A CALCULUS OF CONTRACTING PROCESSES

Massimo Bartoletti and Roberto Zunino

October 2009 (submitted), June 2010 (revised)

Technical Report #DISI-09-056

A Calculus of Contracting Processes

Massimo Bartoletti
Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari

Roberto Zunino
Dipartimento di Ingegneria e Scienza dell'Informazione, Università degli Studi di Trento

Abstract

We propose a formal theory for contract-based computing. A contract is an agreement stipulated between two or more parties, which specifies the duties and the rights of the parties involved therein. We model contracts as formulae in an intuitionistic logic extended with a "contractual" form of implication. Decidability holds for our logic: this allows us to mechanically infer the rights and the duties deriving from any set of contracts. We embed our logic in a core calculus of contracting processes, which combines features from concurrent constraints and calculi for multiparty sessions, while subsuming several idioms for concurrency. We then show how to exploit our calculus as a tool for modelling services, the interaction of which is driven by contracts.

1 Introduction

Security, trustworthiness and reliability of software systems are crucial issues in the rising Information Society. As new online services (e-commerce, e-banking, e-government, *etc.*) are made available, the number and the criticality of the problems related to possible misbehaviour of services keeps growing. Nevertheless, these problems are hardly dealt with in current practice, especially from the client point of view.

The typical dynamics of a Web transaction is that a client chooses a service provider that she trusts, and then proceeds interacting with it, without any provable guarantee that she will eventually obtain the required feature. Standard service infrastructures are focussed on protecting services from undesired interactions, while little effort is devoted to protecting clients. As one can see, the situation is quite unbalanced: clients have to operate with no concrete guarantees, and all they can do is to trust the service providers. At best, the service provider commits himself to respect some "service level agreement". In the case this is not honoured, the only thing the client can do is to take legal steps against the provider. Although this is the normal practice nowadays, it is highly desirable to reverse this trend. Indeed, both clients and services could incur relevant expenses due to the needed legal disputes. This is impractical, especially for transactions dealing with small amounts of money.

We propose to balance this situation, by regulating the interaction among parties by a suitable *contract*. A contract subordinates the behaviour promised by a client (e.g. "I will pay for a service X") to the behaviour promised by a service (e.g. "I will provide you with a service Y"), and *vice versa.* The crucial problems are then how to define the concept of contract, how to understand when a set of contracts gives rise to an agreement among the stipulating parties, and how to actually enforce this agreement, in an environment – the Internet – which is by definition open and unreliable.

1.1 An example

To give the intuition about our contracts, suppose there are two kids, Alice and Bob, who want to play together. Alice has a toy airplane, while Bob has a bike. Both Alice and Bob wish to play with each other's toy. Before sharing their toys, Alice and Bob stipulate the following "gentlemen's agreement":

Alice: I will lend my airplane to you, Bob, provided that I borrow your bike.

Bob: I will lend my bike to you, Alice, provided that I borrow your airplane.

From the above two contracts, we want to formally deduce that Alice and Bob will indeed share their toys, provided that they are real "gentlemen" who always respect their promises. Let us write a for the atomic proposition "Alice lends her airplane" and b for "Bob lends his bike". A (wrong) formalisation of the above commitments in classical propositional logic could be the following, using implication $\rightarrow$. Alice's commitment A is represented as $\mathsf{b} \rightarrow \mathsf{a}$ and Bob's commitment B as $\mathsf{a} \rightarrow \mathsf{b}$. While the above commitments agree with our intuition, they are not enough to deduce that Alice will lend her airplane and Bob will lend his bike. Formally, it is possible to make true the formula $A \wedge B$ by assigning false to both propositions a and b.

The failure to represent scenarios like the one above seems related to the Modus Ponens rule: to deduce b from $\mathsf{a} \rightarrow \mathsf{b}$, we need to prove a. That is, we could deduce that Bob lends his bike, but only *after* Alice has lent Bob her airplane. So, one of the two parties must "take the first step". In a logic for mutual agreements, we would like to deduce $\mathsf{a} \wedge \mathsf{b}$ whenever $A \wedge B$ is true, without requiring any party to take the first step. That is, A and B are *contracts*, that once stipulated imply the duties promised by all the involved parties. Notice that $A \wedge B \rightarrow \mathsf{a} \wedge \mathsf{b}$ does *not* hold neither in classical nor in intuitionistic propositional calculus IPC [37], where the behaviour of implication strictly adheres to Modus Ponens.

To model contracts, we then extend IPC with a new form of implication, which we denote with the symbol $\twoheadrightarrow$. The resulting logic is called PCL, for Propositional Contract Logic. For instance, the contract declared by Alice, "I will lend my airplane to Bob provided that Bob lends his bike to me", will be written $\mathsf{b} \twoheadrightarrow \mathsf{a}$. This form of, say, *contractual* implication, is stronger than the standard implication $\rightarrow$ of IPC. Actually, $\mathsf{b} \twoheadrightarrow \mathsf{a}$ implies a not only when b is true, like IPC implication, but also in the case that a "compatible" contract, e.g. $\mathsf{a} \twoheadrightarrow \mathsf{b}$, holds. In our scenario, this means that Alice will lend her airplane to Bob, provided that Bob agrees to lend his bike to Alice whenever he borrows Alice's airplane, and *vice versa.* Actually, the following formula is a theorem of our logic:

$$(\mathsf{b} \twoheadrightarrow \mathsf{a}) \;\wedge\; (\mathsf{a} \twoheadrightarrow \mathsf{b}) \;\rightarrow\; \mathsf{a} \wedge \mathsf{b}$$

In other words, from the "gentlemen's agreement" stipulated by Alice and Bob, we can deduce that the two kids will indeed share their toys.

All the above shows how to deduce, from a set of contracts, the rights and the duties of the involved parties. Yet, it says nothing about the actual dynamics of these parties, that is how to model their behaviour. To do that, we introduce a calculus of contracting processes, which embeds our logic for contracts. This calculus belongs to the family of concurrent constraints calculi [35, 7], with the peculiarity that in ours the constraints are PCL formulae. A process can assert a constraint c (a PCL formula) through the primitive $\mathsf{tell}\, c$. For instance, the following process models Alice exposing her contract:

$$(x)\ \mathsf{tell}\, \mathsf{b}(x) \twoheadrightarrow \mathsf{a}(x)$$

Formally, this will add $\mathsf{b}(x) \twoheadrightarrow \mathsf{a}(x)$ to the set of constraints. The formal parameter x will be bound to the identifier of the actual session established between Alice and Bob. As it happens for sessions centered calculi [38, 13], sessions are an important aspect also in our calculus, since they allow for distinguishing among different instantiations of the same contract. The outer (x) is a scope delimitation for the variable x, similarly to the Fusion calculus [30].

After having exposed her contract, Alice will wait until finding that she has actually to lend her airplane to Bob. To discover that, Alice uses the primitive $\mathsf{fuse}_x\, c$ as follows:

$$\mathsf{fuse}_x\, \mathsf{a}(x)$$

This is similar to the $\mathsf{ask}\, c$ primitive of concurrent constraints, which probes the constraints to see whether they entail the constraint c. Yet, our fuse has some crucial peculiarities. Besides probing the constraints to find whether they entail c, $\mathsf{fuse}_x\, c$ also binds the variable x to an actual session identifier, shared among all the parties involved in the contract. In other words, $\mathsf{fuse}_x\, c$ actually fuses all the processes agreeing on a given contract.

Summing up, we will model the behaviour of Alice as the following process:

$$\mathit{Alice} = (x)\ \big(\mathsf{tell}\, \mathsf{b}(x) \twoheadrightarrow \mathsf{a}(x).\ \mathsf{fuse}_x\, \mathsf{a}(x).\ \mathit{lendAirplane}\big)$$

where the process *lendAirplane* (no further specified) models the action of Alice actually lending her airplane to Bob. The overall behaviour of Alice is then as follows: *(i)* issue the contract; *(ii)* wait until discovering the duty of lending the airplane; *(iii)* finally, lend the airplane.

Dually, we model the behaviour of Bob as the following process in our calculus:

$$\mathit{Bob} = (y)\ \big(\mathsf{tell}\, \mathsf{a}(y) \twoheadrightarrow \mathsf{b}(y).\ \mathsf{fuse}_y\, \mathsf{b}(y).\ \mathit{lendBike}\big)$$

A possible interaction of Alice and Bob will then be the following, where n stands for a fresh session identifier:

$$\mathit{Alice} \mid \mathit{Bob}\ \rightarrow^*\ (n)\ \big(\mathit{lendAirplane}\{n/x\} \mid \mathit{lendBike}\{n/y\}\big)$$

As expected, the resulting process shows Alice and Bob actually sharing their toys, in the session identified by n.

2 A Logic for Contracts

We now introduce our theory of contracts. We start in Sect. 2.1 by characterizing our logic through a set of properties that we would expect to be enjoyed by any logic for contracts. In Sect. 2.2 we present the syntax of PCL. In Sect. 2.3 we synthesize a minimal set of Hilbert-style axioms for PCL that imply all the desirable properties, and we show the resulting logic consistent (Theorem 1). We then propose a sequent calculus for PCL, which is shown to be equivalent to the Hilbert system (Theorem 2). The hard result is Theorem 3, where we establish cut elimination for our sequent calculus. Together with the subformula property (Theorem 4), this paves us the way to state decidability for PCL in Theorem 5. In Sect. 2.4 we give further details and examples about using the logic PCL to model a variety of contracts.

2.1 Desirable properties

We now characterize, with the help of some examples, the desirable properties of any logic for contracts.

As shown in the Sect. 1, a characterizing property of contractual implication is that of allowing two dual contracting parties to "handshake", so to make their agreement effective. This is resumed by the following *handshaking* property:

$$\vdash (p \twoheadrightarrow q) \land (q \twoheadrightarrow p) \;\rightarrow\; p \land q \tag{1}$$

A generalisation of the above property to the case of n contracting parties is also desirable. It is a sort of "circular" handshaking, where the $(i+1)$-th party, in order to promise some duty p_{i+1}, relies on a promise p_i made by the i-th party (in a circular fashion, the first party relies on the promise of the last one). In the case of n parties, we would expect the following:

$$\vdash (p_1 \twoheadrightarrow p_2) \land \cdots \land (p_{n-1} \twoheadrightarrow p_n) \land (p_n \twoheadrightarrow p_1) \;\;\rightarrow\;\; p_1 \land \cdots \land p_n \tag{2}$$

As a concrete example, consider an e-commerce scenario where a Buyer can buy items from a Seller, and pay them through a credit card. To mediate the interaction between the Buyer and the Seller, there is a Bank which manages payments. The contracts issued by the three parties could then be:

Buyer: I will click "pay" provided that the Seller will ship my item

Seller: I will ship your item provided that I get the money

Bank: I will transfer money to the Seller provided that the Buyer clicks "pay".

Let the atomic propositions ship, $\mathsf{clickpay}$, and pay denote respectively the facts "Seller ships item", "Buyer clicks pay", and "Bank transfers money". The above contracts can then be modelled as:

$$\mathit{Buyer} = \mathsf{ship} \twoheadrightarrow \mathsf{clickpay} \qquad \mathit{Bank} = \mathsf{clickpay} \twoheadrightarrow \mathsf{pay} \qquad \mathit{Seller} = \mathsf{pay} \twoheadrightarrow \mathsf{ship}$$

Then, by the handshaking property (2), we obtain a successful transaction:

$$\vdash \mathit{Buyer} \land \mathit{Bank} \land \mathit{Seller} \;\rightarrow\; \mathsf{pay} \land \mathsf{ship}$$

Note that, in the special case that n equals 1, the above "circular" handshaking property turns into a particularly simple form:

$$\vdash (p \twoheadrightarrow p) \;\rightarrow\; p \tag{3}$$

Intuitively, (3) can be interpreted as the fact that promising p provided that p, implies p (actually, also the converse holds, so that promise is equivalent to p). It also follows from (1) when $p = q$.

Another generalisation of the toy-exchange scenario of Sect. 1 to the case of n kids is also desirable. It is a sort of "greedy" handshaking property, because now a party promises p_i only provided that *all* the other parties promise their duties, i.e. $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n$. The greedy handshaking can be stated as:

$$\vdash \bigwedge_{i \in 1..n} \Big((p_1 \land \ldots \land p_{i-1} \land p_{i+1} \land \ldots \land p_n) \twoheadrightarrow p_i \Big) \;\; \rightarrow \;\; p_1 \land \cdots \land p_n \qquad (4)$$

So far, we have devised some characterizing properties of handshakings. We will now focus on further logical properties of contractual implication. As shown by (1), a contract $p \twoheadrightarrow q$ becomes effective, i.e. implies the promise q, when it is matched by a dual contract $q \twoheadrightarrow p$. Even more directly, $p \twoheadrightarrow q$ should be effective also in the case that the premise p is already true:

$$\vdash p \land (p \twoheadrightarrow q) \;\rightarrow\; q \qquad (5)$$

In other words, contractual implication should be *stronger* than standard implication, i.e. we expect that the following is a theorem of any logic for contracts:

$$\vdash (p \twoheadrightarrow q) \rightarrow (p \rightarrow q) \qquad (6)$$

On the other hand, we do not want that also the converse holds, since this would equate the two forms of implication: that is, $\nvdash (p \rightarrow q) \rightarrow (p \twoheadrightarrow q)$.

We want contractual implication to share with standard implication a number of properties. We discuss some of them below. First, a contract that promises true (written $\top$) is always satisfied, regardless of the precondition. So, we expect the following tautology:

$$\vdash p \twoheadrightarrow \top \qquad (7)$$

However, differently from standard implication, we do not want that a contract with a false precondition (written $\bot$) always holds, i.e. $\nvdash \bot \twoheadrightarrow p$. To see why, assume that $\bot \twoheadrightarrow p$ is a tautology, for all p. Then, it would also be the case for $p = \bot$, and so by (3) we would deduce a contradiction: $(\bot \twoheadrightarrow \bot) \rightarrow \bot$.

Another property of implication that we want to preserve is transitivity:

$$\vdash (p \twoheadrightarrow q) \land (q \twoheadrightarrow r) \;\rightarrow\; (p \twoheadrightarrow r) \qquad (8)$$

Back to our previous example, transitivity would allow the promise of the Buyer ($\mathsf{ship} \twoheadrightarrow \mathsf{clickpay}$) and the promise of the Bank ($\mathsf{clickpay} \twoheadrightarrow \mathsf{pay}$) to be combined in the promise $\mathsf{ship} \twoheadrightarrow \mathsf{pay}$.

Contractual implication should also enjoy a stronger form of transitivity. We illustrate it with the help of an example. Suppose an air-flight customer who wants to book a flight. To do that, she issues the following contract:

$$\mathit{Customer} : \mathsf{bookFlight} \twoheadrightarrow \mathsf{pay}$$

This contract states that the customer promises to pay the required amount, provided that she obtains a flight reservation. Suppose now that an airline company starts a special offer, in the form of a free drink for each customer:

$$\mathit{AirLine} : \mathsf{pay} \twoheadrightarrow \mathsf{bookFlight} \land \mathsf{freeDrink}$$

Of course, the two contracts should give rise to an agreement, because the airline company is promising a better service than the one required by the customer contract. To achieve that, we expect to be able to "weaken" the contract of the airline company, to make it match the contract issued by the customer:

$$\vdash \mathit{AirLine} \quad \rightarrow \quad (\mathsf{pay} \twoheadrightarrow \mathsf{bookFlight})$$

Alternatively, one could make the two contracts match by making stronger the precondition required by the customer, that is:

$$\vdash \mathit{Customer} \quad \rightarrow \quad (\mathsf{bookFlight} \ \wedge \ \mathsf{freeDrink} \twoheadrightarrow \mathsf{pay})$$

More in general, we want the following two properties hold for any logic for contracts. They say that the promise in a contract can be arbitrarily weakened (9), while the precondition can be arbitrarily strengthened (10).

$$\vdash (p \twoheadrightarrow q) \ \wedge \ (q \rightarrow q') \ \rightarrow \ (p \twoheadrightarrow q') \tag{9}$$

$$\vdash (p' \rightarrow p) \ \wedge \ (p \twoheadrightarrow q) \ \rightarrow \ (p' \twoheadrightarrow q) \tag{10}$$

Note that the properties (8), (9), (10) cover three of the four possible cases of transitivity properties which mix standard and contractual implication. Observe, instead, that the fourth case would make standard and contractual implications equivalent, so it is *not* a desirable property.

Another property that should hold is that, if a promise q is already true, then it is also true any contract which promises q:

$$\vdash q \ \rightarrow \ (p \twoheadrightarrow q) \tag{11}$$

Of course, we do not want the converse to hold: it is not always the case that a contract implies its promise: $\nvdash (p \twoheadrightarrow q) \rightarrow q$.

