. This is illustrated by the following informal example.

Example 3.3. Let ϕ be the observable “A shall ship the goods”, and consider the following latent contracts, advertised by A, B, and C, respectively:

- $\downarrow_{x_1} c_1 =$ “in session x_1 , if some principal b pays, then I shall ship the goods”
- $\downarrow_{x_2} c_2 =$ “in session x_2 , I shall pay”
- $\downarrow_{x_3} c_3 =$ “in session x_3 , I shall kiss a frog”.

Note that the first two latent contracts do entail ϕ when $\sigma(b) = B$, and $\sigma(x_1) = \sigma(x_2) = s$. Without the minimality condition, $\sigma(x_3) = s$ would have been included in the agreement of A and B, despite the offer of C being somewhat immaterial.

Example 3.4. Our approach allows for contract models with multiple levels of compliance. For instance, let c_A , c_B , and $c_{B'}$ be the following PCL contracts:

$$c_A = \downarrow_{x_1} A \text{ says } ((b \text{ says } b) \rightarrow a \wedge (b \text{ says } b') \rightarrow a') \quad c_B = \downarrow_{x_2} B \text{ says } b \quad c_{B'} = \downarrow_{x_3} B' \text{ says } b \wedge b'$$

The contract c_A is intuitively compliant with both c_B and $c_{B'}$. When coupled with $c_{B'}$, the contract c_A entails both the obligations a and a' for A. Conversely, when coupled with c_B , we obtain a weaker agreement, since only the obligation a is entailed. Although both levels of agreement are possible, in some sense the contract $c_{B'}$ provides c_A with a better Service-Level Agreement than c_B . Through the primitive ask, principal A can detect the actual service level she has to provide.

3.4 On violations

In Def. 3.4 below we set out when a principal A is honest in a given system S . Intuitively, we consider all the possible runs of S , and require that in every session the principal A eventually fulfils all her duties. To this aim, we shall exploit the fulfilment relation $C \odot A$ from the contract model.

To do that, we need to cope with a few technical issues. First, the α -conversion of session names makes it hard to track the *same* session at different points in a trace. So, we consider traces “without α -conversion of names”. Technically, let

$$\text{freezeNames}(S) = \{S' \mid S \equiv (s_1, \dots, s_j)S' \text{ and } S' \text{ free from } (s) \text{ delimitations}\}$$

and define $S \rightsquigarrow S''$ whenever $S \rightarrow S'$ and $S'' \in \text{freezeNames}(S')$. A \rightsquigarrow -trace is a trace w.r.t. \rightsquigarrow . Such a \rightsquigarrow -trace is *maximal* iff it is either infinite or ending with $S_j \not\rightsquigarrow$.

Another technical issue is that a principal could not get a chance to act in all the traces. For instance, consider the system $S = A[\text{do}_s \text{pay}] \mid B[X] \mid S'$ where S' enables A 's action and $X \stackrel{\text{def}}{=} \tau.X$; note that S generates the infinite trace $S \rightsquigarrow S \rightsquigarrow S \rightsquigarrow \dots$ in which A never pays, despite her honest intention. To account for this fact, we will check the honesty of a principal in *fair* traces, only, i.e. those obtained by running S according to a fair scheduling algorithm. More precisely, we say that \rightsquigarrow -trace $(S_i)_i$ is *fair* if no single prefix can be fired in an infinite number of S_i .

Definition 3.4. A principal A is honest in S iff for all maximal fair \rightsquigarrow -traces $(S_i)_{i \in I}$,

$$\forall s. \exists j \in I. \forall i \geq j, \vec{n}, C, S'. \left(S_i \equiv (\vec{n})(s[C] \mid S') \implies C \odot A \right)$$

In other words, a principal A misbehaves if involved in a session s such that the contracts C of s do not settle the obligations of A .