2.2 Syntax

The syntax of PCL extends that of IPC. It includes the standard logic connectives $\neg, \wedge, \vee, \rightarrow$ and the contractual implication connective $\twoheadrightarrow$. We assume a denumerable set $\{\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \ldots\}$ of prime (atomic) formulae. Arbitrary PCL formulae are denoted with the letters $p, q, r, s, \ldots$ (note that the font differs from that used for prime formulae). The precedence of IPC operators is the following, from highest to lowest: $\neg, \wedge, \vee, \rightarrow$. We stipulate that $\twoheadrightarrow$ has the same precedence as $\rightarrow$.

Definition 1. *The formulae of* PCL *are inductively defined as:*

$$p \ ::= \ \bot \ \mid \ \top \ \mid \ \mathsf{p} \ \mid \ \neg p \ \mid \ p \vee p \ \mid \ p \wedge p \ \mid \ p \rightarrow p \ \mid \ p \twoheadrightarrow p$$

We let $p \leftrightarrow q$ be syntactic sugar for $(p \rightarrow q) \wedge (q \rightarrow p)$.

2.3 Axiomatization

We now present an Hilbert-style axiomatization for PCL.

Definition 2. *The proof system of* PCL *comprises all the axioms of IPC, the Modus Ponens rule* [Cut]*, and the following additional axioms.*

$$\begin{array}{ll} \top \twoheadrightarrow \top & [\textsc{Zero}] \\ (p \twoheadrightarrow p) \to p & [\textsc{Fix}] \\ (p' \to p) \to (p \twoheadrightarrow q) \to (q \to q') \to (p' \twoheadrightarrow q') & [\textsc{PrePost}] \end{array}$$

The above axioms are actually a subset of the properties discussed in Sect. 2.1. The axiom Zero is a subcase of (7), the axiom Fix is just (3), while the axiom PrePost combines (9) and (10). As expected, this set of axioms is actually sound and complete w.r.t. all the properties marked as desirable in Sect. 2.1.

Lemma 1. *The properties* (1)–(11) *are theorems of* PCL*. Also, we have:*

$$\begin{array}{l} \vdash (p \twoheadrightarrow q) \wedge (q \twoheadrightarrow r) \to (p \twoheadrightarrow (q \wedge r)) \\ \vdash (p \twoheadrightarrow (q \wedge r)) \to (p \twoheadrightarrow q) \wedge (p \twoheadrightarrow r) \\ \vdash (p \twoheadrightarrow q) \vee (p \twoheadrightarrow r) \to (p \twoheadrightarrow (q \vee r)) \\ \vdash (p \twoheadrightarrow q) \to ((q \to p) \to q) \end{array}$$

We present below some of the most significant results about our logic. For a more comprehensive account, including detailed proofs of all our results, see [4].

Theorem 1. PCL *is consistent, i.e.* $\nvdash \bot$.

As expected, the following formulae are *not* tautologies of PCL :

$$\begin{array}{ll} \nvdash (p \to q) \to (p \twoheadrightarrow q) & \nvdash (p \twoheadrightarrow q) \to q \\ \nvdash \bot \twoheadrightarrow p & \nvdash ((q \to p) \to q) \to (p \twoheadrightarrow q) \end{array}$$

Note that if we augment our logic with the axiom of excluded middle, then $(p \twoheadrightarrow q) \leftrightarrow q$ becomes a theorem, so making contractual implication trivial. For this reason we use IPC, instead of classical logic, as the basis of PCL .

Another main result about PCL is its decidability. To prove that, we have devised a Gentzen-style sequent calculus, which is equivalent to the Hilbert-style axiomatisation. In particular, we have extended the sequent calculus for IPC presented in [32] with rules for the contractual implication connective $\twoheadrightarrow$.

Definition 3. *The sequent calclus of* PCL *includes all the rules for IPC [4], and the following additional rules.*

$$\frac{\Gamma \vdash q}{\Gamma \vdash p \twoheadrightarrow q}\,[\textsc{Zero}] \qquad \frac{\begin{array}{c}\Gamma, p \twoheadrightarrow q, r \vdash p \\ \Gamma, p \twoheadrightarrow q, q \vdash r\end{array}}{\Gamma, p \twoheadrightarrow q \vdash r}\,[\textsc{Fix}] \qquad \frac{\begin{array}{c}\Gamma, p \twoheadrightarrow q, a \vdash p \\ \Gamma, p \twoheadrightarrow q, q \vdash b\end{array}}{\Gamma, p \twoheadrightarrow q \vdash a \twoheadrightarrow b}\,[\textsc{PrePost}]$$

We now establish the equivalence between the two logical systems of PCL. In the following theorem, we denote with $\vdash_H$ provability in the Hilbert-style system, while $\vdash_G$ is used for Gentzen-style provability.

Theorem 2. *For all* PCL *formulae* p*, we have that:* $\vdash_H p \iff \emptyset \vdash_G p$.

Our sequent calculus enjoys cut elimination. The proof is non-trivial, since the rules for $\twoheadrightarrow$ are not dual, unlike e.g. left/right rules for $\wedge$. Nevertheless, the structural approach of [32] can be adapted. Full details about proofs are in [4].

Theorem 3 (Cut Elimination)**.** *If p is provable in* PCL*, then there exists a proof of p which does not use the* Cut *rule.*

The consistency result can be extended to negation-free formulae. This will be useful in Sect. 3, where we will define our calculus.

Lemma 2. *If p is free from $\{\bot, \neg\}$, then $\nvdash p \to \bot$.*

The subformula property holds in PCL. Cut-free proofs only involve subformulae of the sequent at hand.

Theorem 4 (Subformula Property)**.** *If D is a cut-free proof of $\Gamma \vdash p$, the formulae occurring in D are subformulae of those occurring in Γ and p.*

Decidability then follows from theorems 3 and 4.

Theorem 5. *The logic* PCL *is decidable.*

As a further support to our logic, we have implemented a proof search algorithm, which decides whether any given formula is a tautology or not. Despite the problem being PSPACE complete [36], the performance of our tool is acceptable for the examples presented in this paper. Our tool is available at [31].

We now establish some expressiveness results, relating PCL and IPC. More in detail, we consider whether sound and complete homomorphic encodings exist, that is, whether $\twoheadrightarrow$ can be regarded as syntactic sugar for some IPC context.

Definition 4. *A homomorphic encoding m is a function from PCL formulae to IPC formulae such that: m is the identity on prime formulas, $\top$, and $\bot$; it acts homomorphically on $\wedge, \vee, \to, \neg$; it satisfies $m(p \twoheadrightarrow q) = \mathcal{C}[m(p), m(q)]$ for some fixed IPC context $\mathcal{C}(\bullet, \bullet)$.*

Of course, each homomorphic encoding is uniquely determined by the context $\mathcal{C}$. Several *complete* encodings exist:

Lemma 3. *The following homomorphic encodings are complete, i.e. they satisfy $\vdash p \implies \vdash_{IPC} m_i(p)$. Moreover, they are pairwise non-equivalent in IPC.*

$$m_0(p \twoheadrightarrow q) = m_0(q) \qquad m_1(p \twoheadrightarrow q) = (m_1(q) \to m_1(p)) \to m_1(q)$$
$$m_2(p \twoheadrightarrow q) = \neg\neg(m_2(q) \to m_2(p)) \to m_2(q) \qquad m_3(p \twoheadrightarrow q) = \neg(m_3(q) \to m_3(p)) \vee m_3(q)$$
$$m_4(p \twoheadrightarrow q) = ((m_4(q) \to m_4(p)) \vee \mathsf{a}) \to m_4(q)$$

where a is any prime formula.

However, there can be no *sound* encodings, so $\twoheadrightarrow$ is not just syntactic sugar.

Theorem 6. *If m is a homomorphic encoding of* PCL *into IPC, then m is not sound, i.e. there exists a* PCL *formula p such that $\vdash_{IPC} m(p)$ and $\nvdash p$.*

In [4] we have proved further properties of PCL, including some relations between PCL and IPC, the modal logic S4, and propositional lax logic. Also, we have explored there further properties and application scenarios for our logic.

2.4 Examples

Example 1 (Real Estate)**.** *We now exploit* PCL *to model a typical preliminary contract for a real estate sale in Italy.*

Assume a buyer who is interested in buying a new house from a given seller. Before stipulating the actual purchase contract, the buyer and the seller meet to stipulate a preliminary sale contract, that fixes the terms and conditions of the purchase. Typically, this contract will indicate the price and the date when the deed of sale will take place, and it will outline the obligations for the buyer and the seller. When the preliminary contract is signed by both parties, the buyer will pay a part of the sale price. By the Italian laws, if the seller decides not to sell the house after having signed the preliminary contract and collected the deposit, she must pay the buyer back twice the sum received. Similarly, if the buyer changes his mind and decides not to buy the house, he loses the whole deposited amount. We model the preliminary sale contract as two PCL *formulae, one for the buyer and the other for the seller. The buyer will sign the preliminary contract (*signB*), provided that the seller will actually sell her house (*sellS*), or she refunds twice the sum received (*$\mathsf{refundS}$*). Also, the buyer promises that if he signs the preliminary contract, than either he will pay the stipulated price (*payB*), or he will not pay and lose the deposit (*$\mathsf{refundB}$*).*

$$Buyer \;=((\mathsf{sellS} \lor \mathsf{refundS}) \twoheadrightarrow \mathsf{signB}) \land (\mathsf{signB} \twoheadrightarrow (\mathsf{payB} \lor (\neg\mathsf{payB} \land \mathsf{refundB})))$$

The seller promises to sign the preliminary contract (signS*), provided that either the buyer promises to pay the stipulated amount, or he promises to lose the deposit. Also, the seller promises that if she signs the preliminary contract, then she will either sell her house, or will not sell and refund twice the sum received.*

$$Seller \;=((\mathsf{payB} \lor \mathsf{refundB}) \twoheadrightarrow \mathsf{signS}) \land (\mathsf{signS} \twoheadrightarrow (\mathsf{sellS} \lor (\neg\mathsf{sellS} \land \mathsf{refundS})))$$

A first consequence is that the two contracts lead to an agreement between the buyer and the seller, that is both parties will sign the preliminary contract:

$$\vdash Buyer \land Seller \rightarrow \mathsf{signB} \land \mathsf{signS} \tag{12}$$

As a second consequence, if one of the parties does not finalize the final deed of sale, than that party will refund the other:

$$\vdash Buyer \land Seller \land \neg\mathsf{payB} \rightarrow \mathsf{refundB} \tag{13}$$

$$\vdash Buyer \land Seller \land \neg\mathsf{sellS} \rightarrow \mathsf{refundS} \tag{14}$$

Example 2 (Online sale)**.** *We now describe a possible online sale between two parties. In order to buy an item, the buyer has to contact first the bank, to reserve from his account a specific amount of money for the transaction. When this happens, that amount is no longer available for anything else. We model this reservation with the formula* lock*. Then, the buyer has to make an offer to the seller: this is modelled with* offer*. The seller, when provided with an offer, evaluates it. If she thinks the offer is good, and the money has been reserved, then she will send the item (*send*). Otherwise, she cancels the transaction (*abort*). When the transaction is aborted, the bank cancels the money reservation, so that the buyer can use the amount for other transactions (*unlock*).*

We now formalize this scenario. The buyer agrees to $\mathsf{lock} \land \mathsf{offer}$*, provided that either the item is sent, or the money reservation is cancelled. The seller agrees to evaluate the offer. The bank agrees to cancel the reservation when the transaction is aborted.*

$$\begin{aligned} \mathit{Buyer} &= (\mathsf{send} \lor \mathsf{unlock}) \twoheadrightarrow (\mathsf{lock} \land \mathsf{offer}) \\ \mathit{Seller} &= \mathsf{offer} \twoheadrightarrow ((\mathsf{lock} \to \mathsf{send}) \lor \mathsf{abort}) \\ \mathit{Bank} &= (\mathsf{lock} \land \mathsf{abort}) \twoheadrightarrow \mathsf{unlock} \end{aligned}$$

Under these assumptions, we can see that either the item is sent, or the transaction is aborted and the reservation cancelled.

$$\vdash (\mathit{Buyer} \land \mathit{Seller} \land \mathit{Bank}) \to (\mathsf{send} \lor (\mathsf{abort} \land \mathsf{unlock}))$$

Example 3 (Dining retailers)**.** *Around a table, a group of n cutlery retailers is about to have dinner. At the center of the table, there is a large dish of food. Despite the food being delicious, the retailers cannot start eating right now. To do that, and follow the proper etiquette, each retailer needs to have a complete cutlery set, consisting of n pieces, each of a different kind. Each one of the n retailers owns a distinct set of n piece of cutlery, all of the same kind. The retailers start discussing about trading their cutlery, so that they can finally eat.*

We formalize this scenario as follows. We number the retailers $r_1, \dots, r_n$ together with the kinds of pieces of cutlery, so that r_i initially owns n pieces of kind number i. We then write $\mathsf{g}_{i,j}$ for "r_i gives *a piece (of kind i) to r_j". Since retailers can use their own cutlery, we assume $\mathsf{g}_{i,i}$ to be true. Retailer r_i can start eating whenever $e_i = \bigwedge_j \mathsf{g}_{j,i}$. Instead, he provides the cutlery to others whenever $p_i = \bigwedge_j \mathsf{g}_{i,j}$.*

Suppose that r_1 commits to a simple exchange with r_2: they commit to $\mathsf{g}_{2,1} \twoheadrightarrow \mathsf{g}_{1,2}$ and $\mathsf{g}_{1,2} \twoheadrightarrow \mathsf{g}_{2,1}$, and the exchange takes place since $\mathsf{g}_{2,1} \land \mathsf{g}_{1,2}$ can be derived. While this seems a fair deal, it actually exposes r_1 to a risk: if $r_3, \dots, r_n$ perform a similar exchange with r_2, then we have $\mathsf{g}_{2,i} \land \mathsf{g}_{i,2}$ for all i. In particular, $\mathsf{g}_{i,2}$ holds for all i, so r_2 can start eating. This is however not necessarily the case for r_1, since r_3 has not committed to any exchange with r_1.

A wise retailer would then never agree to a simple exchange $\mathsf{g}_{2,1} \twoheadrightarrow \mathsf{g}_{1,2}$. Instead, the retailer r_1 could commit to a safer contract:

$$e_1 \twoheadrightarrow p_1 = \mathsf{g}_{1,1} \land \mathsf{g}_{2,1} \land \cdots \land \mathsf{g}_{n,1} \twoheadrightarrow \mathsf{g}_{1,1} \land \mathsf{g}_{1,2} \land \cdots \land \mathsf{g}_{1,n}$$

The idea is simple: r_1 requires each piece of cutlery, that is, r_1 requires to be able to start eating (e_1). When this happens, r_1 agrees to provide each other retailer with a piece of his cutlery (p_1). Now, assume each retailer r_i commits to the analogous contract. We then have the desired agreement (proof in [4]).

$$\vdash \bigwedge_i (e_i \twoheadrightarrow p_i) \to \bigwedge_i e_i$$

3 The Calculus

We now define our calculus of contracting processes. It is a variant of Concurrent Constraints Programming [35], featuring primitives for multi-party synchronization via contracts.

3.1 Syntax

We use a denumerable set of *names*, ranged over by $n, m, \ldots$, and a denumerable set of *variables*, ranged over by $x, y, \ldots$. Metavariables a, b range over both names and variables. Intuitively, a name plays the same role as in the π-calculus, while variables roughly behave as names in the fusion calculus [30]. That is, distinct names represent distinct concrete objects, each one with its own identity. Instead, distinct variables can be fused together to some name. A *fusion* σ is a substitution that maps a set of variables to a single name. We write $\sigma = \{n/\vec{x}\}$ for the fusion that replaces each variable in $\vec{x}$ with the name n. Unlike [30], our calculus can fuse a variable only once.

Constraints are represented as PCL formulae, extended so to allow parameters within primes. Note that such extension does not introduce quantifiers, so leabing PCL a propositional logic: indeed, the prime formula $\mathsf{p}(a)$ is still atomic from the point of view of the logic. We let letters c, d range over arbitrary PCL formulae, while letters u, v range over $\{\bot, \neg\}$-free formulae.

The syntax of our calculus follows.

Definition 5 (Processes)**.** *Prefixes and processes are defined as follows:*

$$\begin{array}{ll}\pi ::= \tau \;\big|\; \mathsf{ask}\, c \;\big|\; \mathsf{tell}\, u \;\big|\; \mathsf{check}\, c \;\big|\; \mathsf{fuse}_x\, c \;\big|\; \mathsf{join}_x\, c & \textit{(prefixes)} \\ P ::= u \;\big|\; \sum_{i \in I} \pi_i.P_i \;\big|\; P|P \;\big|\; (a)P \;\big|\; X(\vec{a}) & \textit{(processes)}\end{array}$$

Prefixes π include τ (the standard silent operation as in CCS), as well as tell, ask, and check as in Concurrent Constraints [35]. The prefix $\mathsf{tell}\, u$ augments the context with the formula u. The prefix $\mathsf{check}\, c$ checks if c is consistent with the context. The prefix $\mathsf{ask}\, c$ causes a process to stop until the constraint c is deducible from the context. Note that, since we only allow negation-free formulae u here, the context will always be consistent, by Lemma 2. The prefixes $\mathsf{fuse}_x\, c$ and $\mathsf{join}_x\, c$ drive the fusion of the variable x. The prefix $\mathsf{join}_x\, c$ instantiates x to *any known name*, provided that after the instantiation the constraint c is satisfied. The prefix $\mathsf{fuse}_x\, c$ fuses x with any other set of known variables, provided that, when all the fused variables are instantiated to a fresh name, the constraint c is satisfied. To avoid unnecessary fusion, the set of variables is required to be minimal (see Def. 8). To grasp the intuition behind the two kinds of fusions, think of names as session identifiers. Then, a $\mathsf{fuse}_x\, c$ initiates a new session, while a $\mathsf{join}_x\, c$ joins an already initiated session.

Processes P include the active constraint u, the summation $\sum_i \pi_i.P_i$ of guarded processes, the parallel composition $P|P$, the scope delimitation $(a)P$, and the instantiated constant $X(\vec{a})$, where $\vec{a}$ is a tuple of names/variables. When a constraint c is at the top-level of a process, we say it is *active*. We use a set of defining equations $\{X_i(\vec{x}) \doteq P_i\}_i$ with the provision that each occurrence of X_j in P_k is guarded, i.e. it is behind some prefix. We shall often use $C = \{c_1, c_2, \ldots\}$ as a process, standing for $c_1|c_2|\cdots$. We write 0 for the empty sum. Singleton sums are simply written $\pi.P$. We use $+$ to merge sums:

$$\sum_{i \in I} \pi_i.P_i + \sum_{i \in J} \pi_i.P_i \;=\; \sum_{i \in I \cup J} \pi_i.P_i \qquad \text{if } I \cap J = \emptyset$$

We stipulate that $+$ binds more tightly than $|$.

Free variables and names of processes are defined as usual: they are free whenever they occur in a process not under a delimitation. Alpha conversion and substitutions are defined accordingly. As a special case, we let:

$$(\mathsf{fuse}_x\, c)\{n/x\} = (\mathsf{join}_x\, c)\{n/x\} = \mathsf{ask}\, c\{n/x\}$$

That is, when a variable x is instantiated to a name, the prefixes $\mathsf{fuse}_x\, c$ and $\mathsf{join}_x\, c$ can no longer require the fusion of x, so they behave as a plain $\mathsf{ask}\, c$. Henceforth, we consider processes up-to alpha-conversion.

3.2 Semantics

We provide our calculus with both a reduction semantics and a labelled transition semantics. As usual for CCP, the former explains how a process evolves within the *whole* context (so, it is not compositional), while the latter also explains how a process interacts with the environment.

3.2.1 Transition semantics

We now define a labelled transition semantics of processes. The labelled relation $\xrightarrow{\alpha}$ is *compositional*: all the prefixes can be fired by considering the relevant portion of the system at hand. The only exception is the $\mathsf{check}\, c$ prefix, which is inherently non-compositional. We deal with $\mathsf{check}\, c$ by layering the reduction relation $\rightarrowtail$ over the relation $\xrightarrow{\alpha}$.

We start by introducing in Def. 6 the *actions* of our semantics, that is the set of admissible labels of the LTS. The transition relation is presented in Def. 7.

Definition 6. *Actions α are as follows, where C denotes a set of constraints.*

$$\alpha \;::=\; \tau \;\big|\; C \;\big|\; C \vdash c \;\big|\; C \vdash^F_x c \;\big|\; C \vdash^J_x c \;\big|\; C \nvdash \bot \;\big|\; (a)\,\alpha \qquad \textit{(actions)}$$

The action τ represents an internal move. The action C advertises of a set of active constraints. The action $C \vdash c$ is a *tentative* action, generated by a process attempting to fire an $\mathsf{ask}\, c$ prefix. The set C collects the active constraints discovered so far. Similarly for $C \vdash^F_x c$ and $\mathsf{fuse}_x\, c$, for $C \vdash^J_x c$ and $\mathsf{join}_x\, c$, as well as for $C \nvdash \bot$ and $\mathsf{check}\, c$. In the last case, C also includes c. The delimitation in $(a)\alpha$ is for scope extrusion, as in the labelled semantics of the π-calculus [34]. We write $(\vec{a})\alpha$ to denote a set of distinct delimitations, neglecting their order, e.g. $(ab) = (ba)$. We simply write $(\vec{a}\vec{b})$ for $(\vec{a} \cup \vec{b})$.

Definition 7 (Transition relation)**.** *The transition relations $\xrightarrow{\alpha}$ are the smallest relations between processes satisfying the rules in Fig. 1. The last two rules in Fig. 1 define the reduction relation $\rightarrowtail$.*

Many rules in Fig. 1 are rather standard, so we comment on the most peculiar ones, only. The overall idea is the following: a tentative action label carries all the proof obligations needed to fire the corresponding prefix. The Par* rules allow for exploring the context, and augment the label with the observed constraints. The Close* rules check that enough constraints have been collected so that the proof obligations can be discharged, and transform the label into a τ. The Top* rules act on the top-level, only, and define the semantics of $\mathsf{check}\, c$.

$$\tau.P \xrightarrow{\tau} P \quad [\textsc{Tau}] \qquad \mathsf{ask}\,c.P \xrightarrow{\emptyset\vdash c} P \quad [\textsc{Ask}] \qquad \mathsf{tell}\,u.P \xrightarrow{\tau} u|P \quad [\textsc{Tell}]$$

$$\mathsf{check}\,c.P \xrightarrow{\{c\}\nvdash\bot} P \quad [\textsc{Check}] \quad \mathsf{fuse}_x\,c.P \xrightarrow{\emptyset\vdash^F_x c} P \quad [\textsc{Fuse}] \quad \mathsf{join}_x\,c.P \xrightarrow{\emptyset\vdash^J_x c} P \quad [\textsc{Join}]$$

$$u \xrightarrow{\{u\}} u \quad [\textsc{Constr}] \qquad \textstyle\sum_i \pi_i.P_i \xrightarrow{\emptyset} \sum_i \pi_i.P_i \quad [\textsc{IdleSum}]$$

$$\frac{P \xrightarrow{(\vec a)C} P' \quad Q \xrightarrow{(\vec b)C'} Q'}{P|Q \xrightarrow{(\vec a\vec b)(C\cup C')} P'|Q'} \ [\textsc{ParConstr}] \qquad \frac{P \xrightarrow{(\vec a)C} P' \quad Q \xrightarrow{(\vec b)(C'\vdash c)} Q'}{P|Q \xrightarrow{(\vec a\vec b)(C\cup C'\vdash c)} P'|Q'} \ [\textsc{ParAsk}]$$

$$\frac{P \xrightarrow{(\vec a)C} P' \quad Q \xrightarrow{(\vec b)(C'\vdash^F_x c)} Q'}{P|Q \xrightarrow{(\vec a\vec b)(C\cup C'\vdash^F_x c)} P'|Q'} \ [\textsc{ParFuse}] \qquad \frac{P \xrightarrow{(\vec a)C} P' \quad Q \xrightarrow{(\vec b)(C'\vdash^J_x c)} Q'}{P|Q \xrightarrow{(\vec a\vec b)(C\cup C'\vdash^J_x c)} P'|Q'} \ [\textsc{ParJoin}]$$

$$\frac{P \xrightarrow{(\vec a)C} P' \quad Q \xrightarrow{(\vec b)(C'\nvdash\bot)} Q'}{P|Q \xrightarrow{(\vec a\vec b)(C\cup C'\nvdash\bot)} P'|Q'} \ [\textsc{ParCheck}] \qquad \frac{P \xrightarrow{\tau} P'}{P|Q' \xrightarrow{\tau} P'|Q'} \ [\textsc{ParTau}]$$

$$\frac{\pi_j.P_j \xrightarrow{\alpha} P'}{\sum_i \pi_i.P_i \xrightarrow{\alpha} P'} \ [\textsc{Sum}] \qquad \frac{P\{\vec a/\vec x\} \xrightarrow{\alpha} P'}{X(\vec a) \xrightarrow{\alpha} P'} \ \text{if } X(\vec x) \doteq P \ [\textsc{Def}]$$

$$\frac{P \xrightarrow{\alpha} P'}{(a)P \xrightarrow{\alpha} (a)P'} \ \text{if } a \notin \alpha \ [\textsc{Del}] \qquad \frac{P \xrightarrow{\alpha} P'}{(a)P \xrightarrow{(a)\alpha} P'} \ [\textsc{Open}]$$

$$\frac{P \xrightarrow{(\vec a)(C\vdash c)} P' \quad C \vdash c}{P \xrightarrow{\tau} (\vec a)P'} \ [\textsc{CloseAsk}]$$

$$\frac{P \xrightarrow{(xn\vec a)(C\vdash^J_x c)} P' \quad C\sigma \vdash c\sigma \quad \sigma = \{n/x\}}{P \xrightarrow{\tau} (n\vec a)\big(P'\sigma\big)} \ [\textsc{CloseJoin}]$$

$$\frac{P \xrightarrow{(x\vec y\vec a)(C\vdash^F_x c)} P' \quad \sigma = \{n/x\vec y\} \quad n \text{ fresh} \quad C \vdash^{loc}_\sigma c}{P \xrightarrow{\tau} (n\vec a)(P'\sigma)} \ [\textsc{CloseFuse}]$$

$$\frac{P \xrightarrow{\tau} P'}{P \rightarrowtail P'} \ [\textsc{TopTau}] \qquad \frac{P \xrightarrow{(\vec a)(C\nvdash\bot)} P'}{P \rightarrowtail (\vec a)P' \quad C \nvdash \bot} \ [\textsc{TopCheck}]$$

Figure 1: The transition system for our calculus. Symmetric rules for $+, |$ are omitted. The rules $\textsc{Par}*$ have the following no-capture side condition: $\vec a$ is fresh in $\vec b, C', c, x, Q'$, while $\vec b$ is fresh in C, P'.

The rules for prefixes simply generate the corresponding tentative actions. The rule Tell adds a constraint to the environment, thus making it *active*. Active constraints can then signal their presence through the Constr rule: each constraint generates its own singleton. A sum of guarded processes $\sum \pi.P$

can instead signal that it is *not* a constraint, by generating the empty set of constraints through IdleSum.

An advertised constraint is used to augment the tentative actions through the Par* rules. The rule ParConstr merges the sets of constraints advertised by two parallel processes. The rule ParAsk allows for augmenting the constraints C in the tentative action $C \vdash c$ generated by an Ask, by also accounting for the set of constraints advertised by the parallel process. Similarly for the rules ParFuse, ParJoin, ParCheck. The rule ParTau simply propagates τ actions of parallel processes. The Par* rules also merge the set of delimitations; variable and name captures are avoided through a side condition.

The rules Del and Open handle delimitation. As usual, when a is not mentioned in an action, we can propagate the action across a delimitation (a) by using Del. The rule Open instead allows for scope extrusion. Note that Open has no side conditions: all the checks needed for scope extrusion are already handled by the Par* rules.

The Close* rules are the crucial ones, since they provide the mechanism to finalize the actions generated by ask, join, and fuse. The rule CloseAsk is the simplest: if the collected constraints C entail c, the ask prefix can indeed fire. In that case, a silent action τ is generated, and the delimitations $(\vec{a})$ can be brought back to the process level, stopping the scope extrusion. The rule CloseJoin is similar to CloseAsk, except it also instantiates the variable x to a name n. To this purpose, we apply a substitution $\sigma = \{^n/_x\}$ to the constraints C, c before testing for entailment. If that holds, we apply σ to the residual process P' as well. Of course, we need to ensure that all the occurrences of x are substituted: this is done by requiring x to be present in the delimitations of the action in the premise, hence requiring the whole scope of x is at hand. Further, we constrain n in the same fashion, so that x can only be instantiated to an existing name. The rule CloseFuse acts similarly, yet it cannot instantiate x to an existing name; rather, the name n here is a fresh one, introduced in the process through a delimitation. Note that CloseFuse may fuse several variables together, since n substitutes for the whole set $x\vec{y}$. The set $\vec{y}$ of the variables to be fused with x is chosen according to the *local minimal fusion* relation $C \vdash^{loc}_{\sigma} c$, to be defined in a while (Def. 8).

Summing up, the LTS first generates tentative actions, and then converts them to τ when enough constraints are discovered (rules Close*). Then, the τ action can be propagated towards the top level (rule ParTau). A prefix check c cannot be handled in the same fashion, since it requires to check the consistency of c with respect to *all* the active constraints. To this aim, we use the reduction relation $\rightarrowtail$, layered over the $\xrightarrow{\alpha}$ relation. The reduction $\rightarrowtail$ only includes internal moves $\xrightarrow{\tau}$ (rule TopTau) and successful check moves (rule TopCheck). This effectively discards tentative actions, filtering for the successful ones, only. Note that $\rightarrowtail$ is only applied to the top level.

Definition 8 (Local Minimal Fusion)**.** *A fusion $\sigma = \{n/\vec{z}\}$ is* local minimal *for C, c, written $C \vdash^{loc}_{\sigma} c$, iff:*

$$\exists C' \subseteq C \ : \ \big(\, C'\sigma \vdash c\sigma \ \ \wedge \ \ \nexists \vec{w} \subsetneq \vec{z} \ : \ C'\{^n/_{\vec{w}}\} \vdash c\{^n/_{\vec{w}}\} \,\big)$$

We now briefly discuss the motivations behind this definition. First, we consider a subset C' of C (*locality restriction*). Then, we require that C' entails c

when the variables $\vec{z}$ are fused. The fusion must be *minimal*, that is fusing a proper subset of variables must not cause the entailment.

The rationale for minimality is that we want to fuse those variables only, which are actually involved in the entailment of c – not any arbitrary superset. Pragmatically, we will often use $\mathsf{fuse}_x\, c$ as a construct to establish *sessions*: the participants are then chosen among those actually involved in the satisfaction of the constraint c, and each participant "receives" the fresh name n through the application of σ. In this case, n would act as a sort of *session identifier*.

To understand the motivations underlying the locality restriction, note that a set of variables $\vec{z}$ may be minimal w.r.t. a set of constraints C', yet not minimal for a superset $C \supset C'$, as the following example shows. Let:

$$c = \mathsf{p}(x) \quad C' = \{\mathsf{q}(y)\,,\, \mathsf{q}(z) \vee \mathsf{s} \to \mathsf{p}(y)\} \quad C = C' \cup \{\mathsf{s}\}$$

To obtain $C' \vdash c$, all the variables x, y, z must be fused. It is immediate to see that this fusion is minimal: to produce $\mathsf{p}(x)$ we must exploit the implication, so fusing x with y is mandatory. Further, the hypothesis $\mathsf{q}(z) \vee \mathsf{s}$ can only be discharged through $\mathsf{q}(y)$, and this forces the fusion of y and z. Instead, to obtain $C \vdash c$, we can simply fuse x with y; (and neglect z), because the hypothesis $\mathsf{q}(z) \vee \mathsf{s}$ can now be discharged through s. So, in this case the set of variables x, y, z is *not* minimal. This phenomenon could, in principle, lead to unexpected behaviour. Consider, for instance:

$$\begin{aligned} P &= (x)(y)(z)(\mathsf{fuse}_x\, c.P \mid C') \mid \mathsf{s} \\ Q &= (x)(y)(z)(\mathsf{fuse}_x\, c.P \mid C' \mid \mathsf{s}) \end{aligned}$$

where, with some abuse of notation, C' stands for the parallel composition of its constraints. In P, when dealing with the process $(x)(y)(z)(\mathsf{fuse}_x\, c.P \mid C')$, we can apply CloseFuse to fuse the variables x, y, z, because only C' is known at this time, and the set of variables is minimal for C'. Instead, in Q the application of CloseFuse must be deferred until a delimitation (x) is found; at that time the whole set of constraints C is known, so making the fusion x, y, z *not* minimal for C. Of course, this phenomenon clashes with our intuition that P and Q should be equivalent, since Q is obtained from P through a scope extrusion.

The key issue here is that minimality is intrinsically non-compositional: discovering new constraints can break minimality. To recover compositionality, our definition of $C \vdash^{loc}_{\{n/\vec{z}\}} c$ does not require $\vec{z}$ to be minimal for C, but to be such for any *subset* C' of C. This allows, for instance, to have both $C \vdash^{loc}_{\{n/xyz\}} c$ and $C \vdash^{loc}_{\{n/xy\}} c$ in the example above. The intuition behind this is that it is not necessary to (globally) explore the whole system to decide if a set of contracts leads to an agreement – inspecting any set of (locally) known contracts is enough. To a local observer, a set x, y, z of variables to fuse may appear minimal. To a more informed observer, the same set may appear to be non-minimal (the minimal one being x, y). Both choices for fusion are allowed by our semantics.

Our semantics does not make use of any structural equivalence relation. However, such a relation can be defined as the minimum equivalence satisfying the axioms of Fig. 2. This relation turns out to be a bisimulation.