Example 3.5. Consider the variation of Ex. 3.1, where the seller is modified as follows:

$$\begin{aligned} S = & A[(x, b) \text{tell}_A \downarrow_x ((b \text{ says pay}) \rightarrow \text{ship}). \text{fuse}_x (A \text{ says ship}). \text{do}_x \text{snakeOil}] \\ & \mid B[(y) \text{tell}_A \downarrow_y \text{pay}. \text{ask}_y (B \text{ says pay}). \text{do}_y \text{pay}] \end{aligned}$$

The fraudulent seller A promises to ship — but eventually only provides the buyer with some snake oil. The interaction between A and B leads to a violation:

$$\begin{aligned} S \rightarrow^* (s) (A[\mathbf{0}] \mid B[\mathbf{0}] \mid s[A \text{ says } ((B \text{ says pay}) \rightarrow \text{ship}), A \text{ says !snakeOil}, \\ B \text{ says pay}, B \text{ says !pay}, \dots]) \end{aligned}$$

The buyer has not obtained what he has paid for. Indeed, the seller is dishonest according to Def. 3.4, because the contracts in s entail the promise A says ship, which is not fulfilled by any fact. A judge may thus eventually punish A for her misconduct.

3.5 On protection

We now illustrate some examples where one of the parties is fraudulent.

Example 3.6. Recall Ex. 3.5 and consider a fraudulent seller A , which promises in her contract some snake oil. The buyer B is unchanged from that example.

$$S = A[(x, b) \text{tell}_A \downarrow_x ((b \text{ says pay}) \rightarrow \text{snakeOil}). \text{fuse}_x (A \text{ says snakeOil}). \text{do}_x \text{snakeOil}] \mid B[\dots]$$

The interaction between A and B now goes unhappily for B: he will pay for some snake oil, and A is not even classified as dishonest according to Def. 3.4: indeed, A has eventually fulfilled all her promises.

$$S \longrightarrow^* (s) (A[0] \mid B[0] \mid s[A \text{ says !snakeOil}, B \text{ says !pay}, \dots])$$

Example 3.7. To protect the buyer from the fraud outlined in Ex. 3.6, we change the contract of the buyer B as follows:

$$\begin{aligned} S = & A[(x,b) \text{ tell}_A \downarrow_x ((b \text{ says pay}) \rightarrow \text{snakeOil}). \text{fuse}_x(A \text{ says snakeOil}). \text{do}_x \text{snakeOil}] \\ & \mid B[(a,y) \text{ tell}_A \downarrow_y (a \text{ says ship}) \rightarrow \text{pay}. \text{ask}_y(B \text{ says pay}). \text{do}_y \text{pay}] \end{aligned}$$

Note that we have used contractual implication \rightarrow , rather than standard implication \rightarrow , which allows B to reach an agreement also with the honest seller contract $(b \text{ says pay}) \rightarrow \text{ship}$. Instead, the interaction between the above fraudulent seller A and the buyer B will now get stuck on the fuse in A, because the available latent contracts do not entail $A \text{ says snakeOil}$.

When using contracts-as-processes, a broker can participate in deceiving a principal.

Example 3.8. Consider a simple e-commerce scenario:

$$\begin{aligned} S \stackrel{\text{def}}{=} & A_1[(x) \text{ tell}_B \downarrow_x (\text{pay}^+ \cdot \text{ship}^-). \text{do}_x \text{pay}^+ \cdot \text{do}_x \text{ship}^-] \\ & \mid A_2[(y) \text{ tell}_B \downarrow_y (\text{pay}^- \cdot (\text{ship}^+ + \text{fraud}^0)). \text{do}_y \text{pay}^- \cdot \text{do}_y \text{fraud}^0] \\ & \mid B[(z) \text{ fuse}_z \diamond (\text{ship}^+ \vee \text{fraud}^0)] \end{aligned}$$

Above, the broker B and A_2 dishonestly cooperate and open a session to swindle A_1 . The principal A_2 will be able to fulfil her contract $(C \odot A_2)$, while A_1 will never receive her goods. Nevertheless, A_1 is considered culpable, because he cannot perform the promised action ship^- . In Sect. 3.6 we propose a variant of the contracts-as-processes model allowing principals to protect from such kind of misbehavior.