Theorem 7. *The relation $\equiv$ is a bisimulation, i.e.*

$$P \equiv Q \xrightarrow{\alpha} Q' \implies \exists P'.\, P \xrightarrow{\alpha} P' \equiv Q'$$

$$P|0 \equiv P \quad P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)|R$$

$$(a)(P|Q) \equiv P|(a)Q \quad \text{if } a \notin \mathit{free}(P) \qquad (a)(b)P \equiv (b)(a)Q$$

Figure 2: Structural equivalence.

$$\frac{}{(\vec{a})\,(\tau.P + Q \mid R) \rightarrowtail (\vec{a})\,(P \mid R)}\ [\textsc{Tau}]$$

$$\frac{}{(\vec{a})\,(\mathsf{tell}\, u.P + Q \mid R) \rightarrowtail (\vec{a})(u \mid P \mid R)}\ [\textsc{Tell}]$$

$$\frac{C \vdash c}{(\vec{a})\,(C \mid \mathsf{ask}\, c.P + Q \mid R) \rightarrowtail (\vec{a})(C \mid P \mid R)}\ [\textsc{Ask}]$$

$$\frac{C, c \nvdash \bot \quad R \text{ free from active constraints}}{(\vec{a})\,(C \mid \mathsf{check}\, c.P + Q \mid R) \rightarrowtail (\vec{a})\,(P \mid R)}\ [\textsc{Check}]$$

$$\frac{\sigma = \{n/x\vec{y}\} \quad n \text{ fresh} \quad C \vdash_{\sigma}^{min} c}{(x\vec{y}\vec{a})\,(C \mid \mathsf{fuse}_x\, c.P + Q \mid R) \rightarrowtail (n\vec{a})\,(C \mid P \mid R)\,\sigma}\ [\textsc{Fuse}]$$

$$\frac{C\{{}^{n}\!/\!{}_{x}\} \vdash c\{{}^{n}\!/\!{}_{x}\}}{(xn\vec{a})\,(C \mid \mathsf{join}_x\, c.P + Q \mid R) \rightarrowtail (n\vec{a})\,(C \mid P \mid R)\{{}^{n}\!/\!{}_{x}\}}\ [\textsc{Join}]$$

$$\frac{P \ \equiv\ P' \rightarrowtail Q' \ \equiv\ Q}{P \rightarrowtail Q}\ [\textsc{Struct}]$$

Figure 3: The reduction relation

3.2.2 Reduction Semantics

Our calculus also admits a reduction semantics, which agrees with the $\rightarrowtail$ relation we introduced above. In order to define it, we first augment the structural equivalence relation $\equiv$ of Fig. 2 so to include (possibly recursive) process equations $X(\vec{x}) \doteq P$. Also, we (re-)define the relation $\rightarrowtail$ by working up-to $\equiv$-equivalent processes.

Definition 9 (Reduction)**.** *Reduction $\rightarrowtail$ is the smallest relation between processes satisfying the rules in Fig. 3.*

We now comment the rules for reduction. Rule Tau simply fires the τ prefix. Rule Tell augments the context (R) with a constraint u. Rule Ask checks whether the context has enough active constraints C so to entail c. Rule Check checks the context for consistency with c. Since this requires inspecting every active constraint in the context, a side condition precisely separates the context between C and R, so that all the active constraints are in C, which in this case acts as a global *constraint store*.

Rule Fuse replaces a set of variables $x\vec{y}$, with a fresh name n, hence fusing all the variables together. One variable in the set, x, is the one mentioned in the

$\mathsf{fuse}_x\, c$ prefix, while the others, $\vec{y}$, are taken from the context. The replacement of variables is performed by the substitution σ in the rule premises. We require that σ is a *minimal fusion* for $C \vdash c$, as formally defined below.

Definition 10 (Minimal Fusion)**.** *A fusion $\sigma = \{n/\vec{z}\}$ is* minimal *for $C \vdash c$, written $C \vdash^{min}_{\sigma} c$, iff:*

$$C\sigma \vdash c\sigma \quad \wedge \quad \nexists \vec{w} \subsetneq \vec{z} \ : \ C\{^n/_{\vec{w}}\} \vdash c\{^n/_{\vec{w}}\}$$

A minimal fusion σ must cause the entailment of c by the context C. Furthermore, fusing a proper subset of variables must not cause the entailment. The rationale for minimality is exactly the one we discussed for local minimal fusion in Def. 8. Here, the locality restriction is already implicit in rule Fuse, since the context R could contain active constraints as well.

Rule Join replaces a variable x with a name n taken from the context. Note that, unlike Fuse, n is *not* fresh here. To enable a $\mathsf{join}_x\, c$ prefix, the substitution must cause c to be entailed by the context C. Intuitively, this prefix allows to "search" in the context for some x satisfying a constraint c.

Rule Struct simply allows to consider processes up-to structural equivalence.

The reduction relation coincides with the one defined through our labelled semantics, when processes are considered up-to structural equivalence.

Theorem 8. *Let $\rightarrowtail_r$ be the relation defined in Def. 9 and let $\rightarrowtail_l$ be the relation defined in Def. 7. Then:*

$$P \rightarrowtail_r P' \iff \exists Q, Q'.\ P \equiv Q \rightarrowtail_l Q' \equiv P'$$

Proof. (Sketch) The ($\Rightarrow$) implication follows by rule induction on the definition of $\rightarrowtail_r$. Indeed, most rules are axioms, and are easily checked. The only exception is Struct, which is handled by the inductive hypothesis and by choosing Q, Q' accordingly.

The ($\Leftarrow$) implication requires more care. First, because of rule Struct, we can assume $P = Q$ and $P' = Q'$ without loss of generality. Then, we check the TopTau and TopCheck cases independently. This is done by rule induction on the compositional relation $\xrightarrow{\alpha}$, by proving the following invariants: (below, R contains no active constraints)

$$\begin{array}{lll}
P \xrightarrow{\tau} P' & \Longrightarrow & P \rightarrowtail_r P' \\
P \xrightarrow{(\vec{a})(C \vdash c)} P' & \Longrightarrow & \exists \vec{b}, Q, R, S. \begin{cases} P \equiv (\vec{a}\vec{b})(C|R|Q + \mathsf{ask}\, c.S) \wedge \\ P' \equiv (\vec{b})(C|R|S) \end{cases} \\
P \xrightarrow{(\vec{a})(C \vdash^J_x c)} P' & \Longrightarrow & \exists \vec{b}, Q, R, S. \begin{cases} P \equiv (\vec{a}\vec{b})(C|R|Q + \mathsf{join}_x\, c.S) \wedge \\ P' \equiv (\vec{b})(C|R|S) \end{cases} \\
P \xrightarrow{(\vec{a})(C \vdash^F_x c)} P' & \Longrightarrow & \exists \vec{b}, Q, R, S. \begin{cases} P \equiv (\vec{a}\vec{b})(C|R|Q + \mathsf{fuse}_x\, c.S) \wedge \\ P' \equiv (\vec{b})(C|R|S) \end{cases} \\
P \xrightarrow{(\vec{a})(C \nvdash \bot)} P' & \Longrightarrow & \exists \vec{b}, Q, R, S. \begin{cases} P \equiv (\vec{a}\vec{b})(C|R|Q + \mathsf{check}\, c.S) \wedge \\ P' \equiv (\vec{b})(C|R|S) \end{cases} \\
P \xrightarrow{(\vec{a})C} P' & \Longrightarrow & \exists \vec{b}, R.\ P \equiv (\vec{a}\vec{b})(C|R) \wedge P' \equiv (\vec{b})(C|R)
\end{array}$$

$\square$

3.3 Examples

We illustrate our calculus through some examples.

Example 4 (Handshaking)**.** *Recall the basic handshaking in Sect. 1.1:*

$$\begin{aligned} \mathit{Alice} &= (x)\ \big(\mathsf{tell}\,\mathsf{b}(x) \twoheadrightarrow \mathsf{a}(x).\ \mathsf{fuse}_x\,\mathsf{a}(x).\ \mathit{lendAirplane}\big) \\ \mathit{Bob} &= (y)\ \big(\mathsf{tell}\,\mathsf{a}(y) \twoheadrightarrow \mathsf{b}(y).\ \mathsf{fuse}_y\,\mathsf{b}(y).\ \mathit{lendBike}\big) \end{aligned}$$

A possible trace is the following:

$$\begin{aligned} &\mathit{Alice} \mid \mathit{Bob} \\ \xrightarrow{\tau}\ &(x)\ \big(\mathsf{b}(x) \twoheadrightarrow \mathsf{a}(x) \mid \mathsf{fuse}_x\,\mathsf{a}(x).\ \mathit{lendAirplane}\big) \mid \mathit{Bob} \\ \xrightarrow{\tau}\ &(x)\ \big(\mathsf{b}(x) \twoheadrightarrow \mathsf{a}(x) \mid \mathsf{fuse}_x\,\mathsf{a}(x).\ \mathit{lendAirplane}\big) \mid \\ &(y)\ \big(\mathsf{a}(y) \twoheadrightarrow \mathsf{b}(y) \mid \mathsf{fuse}_y\,\mathsf{b}(y).\ \mathit{lendBike}\big) \\ \xrightarrow{\tau}\ &(n)\ \big(\mathsf{b}(n) \twoheadrightarrow \mathsf{a}(n) \mid \mathit{lendAirplane}\{^n/_x\} \mid \\ &\qquad \mathsf{a}(n) \twoheadrightarrow \mathsf{b}(n) \mid \mathsf{ask}\,\mathsf{b}(n).\ \mathit{lendBike}\{^n/_y\}\big) \\ \xrightarrow{\tau}\ &(n)\ \big(\mathsf{b}(n) \twoheadrightarrow \mathsf{a}(n) \mid \mathit{lendAirplane}\{^n/_x\} \mid \\ &\qquad \mathsf{a}(n) \twoheadrightarrow \mathsf{b}(n) \mid \mathit{lendBike}\{^n/_y\}\big) \end{aligned}$$

In step one, we use Tell,ParTau,Del *to fire the prefix* $\mathsf{tell}\,\mathsf{b}(x) \twoheadrightarrow \mathsf{a}(x)$*. Similarly, in step two we fire the prefix* $\mathsf{tell}\,\mathsf{a}(y) \twoheadrightarrow \mathsf{b}(y)$ *through the very same rules. Step three is the crucial one. The prefix* $\mathsf{fuse}_x\,\mathsf{a}(x)$ *is fired through rule* Fuse*. Through rules* Constr,ParFuse*, we discover the active constraint* $c_a = \mathsf{b}(x) \twoheadrightarrow \mathsf{a}(x)$*. We use rule* Open *to obtain the action* $(x)\{c_a\} \vdash^F \mathsf{a}(x)$ *for the Alice part. For the Bob part, we use rule* Constr *to discover* $c_b = \mathsf{a}(y) \twoheadrightarrow \mathsf{b}(y)$*, which we then merge with the empty set of constraints obtained through rule* Idle*; we finally apply* Open *and obtain* $(y)\{c_b\}$*. At the top level, we can then apply* ParFuse *to deduce* $(x,y)\{c_a,c_b\} \vdash^F \mathsf{a}(x)$*. Finally, rule* CloseFuse *here can be applied, fusing* x *and* y *by instantiating them to the fresh name* n*. It is easy to check that* $\{c_a,c_b\} \vdash^{loc}_{\{n/x,y\}} \mathsf{a}(x)$*. Note that the fusion* $\{n/x,y\}$ *transforms* $\mathsf{fuse}_y\,\mathsf{b}(y)$ *into* $\mathsf{ask}\,\mathsf{b}(n)$*, which is then fired in the last step.*

Example 5 (Unfair handshaking)**.** *To better understand the role of contractual implication, consider the following processes:*

$$\begin{aligned} \mathit{Alice}' &= (x)\ \big(\mathsf{tell}\,\mathsf{a}(x).\ \mathsf{fuse}_x\,\mathsf{b}(x).\ \mathit{lendAirplane}\big) \\ \mathit{Bob}' &= (y)\ \big(\mathsf{tell}\,\mathsf{b}(y).\ \mathsf{fuse}_y\,\mathsf{a}(y).\ \mathit{lendBike}\big) \end{aligned}$$

Note that, differently from Ex. 4, the contracts of Alice' *and* Bob' *do not use* $\twoheadrightarrow$*. It is straightforward to check that* $P' = \mathit{Alice}' \mid \mathit{Bob}'$ *behaves similarly to* $P = \mathit{Alice} \mid \mathit{Bob}$*, since the handshaking is still performed. However, the processes* P *and* P' *behave quite differently in the presence of a third kid, e.g. let:*

$$\mathit{Carl} = (z)\ \mathsf{fuse}_z\,\mathsf{a}(z).\,0$$

The system $P' \mid \mathit{Carl}$ *allows Carl to fire his own prefix, by fusing* x *with* z*. So, Carl will be allowed to play with Alice's airplane, even though Alice receives no promises from Carl. Indeed, after the fusion, Alice will be stuck on an* $\mathsf{ask}\,\mathsf{b}(n)$*.*

Technically, this happens because $\{\mathsf{a}(x)\} \vdash^{loc}_{\{n/x,z\}} \mathsf{a}(z)$ *holds. In this case, Bob will be stuck as well: he cannot play with Alice's airplane, since Carl took it. By contrast, the system* $P \mid \mathit{Carl}$ *does* not *allow Carl to play with the airplane. Indeed, Alice specified in her contract that she will lend her airplane only if a bike is promised in return. Since Carl promises no bike, he cannot use the airplane. Instead, Bob can successfully exchange his bike with Alice's airplane.*

Example 6 (Dining retailers)**.** *We reuse the formulae of Ex. 3, augmenting each prime formula with a parameter* x*.*

$$e_i(x) = \wedge_j \mathsf{g}_{j,i}(x) \qquad\qquad p_i(x) = \wedge_j \mathsf{g}_{i,j}(x)$$
$$R_i = (x)\big(\mathsf{tell}\, e_i(x) \twoheadrightarrow p_i(x).\, \mathsf{fuse}_x\, p_i(x).\, \|_j\, \mathit{give}_{i,j}\big)$$

As discussed in Ex. 3, we have that $\wedge_i e_i(n) \twoheadrightarrow p_i(n)$ *is enough to entail* $p_i(n)$ *for all* i*. So, all the* fuse *prefixes are fired, and all the* $\mathit{give}_{i,j}$ *continuations executed. Note that, as a process the above is actually no more complex than the simple handshaking of Ex. 4. Indeed, all the complexity ends up inside the contract, and not in the process code, as it should be.*

Example 7 (Insured Sale)**.** *We model a seller* S *who will ship any order as long as she is either paid upfront, or she receives an insurance from the insurance company* I*, which she trusts.*

$$s(x) = \mathsf{order}(x) \wedge (\mathsf{pay}(x) \vee \mathsf{insurance}(x)) \twoheadrightarrow \mathsf{ship}(x)$$
$$S = (x)\mathsf{tell}\, s(x).\mathsf{fuse}_x\, \mathsf{ship}(x).(S \mid \mathit{doShip}(x))$$

The process S *above is recursive, so that many orders can me shipped. The exposed contract is straightforward. Below, we model the insurer. As for the seller, this is a recursive process. The contract used here is trivial: a premium must be paid upfront.*

$$\begin{aligned} i(x) &= \mathsf{premium}(x) \twoheadrightarrow \mathsf{insurance}(x) \\ I &= (x)\mathsf{tell}\, i(x).\mathsf{fuse}_x\, \mathsf{insurance}(x). \\ &\qquad \big(I \mid \tau.\mathsf{check}\, \neg\mathsf{pay}(x).(\mathit{refundS}(x) \mid \mathit{debtCollect}(x))\big) \end{aligned}$$

When an insurance is paid for, the insurer will wait for some time, modelled by the τ *prefix. After that is fired, the insurer will check whether the buyer has not paid the shipped goods. In that case, the insurer will immediately indemnify the seller, and contact a debt collector to recover the money from the buyer.*

Note that S *and* I *can be specified without the need to explicitly mention a specific buyer. Indeed, the interaction between the parties is loosely specified, so that many scenarios are possible. For instance, the above* S *and* I *can interact with the following buyer, paying upfront:*

$$b_0(x) = \mathsf{ship}(x) \twoheadrightarrow \mathsf{order}(x) \wedge \mathsf{pay}(x)$$
$$B_0 = (x)\mathsf{tell}\, b(x).\mathit{receive}(x)$$

A buyer can also pay later, if provides insurance:

$$b_1(x) = \mathsf{ship}(x) \twoheadrightarrow \mathsf{order}(x) \wedge \mathsf{premium}(x)$$
$$B_1 = (x)\mathsf{tell}\, b(x).(\mathit{receive}(x) \mid \tau.\mathsf{tell}\, \mathsf{pay}(x))$$

Interaction is also possible with an "incautious" buyer which pays upfront without asking any shipping guarantees.

$$b_2(x) = \mathsf{order}(x) \land \mathsf{pay}(x) \qquad B_2 = B_0$$

Finally, interaction with a malicious buyer is possible:

$$b_3(x) = \mathsf{order}(x) \land \mathsf{premium}(x) \qquad B_3 = B_0$$

Here, the insurer will refund the seller, and start a debt collecting procedure. This is an example where a violated promise can be detected so to trigger a suitable recovery action. Summing up, note the role of the CloseFuse *rule in these scenarios: the minimality requirement makes sure the insurer is involved only when actually needed.*

Example 8 (All-you-can-eat)**.** *Consider a restaurant offering an all-you-can-eat buffet. Customers are allowed to have a single trip down the buffet linemeal, where they can pick anything they want. After the meal is over, they are no longer allowed to return to the buffet. In other words, multiple dishes can be consumed, but only in a single step.*

We model this scenario through the following processes:

$$\begin{aligned} \mathit{Buffet} &= (x)\ (\mathsf{pasta}(x) \mid \mathsf{chicken}(x) \mid \mathsf{cheese}(x) \mid \mathsf{fruit}(x) \mid \mathsf{cake}(x)) \\ \mathit{Bob} &= (x)\ \mathsf{fuse}_x\ \mathsf{pasta}(x) \land \mathsf{chicken}(x).\ \mathit{SatiatedB} \\ \mathit{Carl} &= (x)\ \mathsf{fuse}_x\ \mathsf{pasta}(x).\mathsf{fuse}_x\ \mathsf{chicken}(x).\ \mathit{SatiatedC} \end{aligned}$$

The Buffet above can interact with either Bob or Carl, and make them satiated (we assume that SatiatedB and SatiatedC have no free variables). Here, Bob eats both pasta and chicken in a single meal. By contrast, Carl eats the same dishes but in two different meals, thus violating the Buffet policy:

$$\mathit{Buffet} \mid \mathit{Carl} \ \ \to^* \ \ \mathit{SatiatedC} \mid P$$

Indeed, the Buffet should forbid Carl to eat the chicken, i.e. to fire the second fuse_x *. To enforce the Buffet policy, we first define the auxiliary operator* $\oplus$*. Let* $(p_i)_{i\in I}$ *be arbitrary PCL formulae. Take a fresh prime symbol* r*, a fresh name* o*, and fresh variables* $z, (z_i)_{i\in I}$*. Then,*

$$\bigoplus_{i\in I} p_i = (o)(z)(z_i)_{i\in I}(\mathsf{r}(o,z) \mid \|_{i\in I}\mathsf{r}(o,z_i) \to p_i)$$

To see how this works, consider the process $\oplus_{i\in I}\, p_i|Q$ *where* Q *fires a* fuse_x *which demands a subset of the constraints* $(p_i)_{i\in J}$ *with* $J \subseteq I$*. To deduce* p_i *we are forced to fuse* z_i *with* z *(and* x*); otherwise we can not satisfy the premise* $\mathsf{r}(o,z_i)$*. Therefore all the* $(z_i)_{i\in J}$ *are fused, while minimality of fusion ensures that the* $(z_i)_{i\in I\setminus J}$ *are not. After fusion we then reach*

$$(o)(m)\Big((z_i)_{i\in I\setminus J}(\,\|_{i\in I\setminus J}\ \mathsf{r}(o,z_i) \to p_i)|\,\|_{i\in J}(\mathsf{r}(o,m)|\mathsf{r}(o,m) \to p_i)\Big) \ \mid\ Q'$$

where m *is a fresh name resulting from the fusion. Note that the* $(p_i)_{i\in I\setminus J}$ *can no longer be deduced through fusion, since the variable* z *was "consumed" by the first fusion. The restricted name* o *prevents from interference with other similar*

$$\frac{P \xrightarrow{(\vec{x}\vec{n}\vec{a})(C \vdash^{J}_{\vec{x}} c)} P'}{P \xrightarrow{\tau} (\vec{n}\vec{a})P'\sigma} \text{ if } \begin{array}{l} C\sigma \vdash c\sigma \\ \sigma(\vec{x}) \subseteq \vec{n} \end{array} \qquad [\textsc{CloseJoin}]$$

$$\frac{P \xrightarrow{(x\vec{y}\vec{m}\vec{a})(C \vdash^{F}_{x} c)} P'}{P \xrightarrow{\tau} (n\vec{m}\vec{a})P'\sigma} \text{ if } \begin{array}{l} C \vdash^{\sigma} c \\ \sigma(x) = n \text{ fresh} \\ \sigma(\vec{y}) \subseteq n\vec{m} \end{array} \qquad [\textsc{CloseFuse}]$$

Figure 4: Alternative labelled semantics for fuse and join.

constraints which may be available in the context. The rough result is that $\oplus_i p_i$ allows a subset of the $(p_i)_{i \in I}$ to be demanded through fusion, after which the rest is no longer available.

We can now exploit the $\oplus$ operator to redefine the Buffet as:

$$Buffet' = (x)(\mathsf{pasta}(x) \oplus \mathsf{chicken}(x) \oplus \mathsf{cheese}(x) \oplus \mathsf{fruit}(x) \oplus \mathsf{cake}(x))$$

The new specification actually enforces the desired buffet policy:

$$Buffet' \mid Carl \;\; \not\rightarrowtail^{*} \;\; SatiatedC \mid P$$

3.4 Fusions Involving Many Names

Consider the semantics $\mathsf{fuse}_x\, c$ and $\mathsf{join}_x\, c$ (Def. 7). A $\mathsf{fuse}_x\, c$ fuses x and other variables in the context with a *single* fresh name n. Similarly, a $\mathsf{join}_x\, c$ instantiates x with one name from the context. While we consider multiple variables for fuse, in no case we consider multiple names.

In Def. 11 below we introdce an alternative semantics for fuse and join. This semantics allows these primitives to handle multiple names. A prefix $\mathsf{join}_{\vec{x}}\, c$ can now be indexed with a vector of variables $\vec{x}$, instead of a single variable x.

Definition 11. *The alternative labeled semantics is defined by all the rules in Fig. 1, except that* CloseJoin *and* CloseFuse *are replaced as in Fig. 4.*

The notion of local minimal fusion is adapted as follows.

Definition 12 (Local Minimal Fusion)**.** *The relation $C \vdash^{\sigma} c$ holds whenever:*

$$\exists C' \subseteq C \;:\; (C'\sigma \vdash c\sigma \;\wedge\; \nexists \sigma' \subset \sigma \;:\; C'\sigma' \vdash c\sigma')$$

The new semantics of $\mathsf{join}_{\vec{x}}\, c$ makes the process wait until a set of names $\vec{n}$ is found so that $c\{\vec{n}/\vec{x}\}$ is entailed. The instantiation of $\vec{x}$ need not be injective, in general. For instance, we have the following transition:

$$\Big((n)(m)\, \mathsf{p}(n, m)\Big) \mid \Big((x)(y)\, \mathsf{join}_{xy}\, \mathsf{p}(x, y).Q\Big) \rightarrowtail$$
$$(n)(m)\Big(\mathsf{p}(n, m) \mid Q\{n/x, m/y\}\Big)$$

In the same spirit, a $\mathsf{fuse}_x\, c$ can now involve multiple names. As in Def. 7, x is still instantiated to a fresh name n. Instead, the variables $\vec{y}$ taken from the context do not need to be instantiated with n. We allow them to be instantiated to either n *or* or any other name found in the context. For instance, we have the following transition, where n, m, o are names, and x, y, z, w are variables.

$$\Big((z)(m)(w)\ \mathsf{p}(z,m,w)\Big) \mid \Big((x)(y)(o)\ \mathsf{fuse}_x\ \mathsf{p}(x,y,o).Q\Big) \rightarrowtail$$
$$(n)(m)\Big(\mathsf{p}(n,m,o) \mid Q\{n/x, m/y\}\Big)$$

Note how the variable z in the context had to be instantiated with the existing name o. Dually, the variable y had to be fused with name m from the context. Variables x, w instead are fused with the fresh name n.

Example 9 (Synchronization involving multiple names)**.** *Let:*

$$P = (x)(y)(n)\ \mathsf{p}(x,n,y)$$

The constraint P represents an "open" contract, in which the name n is fixed, but the variables x, y can be instantiated by the context. For instance, the process:

$$P \mid (w)(z)(m)\ \mathsf{fuse}_w\ \mathsf{p}(w,z,m).Q$$

is able to fire the fuse *prefix. The variable w will be fused with x: both are instantiated to a fresh name (say, o). The variable z instead must be instantiated to the existing name n. Similarly, the variable y must be instantiated to the name m. The residual process will then be (assuming o, n fresh in Q):*

$$(o)(n)(m)\Big(\mathsf{p}(o,n,m) | Q\{o/w, n/z\}\Big)$$

Example 10 (Linear disjunction)**.** *We exploit the alternative semantics of* fuse *to construct a variant $\boxplus$ of the all-you-can-eat operator $\oplus$ introduced in Ex. 8. Intuitively, $\oplus$ allows one to consume any number of dishes, as long as a single* fuse *prefix is involved. The operator $\boxplus$ is stricter than $\oplus$, since it allows only* one *dish to be consumed.*

Formally, $\boxplus$ is defined as follows:

$$\boxplus_{i=1}^{n} p_i = (\bar{m}, m_1, \ldots, m_n, y)(\mathsf{r}(\bar{m}, y) | \parallel_{i=1}^{n} (\mathsf{r}(\bar{m}, m_i) \rightarrow p_i))$$

Above, $\bar{m}$ and the m_i are names, while y is a variable. When a fuse *demands a constraint p_i, this will cause the fusion of y and m_i, so that the guard $\mathsf{r}(\bar{m}, m_i)$ can be discharged. The restricted name $\bar{m}$ prevents from interference with other similar constraints which may be available in the context.*

In the following example, Bob is not be allowed to fire its fuse *, since it requires two dishes. Carl can instead fire its first* fuse *and obtain the pasta; however, he cannot eat the chicken as well.*

$$\begin{aligned}
\mathit{Buffet} &= (x)\ (\mathsf{pasta}(x) \boxplus \mathsf{chicken}(x) \boxplus \mathsf{cheese}(x) \boxplus \mathsf{fruit}(x) \boxplus \mathsf{cake}(x))\\
\mathit{Bob} &= (x)\ \mathsf{fuse}_x\ \mathsf{pasta}(x) \wedge \mathsf{chicken}(x).\ \mathit{SatiatedB}\\
\mathit{Carl} &= (x)\ \mathsf{fuse}_x\ \mathsf{pasta}(x).\mathsf{fuse}_x\ \mathsf{chicken}(x).\ \mathit{SatiatedC}
\end{aligned}$$

The above works as intended. Since y can be fused with one name m_i, only one dish can be consumed: therefore, Bob cannot eat two dishes, and Carl is prevented to eat a second time.

Note that, under the standard semantics of fuse *given in Def. 7, the $\boxplus$ operator above would* not *work, since y could be fused only with a fresh name, hence not with any of the m_i. Instead, the semantics in Def. 11 allows to instantiate x to a fresh name while fusing y with one of the m_i.*

In real-world contracts, the binding between the principals and the promises they are making is usually made explicit. For instance, when Alice says "I will lend my airplane provided that I will borrow a bike", she is actually issuing the contract "Alice says that she will lend her airplane provided that she will borrow a bike". To cope with this, in [3] we have shown an extensions of the logic PCL with a *says* modality which allows for expressing, within a contract, the identity of the principal who is promising something. For instance, Alice's contract can now take the form:

$$Alice\ says\ (z\ says\ \mathsf{b}(x)) \twoheadrightarrow \mathsf{a}(x)$$

This means that a principal named Alice is promising her airplane a in a session x, provided that some (unknown) principal z is promising a bike b. The additional information contained in such contract can be exploited, e.g. to single out the principal who is responsible for a violation, and possibly to take countermeasures against him.

The alternative semantics in Def. 11 allows for dealing with such kind of contracts. For instance, consider the formula $x\ says\ \mathsf{p}(y)$. To reach an agreement, the variable x has to be instantiated to the name of a principal, while y has to be instantiated to a fresh session ID. In general, more than two names can be involved in the fusion; for this reason, the semantics Def. 7 is unsuited for dealing with principals.

Example 11 (Principals). *Consider an online market, where buyers and sellers trade items. The contract of a buyer n_B is to pay for an item, provided that some (still unknown) seller x_S promises to send it; dually, the contract of a seller n_S is to send an item, provided that some buyer y_B pays.*

$$c_B = n_B\ says\ \big((x_S\ says\ \mathsf{send}(x)) \twoheadrightarrow \mathsf{pay}(x)\big)$$
$$c_S = n_S\ says\ \big((y_B\ says\ \mathsf{pay}(y)) \twoheadrightarrow \mathsf{send}(y)\big)$$

A buyer first issues her contract c_B, then waits until discovering she has to pay, and eventually proceeds with the process B'. At this point, the buyer may either refuse to pay (process NoPay), or actually pay the item, by issuing a $\mathsf{paid}(x)$. *After the item has been paid, the buyer may wait for the item to be sent or open a dispute with the seller.*

$$B = (x)(x_S)(n_B)\ \big(\mathsf{tell}\, c_B.\ \mathsf{fuse}_x\ (n_B\ says\ \mathsf{pay}(x)).\ B'\big)$$
$$B' = \tau.NoPay\ +\ \tau.\ \mathsf{tell}\ (n_B\ says\ \mathsf{paid}(x)).\ B''$$
$$B'' = \mathsf{ask}\ (x_S\ says\ \mathsf{sent}(x)) + \tau.\ \mathsf{tell}\ (n_B\ says\ \mathsf{dispute}(x))$$

The behaviour of the seller n_S is dual.

$$S = (y)(y_B)(n_S)\ \big(\mathsf{tell}\, c_S.\ \mathsf{fuse}_y\ (n_S\ says\ \mathsf{send}(y)).\ S'\big)$$
$$S' = \tau.NoSend\ +\ \tau.\ \mathsf{tell}\ (n_S\ says\ \mathsf{sent}(y)).\ S'$$
$$S'' = \mathsf{ask}\ (y_B\ says\ \mathsf{pay}(y)) + \tau.\ \mathsf{tell}\ (n_S\ says\ \mathsf{dispute}(y))$$

An handshaking is reached through the fusion $\sigma = \{m/x, m/y, n_S/x_S, n_B/y_B\}$, where m is fresh.

To automatically resolve disputes, a judge J can enter a session initiated between a buyer and a seller, provided that a dispute has been opened, and *either the obligations* pay *or* send *have been inferred. This is done through the* $\mathsf{ask}_{\vec{x}}$ *primitive, where* $\vec{x} = \{z, x_S, y_B\}$*. This binds the variable* z *to the session identifier* m*,* x_S *to the actual name of the seller (*n_S*), and* y_B *to the actual name of the buyer (*n_B*).*

$$\begin{aligned} J = (\vec{x})\big(&\mathsf{ask}_{\vec{x}}\,(y_B \textit{ says } \mathsf{pay}(z) \land x_S \textit{ says } \mathsf{dispute}(z)).\\ &\qquad \mathsf{check}\,\neg(y_B \textit{ says } \mathsf{paid}(z)).\,\textit{jail}(y_B)\\ &\mid \mathsf{ask}_{\vec{x}}\,(x_S \textit{ says } \mathsf{send}(z) \land y_B \textit{ says } \mathsf{dispute}(z)).\\ &\qquad \mathsf{check}\,\neg(x_S \textit{ says } \mathsf{sent}(z)).\,\textit{jail}(x_S)\big)\end{aligned}$$

If the obligation $\mathsf{pay}(z)$ *is found, but the item has not been actually paid then the buyer is convicted (modelled by jail(*y_B*), not further detailed). Similarly, if the obligation* $\mathsf{send}(z)$ *has not been supported by a corresponding* $\mathsf{sent}(z)$*, then the seller is convicted.*

4 Expressive power

We now discuss the expressive power of our synchronization primitives, by showing how to encode some common concurrency idioms into our calculus.

4.1 Semaphores

Semaphores admit a simple encoding in our calculus. Below, n is the name associated with the semaphore, while x is a fresh variable. $P(n)$ and $V(n)$ denote the standard semaphore operations, and process Q is their continuation.

$$P(n).Q = (x)\,\mathsf{fuse}_x\,\mathsf{p}(n,x).Q \qquad V(n) = (x)\,\mathsf{tell}\,\mathsf{p}(n,x)$$

Each $\mathsf{fuse}_x\,\mathsf{p}(n,x)$ instantiates a variable x such that $\mathsf{p}(n,x)$ holds. Of course, the same x cannot be instantiated twice, so it is effectively consumed. New variables are furnished by $V(n)$.

4.2 Memory cell

We model below a memory cell in our calculus. The cell at any time contains a name v as its value.

$$\begin{aligned} \textit{New}(n,v).Q &= (x)\mathsf{tell}\,\mathsf{c}(n,x) \land \mathsf{d}(x,v).Q\\ \textit{Get}(n,y).Q &= (w)\mathsf{fuse}_w\,\mathsf{c}(n,w).\mathsf{join}_y\,\mathsf{d}(w,y).\textit{New}(n,y).Q\\ \textit{Set}(n,v).Q &= (w)\mathsf{fuse}_x\,\mathsf{c}(n,w).\textit{New}(n,v).Q\end{aligned}$$

Process $New(n, v)$ initializes the cell having name n and initial value v. Process $Get(n, y)$ recovers v by fusing it with y: the procedure is destructive so the cell is re-created. Process $Set(n, v)$ destroys the current cell and creates a new one.