Example 3.9. Consider the following formalization of the e-commerce scenario:

$$\begin{aligned} S \stackrel{\text{def}}{=} & A_1[(x, a_2) \text{ tell}_B \downarrow_x (a_2 \text{ says ship} \rightarrow \text{pay}). \text{ask}_x A_1 \text{ says pay}. \text{do}_x \text{pay}] \\ & \mid A_2[(y, a_1) \text{ tell}_B \downarrow_y (a_1 \text{ says pay} \rightarrow (\text{ship} \vee \text{fraud})). \text{do}_y \text{fraud}] \\ & \mid B[(z) \text{ fuse}_z \phi] \end{aligned}$$

Here, choose ϕ so to cause A_1 and A_2 to initiate a session (the actual formula is immaterial).

In Ex. 3.9, even if a session is established by the dishonest broker, A_1 will not pay, and he will not be considered culpable for that. Indeed, the prefix $\text{ask}_x A_1 \text{ says pay}$ is stuck because the contracts in the session do not entail any obligation for A_1 to pay. For the same reason, A_1 will fulfil her contract $(C \odot A_1)$.

In the following example, we show a different flavour of “protection”: the principals A and B are not protected by their contracts, but by the trusted escrow service that acts as a broker.

Example 3.10. Recall the scenario in Ex.2.3. The system is modelled as follows:

$$\begin{aligned} S \stackrel{\text{def}}{=} & A[(x) (\text{tell}_E \downarrow_x (\text{ship}E^+ \cdot \text{pay}^-). \text{do}_x \text{ship}E^+ \cdot \text{ask}_x \diamond \text{pay}^+ \cdot \text{do}_x \text{pay}^-)] \\ & \mid B[(y) (\text{tell}_E \downarrow_y (\text{pay}E^+ \cdot \text{ship}^-). \text{do}_y \text{pay}E^+ \cdot \text{ask}_y \diamond \text{ship}^+ \cdot \text{do}_x \text{ship}^-)] \\ & \mid E[(z) (\text{tell}_E \downarrow_z c_E \cdot \text{fuse}_z \phi' \cdot P)] \\ c_E \stackrel{\text{def}}{=} & \text{ship}E^- \cdot \text{pay}E^- \cdot (\text{pay}^+ \mid \text{ship}^+) + \text{pay}E^- \cdot \text{ship}E^- \cdot (\text{pay}^+ \mid \text{ship}^+) \end{aligned}$$

where P is the obvious realization of the c_E , and $\phi' = \diamond(\text{ship}E^+ \wedge \diamond \text{pay}^+) \wedge \diamond(\text{pay}E^+ \wedge \diamond \text{ship}^+)$.

The escrow service guarantees each of the participants A and B that the other party has to fulfil its obligation for the contract to be completed. The systems S can perform the wanted interaction:

$$S \rightarrow^* (s) (A[0] \mid B[0] \mid E[0] \mid s[0])$$

3.6 Variants to the basic calculus

Several variants and extensions are germane to CO_2 . We mention a few below.

Protection for contracts-as-processes. The contracts-as-processes model can be adapted so that CO_2 processes can protect themselves from untrusted brokers. In this variant, contractual obligations would derive only from mutually *compliant* contracts, i.e. $c \vdash \phi$ should hold only when all the (fair) traces of c lead to $\mathbf{0}$, in addition to $c \models_{\text{LTL}} \phi$. Similarly, $c \odot A$ should also hold on non-mutually-compliant c , since in this case no obligation arises. With this change, even if a principal A is somehow put in a session with fraudulent parties, A can discover (via *ask*) that no actual obligation is present and avoid being swindled.

Contracts-as-processes with explicit sender and receiver. The syntax of contracts-as-processes (Def. 2.2) can be extended so to make explicit the intended senders/recipients of inputs/outputs (permitting e.g. to clearly state who is paying whom). For this, we could use e.g. $\text{pay}^+ @ B$ as atoms, and adapt the semantics $\xrightarrow{\mu}$ accordingly.