4.3 Linda

Our calculus can model a tuple space, and implement the insertion and retrieval of tuples as in Linda [23]. For illustration, we only consider p-tagged pairs here.

$$\begin{aligned} Out(w,y).Q &= (x)\mathsf{tell}\ \mathsf{p}(x) \wedge \mathsf{p}_1(x,w) \wedge \mathsf{p}_2(x,y).Q \\ In(w,y).Q &= (x)\mathsf{fuse}_x\ \mathsf{p}_1(x,w) \wedge \mathsf{p}_2(x,y).Q \\ In(?w,y).Q &= (x)\mathsf{fuse}_x\ \mathsf{p}_2(x,y).\mathsf{join}_w\ \mathsf{p}_1(x,w).Q \\ In(w,?y).Q &= (x)\mathsf{fuse}_x\ \mathsf{p}_1(x,w).\mathsf{join}_y\ \mathsf{p}_2(x,y).Q \\ In(?w,?y).Q &= (x)\mathsf{fuse}_x\ \mathsf{p}(x).\mathsf{join}_w\ \mathsf{p}_1(x,w).\mathsf{join}_y\ \mathsf{p}_2(x,y).Q \end{aligned}$$

Operation Out inserts a new pair in the tuple space. A fresh variable x is related to the pair components through suitable predicates. Operation In retrieves a pair through pattern matching. The pattern $In(w,y)$ mandates an exact match, so we require that both components are as specified. Note that the fuse prefix will instantiate the variable x, effectively consuming the tuple. The pattern $In(?w,y)$ requires to match only against the second component y. We do exactly that in the fuse prefix. Then, we use join to recover the first component of the pair, and bind it to variable w. Pattern $In(w,?y)$ is symmetric. Pattern $In(?w,?y)$ matches with any pair, so we specify a weak requirement for the fusion. Then we recover the pair components.

4.4 Synchronous π-calculus

We encode the synchronous π-calculus [28] into our calculus as follows:

$$\begin{aligned} [P \mid Q] &= [P] \mid [Q] \\ [(\nu n)P] &= (n)\,[P] \\ [\bar{a}\langle b\rangle.P] &= (x)\big(\mathsf{msg}(x,b)|\mathsf{fuse}_x\ \mathsf{in}(a,x).[P]\big) && (x \text{ fresh}) \\ [a(z).Q] &= (y)\big(\mathsf{in}(a,y)|(z)\mathsf{join}_z\ \mathsf{msg}(y,z).[Q]\big) && (y \text{ fresh}) \\ [X(\vec{a})] &= X(\vec{a}) \\ [X(\vec{y}) \doteq P] &= X(\vec{y}) \doteq [P] \end{aligned}$$

Our encoding preserves parallel composition, and maps name restriction to name delimitation, as one might desire. The output can not proceed with P until x is fused with some y. Dually, the input can not proceed until y is instantiated to a name, that is until y is fused with some x — otherwise, there is no way to satisfy $\mathsf{msg}(y,z)$.

The encoding above satisfies the requirements of [25]. It is *compositional*, mapping each π construct in a context of our calculus. Further, the encoding is *name invariant* and preserves *termination, divergence* and *success*. Finally it is *operationally corresponding* since, writing $\rightarrow_\pi$ for reduction in π,

$$\begin{aligned} P \rightarrow_\pi^* P' &\implies [P] \rightarrowtail^* \sim [P'] \\ [P] \rightarrowtail^* Q &\implies \exists P'.\ Q \rightarrowtail^* \sim [P'] \wedge P \rightarrow_\pi^* P' \end{aligned}$$

where $\sim$ is $\rightarrowtail$-bisimilarity. For instance, note that

$$\begin{aligned} &[(\nu m)(n\langle m\rangle.P|n(z).Q)] \rightarrowtail^* \\ &(m)(o)\big(\mathsf{msg}(o,m)|[P]|\mathsf{in}(n,o)|[Q]\{^m/_y\}\big) \sim (m)[P|Q\{^m/_y\}] \end{aligned}$$

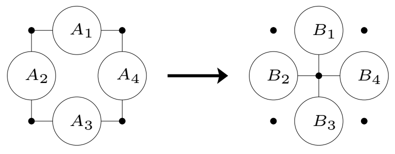
since the name o is fresh and the constraints $\mathsf{msg}(o, m), \mathsf{in}(n, o)$ do not affect the behaviour of P, Q. To see this, consider the inputs and outputs occurring in P, Q. Indeed, in the encoding of inputs, the fuse_x prefix will instantiate x to a fresh name, hence not with o. On the other hand, in the encoding of outputs, the join_z prefix can fire only after y has been fused with x, hence instantiated with a fresh name. The presence of $\mathsf{msg}(o, m)$ has no impact on this firing.

In our encoding above, we did not handle the choice operator. This is however manageable through the very same technique we will use below to encode graph rewriting, where the operator $\oplus$ introduced in Ex. 8 is used to properly encode non-deterministic choices.

4.5 Graph rewriting

In the encoding of the π-calculus we have modelled a simple interaction pattern; namely, Milner-style synchronization. Our calculus is also able to model more sophisticated synchronization mechanisms, such as those employed in graph rewriting techniques [18]. Before dealing with the general case, we introduce our encoding through a simple example.

Example 12. *Consider the following "ring-to-star" rewriting rule:*



Above, $A_1 \ldots A_4$ are processes, while bullets represent shared names. Whenever these processes are in a configuration matching the left side of the rule, a transition is enabled leading to the right side. Processes change to $B_1 \ldots B_4$, and a new name is created, which is shared among all of them, while the old names are forgotten. Modelling this kind of synchronization in, e.g., the π-calculus would be cumbersome, since a discovery protocol must be devised to allow processes to realize the transition is enabled. Note that no process is directly connected to the others, so this protocol is non-trivial.

Our calculus allows for an elegant, symmetric translation of the rule above, which is interpreted as an agreement among the processes $A_1 \ldots A_4$. Intuitively, each process A_i promises to change into B_i, while adjusting the names, provided all the others perform the analogous action. Since each A_i shares two names with the others, we write it as $A_i(n, m)$. We can now define the advertised contract: below, we denote addition and subtraction modulo four as $\oplus_4$ and $\ominus_4$, respectively.

$$a_i(n, m, x) = \mathsf{f}_{i \oplus_4 1}(x, m) \wedge \mathsf{s}_{i \ominus_4 1}(x, n) \twoheadrightarrow \mathsf{f}_i(x, n) \wedge \mathsf{s}_i(x, m) \tag{15}$$

An intuitive interpretation of f, s is as follows: $\mathsf{f}_i(x, n)$ states that n is the first *name of some process $A_i(n, -)$ which is about to apply the rule. Similarly for $\mathsf{s}_i(x, m)$ and the* second *name. The parameter x is a session ID, uniquely identifying the current transition. The contract $a_i(n, m, x)$ states that A_i agrees to fire the rule provided both its neighbours do as well. The actual A_i process is as follows.*

$$A_i(n, m) = (x)\mathsf{tell}\, a_i(n, m, x).\mathsf{fuse}_x\, \mathsf{f}_i(x, n) \wedge \mathsf{s}_i(x, m).\, B_i(x)$$

Our PCL *logic enables the wanted transition:*

$$P = \|_i\, A_i(n_i, n_{i\oplus_4 1}) \rightarrowtail^* (m)\|_i\, B_i(m)$$

Further, note that the above works even when nodes n_i are shared among multiple parallel copies of the same processes. For instance, $P|P$ will fire the rule twice, possibly mixing A_i components between the two P's.

We now deal with the general case of a graph rewriting system.

Definition 13. *An hypergraph G is a pair (V_G, E_G) where V_G is a set of vertices and E_G is a set of hyperedges. Each hyperedge $e \in E_G$ has an associated tag $tag(e)$ and an ordered tuple of vertices $(e_1, \ldots, e_k)$ where $e_j \in V_G$. The tag $tag(e)$ uniquely determines the arity k.*

Definition 14. *A graph rewriting system is a set of graph rewriting rules $\{G_i \Rightarrow H_i\}_i$ where G_i, H_i are the* source *and* target *hypergraphs, respectively. No rule is allowed to discard vertices, i.e. $V_{G_i} \subseteq V_{H_i}$. Without loss of generality, we require that the sets of hyperedges E_{G_i} are pairwise disjoint.*

In Def. 15 below, we recall how to apply a rewriting rule $G \Rightarrow H$ to a given graph J. The first step is to identify an embedding σ of G inside J. The embedding σ roughly maps $H \setminus G$ to a "fresh extension" of J (i.e. to the part of the graph that is created by the rewriting). Finally, we replace $\sigma(G)$ with $\sigma(H)$.

Definition 15. *Let $\{G_i \Rightarrow H_i\}_i$ be a graph rewriting system, and let J be a hypergraph. An embedding σ of G_i in J is a function such that:*

- $\sigma(v) \in V_J$ *for each* $v \in V_{G_i}$*, and* $\sigma(v) \notin V_J$ *for each* $v \in V_{H_i} \setminus V_{G_i}$
- $\sigma(e) \in E_J$ *for each* $e \in E_{G_i}$*, and* $\sigma(e) \notin E_J$ *for each* $e \in E_{H_i} \setminus E_{G_i}$
- $\sigma(v) = \sigma(v') \implies v = v'$ *for each* $v, v' \in V_{H_i} \setminus V_{G_i}$
- $\sigma(e) = \sigma(e') \implies e = e'$ *for each* $e, e' \in E_{G_i} \cup E_{H_i}$
- $tag(e) = tag(\sigma(e))$ *for each* $e \in E_{G_i} \cup E_{H_i}$
- $\sigma(e)_h = \sigma(e_h)$ *for each* $e \in E_{G_i} \cup E_{H_i}$ *and* $1 \leq h \leq k$

The rewriting relation $J \to K$ holds iff, for some embedding σ, we have

$$V_K = (V_J \setminus \sigma(V_{G_i})) \cup \sigma(V_{H_i}) \qquad E_K = (E_J \setminus \sigma(E_{G_i})) \cup \sigma(E_{H_i})$$

Note that the assumption $V_{G_i} \subseteq V_{H_i}$ of Def. 14 ensures that $V_J \subseteq V_K$, so no dangling hyperedges are created through rewriting.

4.5.1 A First (Unsuccessful) Attempt

We now try to encode graph rewriting in our calculus. We anticipate that our first attempt will *not* lead to a correct encoding, as we will see. Yet, in the process we will introduce several fundamental concepts, paving the way to the correct (but more complex) encoding, which we will describe later on.

To simplify our encoding, we make a mild assumption: we require each G_i to be a *connected* hypergraph. Then, we will try to encode a generic hypergraph

J in a compositional way: we assign a unique name n to each vertex in V_J, and then build a parallel composition of processes $A_{tag(e)}(\vec{n})$, one for each hyperedge e in E_J, where $\vec{n} = (n_1, \ldots, n_k)$ identifies the adjacent vertices. Note that since the behaviour of an hyperedge e depends on its tag, only, we index A with $t = tag(e)$. Note that t might be the tag of several hyperedges in each source hypergraph G_i. We stress this point: tag t may occur in distinct source graphs G_i, and each of these may have multiple hyperedges tagged with t. The process A_t must then be able to play the role of any of these hyperedges. The willingness to play the role of such a hyperedge e relatively to a single node n is modelled by a formula $\mathsf{p}_{e,h}(x, n)$ meaning "I agree to play the role of e in session x, and my h-th node is n". The session variable x is exploited to "group" all the constraints related to the same rewriting. We use the formula $\mathsf{p}_{e,h}(x, n)$ in the definition of A_t. The process $A_t(\vec{n})$ promises $\mathsf{p}_{e,1}(x, n_1), \ldots, \mathsf{p}_{e,k}(x, n_k)$ (roughly, "I agree to be rewritten as e"), provided that all the other hyperedges sharing a node n_h agree to be rewritten according to their roles $\bar{e}$. Formally, the contract related to $e \in E_{G_i}$ is the following:

$$a_e(x, \vec{n}) = \bigwedge_{\substack{1 \le h \le k \\ \bar{e}_{\bar{h}} = e_h}} \mathsf{p}_{\bar{e},\bar{h}}(x, n_h) \twoheadrightarrow \bigwedge_{1 \le h \le k} \mathsf{p}_{e,h}(x, n_h) \tag{16}$$

Note that in Example 12 we indeed followed this schema of contracts. There, the hypergraph J has four hyperedges e_1, e_2, e_3, e_4, each with a unique tag. The formulae f_i and s_i in (15) are rendered as $\mathsf{p}_{e_i,1}$ and $\mathsf{p}_{e_i,2}$ in (16). Also the operators $\oplus_4$ and $\ominus_4$, used in (15) to choose neighbours, are generalized in (16) through the condition $\bar{e}_{\bar{h}} = e_h$.

Back to the general case, the process A_t will advertise the contract a_e for each e having tag t, and then will try to fuse variable x. Note that, since the neighbours are advertising the analogous contract, we can not derive any $\mathsf{p}_{e,h}(x, n_h)$ unless *all* the hyperedges in the connected component agree to be rewritten. Since G_i is connected by hypothesis, this means that we indeed require the whole graph to agree.

However, advertising the contracts a_e using a simple parallel composition can lead to unwanted results when non-determinism is involved. Consider two unary hyperedges, which share a node n, and can be rewritten using two distinct rules: $G \Rightarrow H$ with $e1, e2 \in E_G$, and $\bar{G} \Rightarrow \bar{H}$ with $\bar{e}1, \bar{e}2 \in E_{\bar{G}}$. Let $tag(e1) = tag(\bar{e}1) = t1$ and $tag(\bar{e}2) = tag(\bar{e}2) = t2$. Each process thus advertises two contracts, for instance:

$$\begin{aligned} A_{t1} &= (x)(a_{e1}(x, n) | a_{\bar{e}1}(x, n) | Fusion_{t1}) \\ A_{t2} &= (x)(a_{e2}(x, n) | a_{\bar{e}2}(x, n) | Fusion_{t2}) \end{aligned}$$

Processes $Fusion_{ti}$ above are meant to trigger the fusion of variables x. Consider now $A_{t1} | A_{t2}$. After the fusion of x, it is crucial that both hyperedges agree on the rewriting rule that is being applied – that is either they play the roles of $e1, e2$ or those of $\bar{e}1, \bar{e}2$. However, only *one* $Fusion$ process above will fire its fuse prefix, say the first one:

$$(m)(a_{e1}(m, n) | a_{\bar{e}1}(m, n) | Rewrite_{e1} | a_{e2}(m, n) | a_{\bar{e}2}(m, n) | Fusion_{t2}\{^m/_x\})$$

Above m is a fresh name. Note however that the process $Fusion_{t2}\{^m/_x\}$ can still proceed with the *other* rewriting, since the substitution above can not disable a fuse prefix which was enabled before. So, we can end up with $\mathit{Rewrite}_{\bar{e}2}$, leading to an inconsistent rewriting. Indeed, A_{t1} was rewritten using $G \Rightarrow H$, while A_{t2} according to $\bar{G} \Rightarrow \bar{H}$.

To avoid this, we resort to the construction $\oplus_i p_i$ discussed in Ex. 8. We can then define A_t as follows.

$$A_t(\vec{n}) = (x)(\bigoplus_{tag(e)=t} a_e(x, \vec{n})| \sum_{tag(e)=t} \mathsf{fuse}_x \bigwedge_{1 \le h \le k} \mathsf{p}_{e,n}(x, n_h).B_e(x, \vec{n}))$$

In each A_t, the contracts a_e are exposed under the $\oplus$. The consequences of these contracts are then demanded by a sum of fuse_x. We defer the definition of B_e.

Consider now the behaviour of the encoding of a whole hypergraph: $A_t(\vec{n})| \cdots |A_{t'}(\vec{n}')$. If the hypergraph J contains an occurrence of G, where $G \Rightarrow H$ is a rewriting rule, each of the processes involved in the occurrence $P_1, \ldots, P_l$ may fire a fuse_x prefix. Note that this prefix demands *exactly one* contract a_e from each process inside of the occurrence of G. This is because, by construction, each a_e under the same $\oplus$ involves distinct $\mathsf{p}_{e,n}$. This implies that, whenever a fusion is performed, the contracts which are not directly involved in the handshaking, but are present in the occurrence of G triggering the rewriting, are then effectively disabled. In other words, after a fusion the sums in the other involved processes have exactly one enabled branch, and so they are now committed to apply the rewriting coherently.

After the fusion $B_e(x, \vec{n})$ is reached, where x has been instantiated with a fresh session name m which is common to all the participants to the rewriting. It is then easy to exploit this name m to reconfigure the graph. Each involved vertex (say, with name n) can be exposed to all the participants through e.g. $\mathsf{tell}\,\mathsf{vert}_h(m, n)$, and retrieved through the corresponding $\mathsf{join}_y\,\mathsf{vert}_h(m, y)$. Since m is fresh, there is no risk of interference between parallel rewritings. New vertices (those in $V_H \setminus V_G$) can be spawned and broadcast in a similar fashion. Once all the names are shared, the target hypergraph H is formed by spawning its hyperedges E_H through a simple parallel composition of $A_t(\vec{n})$ processes – each one with the relevant names. Note in passing that the processes A_t, where t ranges over all the tags, are mutually recursive. Summing up, it is easy to construct $B_e(x, \vec{n})$ from the above discussion, so we omit a verbose formal definition.

Correctness. Whenever we have a rewriting $J \to K$, it is simple to check that the contracts used in the encoding yield an handshaking, so causing the corresponding transitions in our process calculus. The reader might wonder whether the opposite also holds, hence establishing an *operational correspondence.* It turns out that our encoding actually allows *more* rewritings to take place, with respect to Def. 15. Using the A_i from Ex. 12, we have that the following loop of length 8 can perform a transition.

$$\begin{aligned} P = & A_1(n_1, n_2)|A_2(n_2, n_3)|A_3(n_3, n_4)|A_4(n_4, n_5)| \\ & A_1(n_5, n_6)|A_2(n_6, n_7)|A_3(n_7, n_8)|A_4(n_8, n_1) \end{aligned}$$

Indeed, any edge here has exactly the same "local view" of the graph as the corresponding G of the rewriting rule. So, handshaking takes place. Roughly, if a graph J_0 triggers a rewriting rule in the encoding, then each "bisimilar" graph J_1 will trigger the same rewriting rule.

A possible solution to capture graph rewriting in an exact way would be to mention *all* the vertices in each contract. That is, edge A_1 would use $p_{A_1}(n_1, n_2, x, y)$, while edge A_2 would use $p_{A_2}(w, n_2, n_3, z)$, and so on, using fresh variables for each locally-unknown node. Then, we would need the fuse prefix to match these variables as well, so that variables in each contract are fused with names used in the others. This would precisely establish the embedding σ of Def. 15. This kind of extended fusion is actually provided by the alternative semantics of fuse, which we discussed in Sect. 3.4, and which was first introduced in [3]. By exploiting this general fusion, we now give a correct encoding.

4.5.2 Towards a Correct Encoding

We now use the alternative semantics for $\mathsf{fuse}_x\, c$ given in Def. 11 to construct a correct encoding. We present our encoding in an incremental way. In this section, we make some simplifying assumptions on the rewriting rules $G_i \Rightarrow H_i$. All of these will be discharged in the following sections.

- *(Connection)* Each G_i is connected.
- *(Tag Uniqueness)* A tag occurs at most once in each G_i. That is, for all $e, e' \in E_{G_i}$, if $tag(e) = tag(e')$ then $e = e'$.
- *(Single Adjacency)* No vertex in G_i can have multiple adjacency with an edge. That is, for each $e \in E_{G_i}$, if $e_i = e_j$ then $i = j$.

Note that the above requirements indeed restrict the rewriting rules, only. No assumption is made on the graph J which is being rewritten, which can violate any of the above.

For each rewriting rule $G_i \Rightarrow H_i$ we fix an arbitrary ordering for the vertices and edges of G_i.

$$V_{G_i} = (v^{i,1}, \ldots, v^{i,l}) \qquad E_{G_i} = (e^{i,1}, \ldots, e^{i,k})$$

When G_i is clear from the context, we shall omit the index i.

Similarly to the previous encoding, we use one predicate $\mathsf{p}_e(x, \hat{v}^1, \cdots, \hat{v}^l)$ for each edge $e \in E_{G_i}$. The arity of p_e is $l + 1$. The intended meaning of the argument is as follows. The first argument is intended to be the "session identifier", as usual. The rest of the arguments shall list the names representing the identities of the vertices of the embedding of G_i in J. As before each hyperedge in the current graph will be encoded by a process $A_t(\vec{n})$, where the names $\vec{n}$ represent the adjacent vertices to the edge having tag t.

Below, x, w_i are variables, while id, n_i are names. We also let $\hat{l} = \max_i |V_{G_i}|$.

$$A_t(\vec{n}) = (x, w_1, \ldots, w_{\hat{l}})(\bigoplus_{\substack{e \in E_{G_i} \\ tag(e)=t}} a_e(\mathcal{S}) | \sum_{\substack{e \in E_{G_i} \\ tag(e)=t}} \mathsf{fuse}_x\, \mathsf{p}_e(\mathcal{S}).B_e(x))$$
$$\text{where } \mathcal{S} = \mathcal{SHUFFLE}(e, x, \vec{n}, \vec{w})$$

$$a_e(\vec{b}) = \Big(\bigwedge_{\substack{\bar{e} \text{ s.t.} \\ \exists \bar{h}, h.\ \bar{e}_{\bar{h}} = e_h}} \mathsf{p}_{\bar{e}}(\vec{b}) \Big) \twoheadrightarrow \mathsf{p}_e(\vec{b})$$

$$\mathcal{SHUFFLE}(e, x, \vec{n}, \vec{w}) = (x, \hat{v}^1, \ldots, \hat{v}^l)$$
$$\text{where} \quad \hat{v}^j = \begin{cases} n_m & \text{if } e_m = v^{i,j} \text{ for some } m \in [1..arity(e)],\ E_{G_i} \ni e \\ w_j & \text{otherwise} \end{cases}$$

Note that the last line above is well-defined since m is unique, as implied by the Single Adjacency assumption.

The contracts $\mathsf{p}_e(x, \hat{v}^1, \ldots, \hat{v}^l)$ involve all the vertices of G_i, with $e \in G_i$. Since $A_t(\vec{n})$ knows its adjacent nodes $\vec{n}$, these can be used to specify the related $\hat{v}^i$ parameters. Instead, $A_t(\vec{n})$ has no knowledge of the other vertices, so it is not able to precisely specify them in its contract p_e. These parameters are instead filled with "wildcard" variables w_j. Formally, the correct arrangement of $\vec{n}$ and $\vec{w}$ is defined through the $\mathcal{SHUFFLE}$ function. These contracts are then combined so to form the constraint $a_e(\cdots)$. Finally, the $\oplus$ operator is applied to each a_e, where e ranges over all edges having tags t; this is because A_t can play the role of each such e.

Firing a $\mathsf{fuse}_x\, \mathsf{p}_e(x, \ldots)$ prefix performs a number of variable fusions, "unifying" the contracts. Suppose that in the context the following contracts can be found, encoding a rewriting rule where $E_G = \{e, e'\}$.

$$(x, w_1, w_2, w_3)(\mathsf{p}_{e'}(x, n_1, w_2, n_3) \twoheadrightarrow \mathsf{p}_e(x, n_1, w_2, n_3) \oplus \cdots)$$
$$(x, w_1, w_2, w_3)(\mathsf{p}_e(x, w_1, n_2, n_3) \twoheadrightarrow \mathsf{p}_{e'}(x, w_1, n_2, n_3) \oplus \cdots)$$

Above, the first process/edge is adjacent to n_1 and n_3, and is willing to play the role of e. The second is adjacent to n_2 and n_3. Suppose that the first edge wants to fire the following fuse prefix:

$$\mathsf{fuse}_x\, \mathsf{p}_e(x, n_1, w_2, n_3).B_e(x)$$

This can be done by fusing $w_1 = n_1, w_2 = n_2, w_3 = n_3$ so that the contractual implications indeed form a handshaking, implying the constraint required by the fuse [1]. As another result of the fusion, x is replaced with a fresh session ID. (For the sake of completeness, also some variables hidden in the definition of $\oplus$ are fused as well.)

The above argument supports the fact that the intended rewriting does take place, as it happened for the tentative encoding of Sect. 4.5.1. We now support the fact that no other rewritings but the intended one occur.

First, recall the counterexample of Sect. 4.5.1, where an 8-ring could trigger a rewriting requiring a e-ring. This does not happen in the new encoding,

[1] Note in passing that this fusion requires the semantics of Def. 11.

since each vertex is made explicit in contracts. Indeed, to trigger such a wrong rewriting fusion would have to unify 8 contracts which carry 8 distinct names into a 4-tuple of names — which is impossible. More in general, since vertices are enumerated in the contracts p_e, no misunderstanding can happen regarding vertices.

We now consider edges. A possible source of confusion here is the fact that the $\oplus_e\ a_e(\cdots)$ construct allows a fuse prefix to demand many a_e constraints at the same time. If this happens, we would have that a single process A_t is playing the role of *many* edges in the rewriting rule: this would be unsound, since graph embeddings are meant to be injections over edges. To rule out this possibility, we first observe that the fuse prefix will demand a p_e contract for some $e \in E_{G_i}$. By construction of a_e, only contracts arising from the *same* G_i can contribute to the entailment of p_e. Hence, by the minimality of our fusions, no contracts will be demanded from $\oplus_e\ a_e(\cdots)$ but those related to G_i. So, if two constraints $a_e, a_{e'}$ are demanded then e, e' share the same G_i, and by construction of A_t they also share the same tag. By the Tag Uniqueness assumption, this can only happen when $e = e'$.

By the above reasoning, when a fusion occurs, each involved process plays the role of exactly one of the edges in a G_i. Also, by the definition of the constraints a_e, and the fact that the G_i are connected, it is easy to see that contracts p_e are involved for all the edges of G_i. This can only happen if each of these p_e is provided by some A_t. So, when a fusion happens, the processes involved indeed form the encoding of an embedding of G_i. Hence, no rewritings are allowed other that those allowed by the rewriting rules.

4.5.3 Relaxing Tag Uniqueness

In order to relax tag uniqueness, we need to watch out for those G_i having multiple edges e, e' sharing the same tag. As we discussed in the previous section, the danger here is that the same process could pose *both* as the embedding of two distinct edges e and e'. We need to prevent this from happening.

Essentially, all we have to ensure is that only a single a_e constraint from each A_t can be demanded by a fuse prefix. A simple way to do this is to exploit the linear disjunction operator $\boxplus$ we introduced in Ex. 10. By just replacing $\oplus$ with $\boxplus$, we recover the correctness of the encoding.

$$A_t(\vec{n}) = (x, w_1, \ldots, w_{\hat{l}})(\boxplus_{\substack{e \in E_{G_i}\\ tag(e)=t}} a_e(\mathcal{S}) | \sum_{\substack{e \in E_{G_i}\\ tag(e)=t}} \mathsf{fuse}_x\ \mathsf{p}_e(\mathcal{S}).B_e(x))$$

The definitions of $\hat{l}, a_e$, and $\mathcal{S}$ are unchanged.

4.5.4 Relaxing Single Adjacency

The *single adjacency* requirement implies that any edge $\bar{e}$ having tag equal to $tag(e)$ is an embedding of e. Without this requirement, instead, sharing a tag is a necessary but (in general) not sufficient condition for being an embedding. This is because it may happen that we have $e_i = e_j$ in the rewriting rule (i.e. in G), but $\bar{e}_i \neq \bar{e}_j$ in the graph to be rewritten (i.e. in J). In this case, no embedding σ may map e to $\bar{e}$. Technically, this can be seen by checking the requirements of Def. 15. Indeed, the requirements imply $\bar{e}_i = \sigma(e)_i = \sigma(e_i) = \sigma(e_j) = \sigma(e)_j = \bar{e}_j$ which is not the case. So, no σ is allowed to map e to $\bar{e}$.

The above reasoning show that our encoding $A_t(\vec{n})$ is *not* allowed to advertise its contracts p_e under the lone assumption $tag(e) = t$. We need to check whether the vertices $\vec{n}$ satisfy the required multiplicities (w.r.t. e) before an edge can expose its contract (p_e). Our logic PCL does not feature an equality predicate, which would help in expressing the multiplicity requirements as, for instance, $(n_j = n_k) \to (\cdots \twoheadrightarrow \mathsf{p}_e(\cdots))$. However, we can achieve the same effect exploiting the freshness of names. The idea is the following: let id be a fresh name (i.e. local to the edge), and q be a fixed predicate symbol. Then, the proposition

$$\mathsf{q}(id, n_j) \land \Big(\mathsf{q}(id, n_k) \to (\cdots \twoheadrightarrow \mathsf{p}_e(\cdots))\Big)$$

allows p_e to be demanded by a fusion only if $n_j = n_k$. This is because PCL is a propositional logic, hence the *prime* hypothesis $\mathsf{q}(id, n_k)$ can only be discharged if it occurs in the context[2]. Since id is local to the edge, no interference may occur from other edges: the hypothesis can indeed be discharged if and only if $n_j = n_k$, as we wanted. This idea can then be generalized to several equality constraints in a simple way. Indeed, it suffices to use one fresh symbol q_j for each name n_j in $\vec{n}$.

Our encoding is then changed as follows. Below, id is a name, while x, w_i are variables. We let $\hat{l}$ to be $\max_i |V_{G_i}|$, as before.

$$A_t(\vec{n}) = (id, x, w_1, \ldots, w_{\hat{l}}) \left(\begin{array}{ll} \|_{j=1}^{|\vec{n}|} \ \mathsf{q}_j(id, n_j) & | \\ \boxplus_{\substack{e \in E_{G_i} \\ tag(e) = t}} \ a_e(id, \vec{n}, \mathcal{S}) & | \\ \sum_{\substack{e \in E_{G_i} \\ tag(e) = t}} \ \mathsf{fuse}_x \ \mathsf{p}_e(\mathcal{S}).B_e(x) & \end{array} \right)$$
$$\text{where } \mathcal{S} = \mathcal{SHUFFLE}(e, x, \vec{n}, \vec{w})$$

$$a_e(id, \vec{n}, \vec{b}) = \Big(\bigwedge_{\substack{j,k \text{ s.t.} \\ e_j = e_k}} \mathsf{q}_j(id, n_k) \Big) \to \Big[\Big(\bigwedge_{\substack{\bar{e} \text{ s.t.} \\ \exists h, h'.\, \bar{e}_h = e_{h'}}} \mathsf{p}_{\bar{e}}(\vec{b}) \Big) \twoheadrightarrow \mathsf{p}_e(\vec{b}) \Big]$$

$$\mathcal{SHUFFLE}(e, x, \vec{n}, \vec{w}) = (x, \hat{v}^1, \ldots, \hat{v}^l)$$
$$\text{where } \hat{v}^j = \begin{cases} n_m & \text{if } m = \min\{m | e_m = v^{i,j}\}, E_{G_i} \ni e \\ w_j & \text{if } \{m | e_m = v^{i,j}\} = \emptyset, E_{G_i} \ni e \end{cases}$$

Above, in A_t we provide the $\mathsf{q}_j(id, n_j)$ constraints to state that n_j is the j-th vertex of the edge at hand. The rest of the process A_t is unchanged. Instead, in $a_e(\cdots)$ we now require $\mathsf{q}_j(id, n_k)$, which plays the role of $n_j = n_k$, for each multiple adjacency found in e.

The definition of $\mathcal{SHUFFLE}$ is adapted so to consider multiple adjacency. When arranging the known vertices in the contracts p_e, we might find *many*

[2] Actually, in PCL a prime formula could also be derived from $\bot$. In our encoding, however, we never employ negative formulae, so the consistency of the context is ensured by Lemma 2.

copies of the same vertex in the vector $\vec{n}$, so we pick one of them: n_m in the definition above. When A_t is an embedding of e, that is when A_t respects the multiplicities of e, the set $\{n_m | e_m = v^{i,j}\}$ is a singleton, so our choice of a specific m is actually immaterial. When instead A_t is not a encoding of e, that is when multiplicities are not respected, the set $\{n_m | e_m = v^{i,j}\}$ is not a singleton, and our choice of m selects a vertex n_m carrying no useful meaning w.r.t. graph rewriting. In this case, the contract $\mathsf{p}_e(\cdots)$ will indeed be filled with "garbage" names. In principle, it could be dangerous to advertise such "garbage" contracts, since they may trigger unintended rewritings. However, in our case the "garbage" contract $\mathsf{p}_e(\cdots)$ is guarded by a $\mathsf{q}_j(id, n_k)$ premise which can not be discharged because $n_j \neq n_k$, so this is actually safe. This argument supports the fact that no other rewriting is triggered except the intended ones.

4.5.5 Relaxing Connection

Our a_e constraints provide p_e whenever for every edge $\bar{e}$ sharing (at least) a vertex with e we have $\mathsf{p}_{\bar{e}}$. That is, in the contractual implication, only edges $\bar{e}$ which are "neighbours" of e are mentioned. When the graph G_i is connected, in order to cause an handshaking we actually need to involve all the $\mathsf{p}_{\bar{e}}$ for each $\bar{e} \in V_{G_i}$. This is because the definition of a_e forces us to close the set of involved edges $\bar{e}$ under the neighbourhood relation. This is the intended behaviour, since we do not want to trigger a rewriting unless a process for each edge in G_i is involved.