Note that the logic for observables Φ in Def. 2.2 should be adapted as well. Indeed, LTL does not distinguish between actions performed by distinct principals. To this purpose, we would allow $A \text{ says } a^+ @ B$ (and related a^- , a^0 variants) as prime formulae, so that it is now possible to observe who are the principals involved by an action.

Note that the changes discussed above do not alter the general calculus CO_2 , but merely propose a different contract model.

Local actions. In CO_2 , agents $A[P]$ only carry latent contracts in P . Hence, communication between principals is limited to latent contract exchange. Allowing general data exchange in CO_2 can be done in a natural way by following CCP. Basically, P would include CCP constraints t , ranging over a constraint system $\langle T, \vdash \rangle$, a further parameter to CO_2 . Assume that $A \text{ says } t \in T$ for all $t \in T$ and for all A . The following rules will augment the semantics of CO_2 (the syntax is extended accordingly):

$$\begin{aligned} A[\text{tell}_A t.P + P' \mid Q] &\rightarrow A[A \text{ says } t \mid P \mid Q] \\ A[\text{tell}_B t.P + P' \mid Q] \mid B[R] &\rightarrow A[P \mid Q] \mid B[A \text{ says } t \mid R] \\ A[\text{ask}_A t.P + P' \mid T \mid Q] &\rightarrow A[P \mid T \mid Q] \quad \text{provided that } T \vdash t \end{aligned}$$

Note that the above semantics does not allow A to corrupt the constraint store of B augmenting it with arbitrary constraints. Indeed, exchanged data is automatically tagged on reception with the name of the sender. So, in the worst case, a malicious A can only insert garbage $A \text{ says } t$ into the constraint store of B .

Retracting latent contracts. A retract primitive could allow a principal A to remove a latent contract of hers after its advertisement. Therefore, A could change her mind until her latent contract is actually used to establish a session, where A is bound to her duties.

Consistency check. The usual check t primitive from CCP, which checks the consistency of the constraint store with t , can also be added to CO_2 . When check t is executed by a principal $A[P]$, t is checked for consistency against the constraints in P . Note that checking the whole world would be unfeasible in a distributed system.

Forwarding latent contracts. A forward primitive could allow a broker A to move a latent contract from her environment to that of another broker B, without tagging it with $A \text{ says}$. In this way A delegates to B the actual opening of a session.

Remote queries. More primitives to access the remote principals could be added. Note however that while it would be easy e.g. to allow $A[\text{ask}_B t]$ to query the constraint of B, this would probably be undesirable for security reasons. Ideally, B should be allowed to express whether A can access his own constraint store. This requires some access control mechanism.

4 Conclusions

We have developed a formal model for reasoning about contract-oriented distributed programs. The overall contribution of this paper is a contract calculus (CO_2) that is parametric in the choice of contract model. In CO_2 , principals can advertise their own contracts, find other principals with compatible contracts, and establish a new multi-party session with those which comply with the global contract. We have set out two crucial issues: how to reach agreements, and how to detect violations. We have presented two concretisations of the abstract contract model. The first is an instance of the contracts-as-processes paradigm, while the second is an instance of the contracts-as-formulae paradigm.

As a first step towards relating contract-as-processes and contracts-as-formulae, we have devised a mapping from contracts based on the logic PCL [3] into CCS-like contracts. One can then use contract-as-formulae at design-time, reason about them using the entailment relation of PCL, and then concretise them to contracts-as-processes through the given mapping. Theorem 2.7 guarantees that contracts-as-processes can reach success in those cases in which an agreement would be possible in the logic model, hence providing a connection between the two worlds.

Acknowledgments. This work has been partially supported by Autonomous Region of Sardinia Project L.R. 7/2007 TESLA, by PRIN Project SOFT (Tecniche Formali Orientate alla Sicurezza) and by the Leverhulme Trust Programme Award “Tracing Networks”.