If instead the graph G_i is not connected, it is no longer enough to consider only the neighbourhood, since that will not include all the required edges in the handshaking. We can however change our definition of a_e so to consider all the $\bar{e}$ which belong to the same E_{G_i} as e. Essentially, the new a_e uses a "greedy handshaking" as its contract.

$$a_e(id, \vec{n}, \vec{b}) = \Big(\bigwedge_{\substack{j,k \text{ s.t.} \\ e_j = e_k}} \mathsf{q}_j(id, n_k) \Big) \to \Big[\Big(\bigwedge_{\substack{\bar{e} \text{ s.t.} \\ \exists i.\ \bar{e}, e \in E_{G_i}}} \mathsf{p}_{\bar{e}}(\vec{b}) \Big) \twoheadrightarrow \mathsf{p}_e(\vec{b}) \Big]$$

The definitions of A_t and $\mathcal{SHUFFLE}$ are unchanged.

5 Related Work

Various approaches to the problem of providing both clients and services with provable guarantees about each other's functional behaviour have been studied over the last few years. Yet, at the present no widespread technology seems to give a general solution to this problem.

Recent research papers address the problem of defining contracts that specify the interaction patterns among (clients and) services [9, 10, 11, 29]. For instance, in [10] a contract is a process in a simplified CCS featuring only prefixing, internal and external choice. A client contract is compliant with a service contract if any possible interaction between the client and the service will always succeed. There, a main problem is how to define (and decide) a subcontract relation, that allows for safely substituting services without affecting the compliance with their clients. Even assuming that services are trusted and respect the published contract, this approach provides the client with no provable guarantees, except that the interaction with the service will "succeed", that is all the expected synchronizations will take place.

For instance, consider a simple buyer-seller scenario. In our vision, it is important to provide the buyer with the guarantee that, after the payment has been made, then either the payed goods are made available, or a full refund is issued. For the seller, it is important to be sure that a buyer will not repudiate a completed transaction, so to obtain for free the goods already delivered. This could be modelled by the following contracts, assuming a perfect duality between buyer and seller:

$$\mathit{Buyer} = (\mathsf{ship} \lor \mathsf{refund}) \twoheadrightarrow \mathsf{pay} \qquad\qquad \mathit{Seller} = \mathsf{pay} \twoheadrightarrow (\mathsf{ship} \lor \mathsf{refund})$$

The above two contracts lead to an agreement, which allows the buyer for paying, and the seller for shipping or issuing a refund.

Instead, in [10] the contracts of the buyer and of the seller would take a very different form, e.g.:

$$\mathit{Buyer} = \overline{\mathsf{pay}}.\,(\mathsf{ship} + \mathsf{refund}) \qquad\qquad \mathit{Seller} = \mathsf{pay}.\,(\overline{\mathsf{ship}} \oplus \overline{\mathsf{refund}})$$

Intuitively, this means that the client will first output a payment, and then either receive the item, or receive a refund (at service discretion). Dually, the service will first input a payment, and then opt for shipping the item or issuing a refund. However, this is very distant from our notion of contracts.

First, the contracts exposed above are quite rigid, in that they precisely fix the order in which the actions must be performed. Even though in some cases this may be desirable, many real-world contracts seem to allow for a more liberal way of constraining the involved parties (e.g., "I will pay before the deadline").

Second, while the crucial notion if the contracts in [10] is *compatibility*, in our model we focus on the inferring the *obligations* that arise from a set of contracts. The key difference between the two notions is that, given a set of contracts, a compatibility check results in a yes/no output, while inferring the obligations provides a fine-grained quantification of the reached agreement. For instance, the obligations may identify who is responsible of each action mentioned in the contract. Our processes can then exploit this information to take some recovery action against clients and services which do not respect their promises.

A lot of work addresses the problem of managing service failures in long-running business transactions, see e.g. [12, 5, 8, 6]. Since, in a long-lived transaction, the standard rollback mechanism of database systems does not scale, the idea is to partition the long transaction into a sequence of smaller transactions, each of which is associated with a given compensation [20]. Compensations are recovery actions specified by the service designer, that will be run upon failures of the standard execution. For instance, we can model our buyer-seller scenario as follows in Compensating CSP [8]. The seller charges the buyer credit card, and then proceeds by shipping the ordered item. Simultaneously, the seller performs an availability check, to see whether the ordered item is in stock. If the item is not available, the service throws an exception, which triggers the compensation refundAmount. This compensation restores the original state of the buyer account, so it will actually perform a transaction rollback:

$$\begin{aligned}\mathit{Seller} \ = [\ & \mathsf{availCheck}; (\mathsf{ok}; \mathit{SKIPP} \ \Box\ \mathsf{notOk}; \mathit{THROWW}) \\ & \ || \ (\mathsf{debitAmount} \div \mathsf{refundAmount})\ ]; \mathsf{ship}\end{aligned}$$

Notice that the choice of the compensation is crucial; while "refundAmount" may be acceptable by any client, if the compensation was instead just a "15%Discount" on the next order, then not all the clients would have been perfectly happy. Actually, our main criticism to long-running transactions is that clients have no control on the compensations provided by services.

In our vision, instead, clients have the right to select those services that offer the desired compensations. For instance, we may exploit our logic to model the contract of a buyer that will pay provided that, if the ordered item is unavailable, then she will obtain a full refund, as well as a 15% discount on the next order:

$$Buyer \ = \big(\mathsf{unavailable} \to (\mathsf{refund} \land \mathsf{15\%discount})\big) \twoheadrightarrow \mathsf{pay}$$

Several logics for modelling contracts have been introduced over the years, taking inspiration and extending e.g. classical [16], modal [15, 1, 21], intuitionistic [2], deontic [33, 22], default [24] and defeasible logics [26]. This flourishing of logics is justified by the many facets of contracts that arise when modelling real-world scenarios, e.g. principals, authorizations, duties, delegation, mandates, regulations, assume-guarantee specifications, *etc.* (see [4] for a more direct comparison between these logics and PCL). We think none of these logics, including our PCL, captures *all* the facets of contracts. Actually, each of the aforementioned logics is designed to represent some particular aspect of contracts, e.g. obligations, permissions and prohibitions in deontic logics, violation of contracts in default and defeasible logics, and agreement in PCL. We argue that, since these aspects are orthogonal, it is possible to extend PCL with features from some of these logics. Note that the key feature of PCL, i.e. the ability of discovering which obligations arise from any set of contracts, has allowed us to design the contract-based mechanism used in our calculus to establish sessions.

6 Research Directions

The main design goals for the logic and the calculus proposed in this paper have been minimality and decidability. We expect, however, that some useful constructs can be added to our framework, to make it suitable for modelling even more complex scenarios. Of course, preserving the decidability will be a major concern while considering these extensions, as it will require to revisit the proof of the cut elimination theorem. We discuss below some of the more promising research directions.

First order features. We plan to extend logic with predicates and quantifiers. This will allow us to model more accurately those scenarios where a party issues a "generic" contract that can be matched by many parties. While this first order extension shall force us to drop the decidability result, we expect to find interesting decidable fragments of the logic, yet expressive enough to model many relevant situations. For instance, consider an e-commerce scenario, where a seller promises to ship the purchased item to a given address, provided that the customer will pay for that item. Aiming at generality, we make the seller contract parametric with respect to the item, customer and address. This can be modelled using a universal quantification over these three formal parameters:

$$\begin{aligned} Seller = \forall item, cust, addr :& \\ & \mathsf{pay}(item, cust, addr) \twoheadrightarrow \mathsf{ship}(item, addr) \end{aligned} \tag{17}$$

Now, assume that a customer (say, Bob) promises that he will pay for a drill, provided that the seller will ship the item to his address. This will be modelled by the following contract issued by Bob, where the actual parameters remark that the payment is made by Bob, and that the destination address is Bob's.

$$\mathit{Bob} = \mathsf{ship}(\mathsf{drill}, \mathsf{bobAddress}) \twoheadrightarrow \mathsf{pay}(\mathsf{drill}, \mathsf{Bob}, \mathsf{bobAddress})$$

Joining the two contracts above will yield the intended agreement:

$$\mathit{Seller} \wedge \mathit{Bob} \quad \rightarrow \quad \mathsf{pay}(\mathsf{drill}, \mathsf{Bob}, \mathsf{bobAddress}) \wedge \mathsf{ship}(\mathsf{drill}, \mathsf{bobAddress})$$

Explicit time. Time is another useful feature that may arise while modelling real-world scenarios. For instance, in an e-commerce transaction, a contract may state that if the customer returns the purchased item within 10 days from the purchase date, then she will have a full refund within 21 days from then. We plan a temporal extension of our logic, so to reason about the obligations that arise when the deadlines expire. Back to our e-commerce example, we could imagine to express the seller contract as the following formula, where the parameter t in $\mathsf{p}(t)$ tells the point in time where the "event" p occurs:

$$\mathit{Seller}(t) \;= \forall t' : (\mathsf{pay}(t) \wedge \mathsf{return}(t') \wedge t' < t + 10) \twoheadrightarrow \exists t'' < t' + 21 : \mathsf{refund}(t'')$$

From the point of view of the buyer, the contract says that the buyer is willing to pay, provided that she can obtain a full refund (within 21 days from the date of payment), whenever she returns the item within 7 days from the date of payment:

$$\mathit{Buyer}(t) \;=\; \big(\forall t' : \mathsf{return}(t') \wedge t' < t + 10 \rightarrow \exists t'' < t' + 21 : \mathsf{refund}(t'')\big) \twoheadrightarrow \mathsf{pay}(t)$$

In the presence of an agreement (i.e. a completed e-commerce transaction) between the customer and the seller on (say) January the 1st, 2009, we expect our extended logic able to deduce that, if the customer has returned the purchased item on January the 5th, then the seller is required to issue a full refund to the customer within January, the 26th.

$$\mathit{Buyer}(1.1.09) \wedge \mathit{Seller}(1.1.09) \wedge \mathsf{return}(5.1.09) \;\rightarrow\; \exists t'' < 26.1.09 : \mathsf{refund}(t'')$$

There are a number of techniques to explicitly represent time in logical systems, so we expect to be able to reuse some of them for extending PCL. These techniques range from Temporal Logic [19], to more recent approaches on temporal extensions of authorization logics like [17].

Such a temporal extension to the logic demands for a timed semantics for our calculus. This will allow to check whether a promise is violated in a given trace (e.g. the deadline passed). Another key aspect is characterizing honest and timely processes, i.e. processes that always fulfill their duties on time.

Analysis techniques. We plan to develop analysis techniques to formally and automatically prove the correctness of a service infrastructure, i.e. that the contracts are always respected, without the need for resorting to third parties (e.g. legal offices) external to the model. We will investigate analysis techniques for services and contracts, in order to provide clients and service providers with provable assurances about the behaviour of services. To do that, we will consider a variety of properties, concerning both standalone services and aggregations of services. One of the primary goals of these analyses will be that of determining whether a service actually respects the declared contract, i.e. if the facts promised by the service agree with its behaviour. It is highly desirable to verify a property of this kind before making a service available, because it prevents from the possibility of being legitimately charged for not having fulfilled a signed contract. We will analyse a variety of global properties about aggregations of cooperating services. For instance, we will study what happens when a service provider is not able to fulfill a contract because of an external service that does not deliver the negotiated functionalities. In such a scenario, the goal of our analysis will be that of exactly determining the liabilities of the involved parties, and to decide if it is always possible to compensate the client through suitable recovery actions. We will also focus on deciding if a party can always detect that a signed contract has not been fulfilled. This property is particularly relevant, because in the scenarios where it is respected, we can provide the offended party with the possibility of contesting a contract, by resorting to a third party that enforces its fulfillment.

Whenever the analysis techniques establish the compliance of a service with its own promises, one might wish to apply the techniques of [14] to derive an implementation which securely runs over an untrusted network.

Implementation issues. In our model of contracts we have abstracted from most of the implementation issues. For instance, in insecure environments populated by attackers, the operation of exchanging contracts requires particular care. Clearly, integrity of contracts is a main concern, so we expect that suitable mechanisms have to be applied to ensure that contracts are not tampered with.

Further, establishing an agreement between participants in a distributed system with unreliable communications appears similar to establishing *common knowledge* among the stipulating parties [27], so an implementation has to cope with the related issues. For instance, the fuse_x prefix requires a fresh name to be delivered among all the contracting parties, so the implementation must ensure everyone agrees on that name. Also, it is important that participants can be coerced to respect their contracts after the stipulation: to this aim, the implementation should at least ensure the non repudiation of contracts [39].

7 Conclusions

We have investigated the notion of contract from a logical perspective. To do that, we have first extended intuitionistic propositional logic with a new connective, that models contractual implication. We have provided the new connective with an Hilbert-style axiomatisation, which has allowed us to show some interesting properties and application scenarios for our logic. The main result about our logic is its decidability. To prove that, we have devised a Gentzen-style sequent calculus for the logic, which is equivalent to the Hilbert-style axiomatisation. Decidability then follows from the subformula property, which is enjoyed by our Gentzen rules, and by a cut elimination theorem, which we have proved in full details in [4]. We have implemented a proof search algorithm for PCL.

Our logic for contracts has served as a basic building block for designing a calculus of contracting processes. This is an extension of Concurrent Constraints, featuring a peculiar mechanism for the fusion of variables, which well suites to formalise contract agreements. We have shown our calculus expressive enough to model a variety of typical scenarios, and to encode some common idioms for concurrency, among which the π-calculus and graph rewriting.

Acknowledgements. Work partially supported by EU-FETPI Global Computing Project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers) and by the MIUR-PRIN project SOFT. (Tecniche Formali Orientate alla Sicurezza).

References

[1] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 4(15):706–734, 1993.

[2] Martín Abadi and Gordon D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, 1993.

[3] Massimo Bartoletti and Roberto Zunino. A calculus of contracting processes. To appear in LICS 2010.

[4] Massimo Bartoletti and Roberto Zunino. A logic for contracts. Technical Report DISI-09-034, DISI - Università di Trento, 2009.

[5] Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calculus for long running transactions. In *Proc. FMOODS*, 2003.

[6] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. POPL*, 2005.

[7] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *Proc. ESOP*, 2007.

[8] Michael J. Butler, C. A. R. Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, 2004.

[9] Samuele Carpineti and Cosimo Laneve. A basic contract language for web services. In *Proc. ESOP*, 2006.

[10] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.

[11] Giuseppe Castagna and Luca Padovani. Contracts for mobile processes. In *Proc. CONCUR*, volume 5710 of *Lecture Notes in Computer Science*. Springer, 2009.

[12] Mandy Chessell, Catherine Griffin, David Vines, Michael J. Butler, Carla Ferreira, and Peter Henderson. Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4):743–758, 2002.

[13] Mario Coppo and Mariangiola Dezani-Ciancaglini. Structured communications with concurrent constraints. In *Proc. Trustworthy Global Computing (TGC)*, volume 5474 of *Lecture Notes in Computer Science*. Springer, 2008.

[14] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5):573–636, 2008.

[15] Aspassia Daskalopulu and Tom Maibaum. Towards electronic contract performance. In *Proc. 12th International Workshop on Database and Expert Systems Applications*, 2001.

[16] Hasan Davulcu, Michael Kifer, and I.V. Ramakrishnan. CTR-S: A logic for specifying contracts in semantic web services. In *Proc. WWW04*, 2004.

[17] Henry DeYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time. In *Proc. CSF*, 2008.

[18] Hartmut Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In *Proc. of the Workshop on Graph-Grammars and Their Application to Computer Science*, 1987.

[19] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

[20] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD Conference*, 1987.

[21] Deepak Garg and Martín Abadi. A modal deconstruction of access control logics. In *Proc. FoSSaCS*, pages 216–230, 2008.

[22] Jonathan Gelati, Antonino Rotolo, Giovanni Sartor, and Guido Governatori. Normative autonomy and normative co-ordination: Declarative power, representation, and mandate. *Artificial Intelligence and Law*, 12(1-2):53–81, 2004.

[23] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[24] Georgios K. Giannikis and Aspassia Daskalopulu. The representation of e-contracts as default theories. In *New Trends in Applied Artificial Intelligence*, 2007.

[25] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. In *Proc. CONCUR*, 2008.

[26] Guido Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14(2-3), 2005.

[27] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990.

[28] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[29] Luca Padovani. Contract-based discovery and adaptation of web services. In *SFM*, 2009.

[30] Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *LICS*, 1998.

[31] The PCL web site. http://www.disi.unitn.it/∼zunino/PCL.

[32] Frank Pfenning. Structural cut elimination - I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, 2000.

[33] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *Proc. FMOODS*, 2007.

[34] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.

[35] Vijay Saraswat, Prakash Panangaden, and Martin Rinard. Semantic foundations of concurrent constraint programming. In *Proc. POPL*, pages 333–352. ACM, 1991.

[36] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.

[37] Anne Troelstra and Dirk van Dalen. *Constructivism in Mathematics, vol. 1.* North-Holland, 1988.

[38] Martin Wirsing, Rocco De Nicola, Stephen Gilmore, Matthias M. Hölzl, Roberto Lucchi, Mirco Tribastone, and Gianluigi Zavattaro. Sensoria process calculi for service-oriented computing. In *Proc. TGC*, 2006.

[39] Jianying Zhou. *Non-repudiation in Electronic Commerce.* Artech House, 2001.