References

- [1] Alexander Artikis, Marek J. Sergot & Jeremy V. Pitt (2009): *Specifying norm-governed computational societies*. *ACM Trans. Comput. Log.* 10(1), doi:10.1145/1459010.1459011.
- [2] Massimo Bartoletti & Roberto Zunino (2009): *A logic for contracts*. Technical Report DISI-09-034, DISI - Università di Trento.
- [3] Massimo Bartoletti & Roberto Zunino (2010): *A Calculus of Contracting Processes*. In: *LICS*, doi:10.1109/LICS.2010.25.
- [4] Massimo Bartoletti & Roberto Zunino (2010): *Primitives for Contract-based Synchronization*. In: *ICE*.
- [5] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): *A theory of design-by-contract for distributed multiparty interactions*. In: *CONCUR*, doi:10.1007/978-3-642-15375-4_12.
- [6] Mario Bravetti & Gianluigi Zavattaro (2007): *Towards a Unifying Theory for Choreography Conformance and Contract Compliance*. In: *Software Composition*, doi:10.1007/978-3-540-77351-1_4.
- [7] Maria Grazia Buscemi & Hernán C. Melgratti (2007): *Transactional Service Level Agreement*. In: *TGC*, doi:10.1007/978-3-540-78663-4_10.

- [8] Maria Grazia Buscemi & Ugo Montanari (2007): *CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements*. In: *ESOP*, doi:10.1007/978-3-540-71316-6_3.
- [9] Felice Cardone (2011): *The geometry and algebra of commitment*. In: *Ludics, dialogue and interaction*.
- [10] Samuele Carpineti & Cosimo Laneve (2006): *A Basic Contract Language for Web Services*. In: *ESOP*, doi:10.1007/11693024_14.
- [11] Giuseppe Castagna, Nils Gesbert & Luca Padovani (2009): *A theory of contracts for Web services*. *ACM Transactions on Programming Languages and Systems* 31(5), doi:10.1145/1538917.1538920.
- [12] Giuseppe Castagna & Luca Padovani (2009): *Contracts for Mobile Processes*. In: *Proc. CONCUR*, doi:10.1007/978-3-642-04081-8_15.
- [13] Mario Coppo & Mariangiola Dezani-Ciancaglini (2008): *Structured Communications with Concurrent Constraints*. In: *TGC*, doi:10.1007/978-3-642-00945-7_7.
- [14] E. Allen Emerson (1990): *Temporal and Modal Logic*. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, North-Holland Pub. Co./MIT Press.
- [15] Gian Luigi Ferrari & Alberto Lluch-Lafuente (2006): *A Logic for Graphs with QoS*. *ENTCS* 142, doi:10.1016/j.entcs.2004.10.030.
- [16] Deepak Garg & Martín Abadi (2008): *A Modal Deconstruction of Access Control Logics*. In: *FoSSaCS*, doi:10.1007/978-3-540-78499-9_16.
- [17] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Disciplines for Structured Communication-based Programming*. In: *ESOP*, doi:10.1007/BFb0053567.
- [18] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL*, doi:10.1145/1328438.1328472.
- [19] Ashley McNeile (2010): *Protocol contracts with application to choreographed multiparty collaborations*. *SOCA* 4, doi:10.1007/s11761-010-0060-9.
- [20] Robin Milner (1989): *Communication and concurrency*. Prentice-Hall, Inc.
- [21] Cristian Prisacariu & Gerardo Schneider (2007): *A Formal Language for Electronic Contracts*. In: *FMOODS*, doi:10.1007/978-3-540-72952-5_11.
- [22] Vijay Saraswat, Prakash Panangaden & Martin Rinard (1991): *Semantic Foundations of Concurrent Constraint Programming*. In: *POPL*, doi:10.1145/99583.99627.
- [23] Anne Troelstra & Dirk van Dalen (1988): *Constructivism in Mathematics, vol. 1*. North-Holland.