



UNIVERSITY
OF TRENTO

DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL'INFORMAZIONE

38050 Povo – Trento (Italy), Via Sommarive 14
<http://www.disi.unitn.it>

A CALCULUS OF CONTRACTING PROCESSES

Massimo Bartoletti and Roberto Zunino

October 2009

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 $b \rightarrow a$ and Bob’s commitment B as $a \rightarrow b$. 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 $a \rightarrow b$, 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 $a \wedge b$ 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 $A \wedge B \rightarrow a \wedge b$ does *not* hold neither in classical nor in intuitionistic propositional calculus IPC [35], 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 $b \twoheadrightarrow a$. This form of, say, *contractual* implication, is stronger than the standard implication \rightarrow of IPC. Actually, $b \twoheadrightarrow a$ implies a not only when b is true, like IPC implication, but also in the case that a “compatible” contract, e.g. $a \twoheadrightarrow b$, holds. 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:

$$(b \twoheadrightarrow a) \wedge (a \twoheadrightarrow b) \rightarrow a \wedge b$$

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 [33, 6], with the peculiarity that in ours the constraints are PCL formulae. A process can assert a constraint c (a PCL formula) through the primitive $\text{tell } c$. For instance, the following process models Alice exposing her contract:

$$(x) \text{ tell } \mathbf{b}(x) \rightarrow \mathbf{a}(x)$$

Formally, this will add $\mathbf{b}(x) \rightarrow \mathbf{a}(x)$ 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 [36, 12], 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 [28].

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 $\text{fuse}_x c$ as follows:

$$\text{fuse}_x \mathbf{a}(x)$$

This is similar to the $\text{ask } c$ 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 , $\text{fuse}_x c$ also binds the variable x to an actual session identifier, shared among all the parties involved in the contract. In other words, $\text{fuse}_x c$ actually fuses all the processes agreeing on a given contract.

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

$$Alice = (x) (\text{tell } \mathbf{b}(x) \rightarrow \mathbf{a}(x). \text{fuse}_x \mathbf{a}(x). \text{lendAirplane})$$

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:

$$Bob = (y) (\text{tell } \mathbf{a}(y) \rightarrow \mathbf{b}(y). \text{fuse}_y \mathbf{b}(y). \text{lendBike})$$

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

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

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

1.2 Contributions

We propose a unified framework for modelling distributed systems that base their interactions on contracts. Technically, our contribution consists of the following points:

- We devise a minimal set of properties which are desirable in any logical formalization of contracts.
- We define a logic to model contracts, and to infer the duties deriving from those. Our logic satisfies all the properties identified above as desirable. Further, we support the logic with several examples.
- We study the proof theory of our logic. In particular, we show that our logic is consistent and decidable.
- We define a calculus for contracting processes, which exploits the logic to drive synchronizations between processes.
- We show that our calculus is suitable for modelling complex interactions of contracting parties. We also show how to embed the π -calculus and graph rewriting in our calculus.

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:

$$\vdash (p \multimap q) \wedge (q \multimap p) \rightarrow p \wedge q \quad (1)$$

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:

$$\vdash (p_1 \multimap p_2) \wedge \cdots \wedge (p_{n-1} \multimap p_n) \wedge (p_n \multimap p_1) \rightarrow p_1 \wedge \cdots \wedge p_n \quad (2)$$

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`, `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:

$$\textit{Buyer} = \text{ship} \multimap \text{clickpay} \quad \textit{Bank} = \text{clickpay} \multimap \text{pay} \quad \textit{Seller} = \text{pay} \multimap \text{ship}$$

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

$$\vdash \textit{Buyer} \wedge \textit{Bank} \wedge \textit{Seller} \multimap \text{pay} \wedge \text{ship}$$

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

$$\vdash (p \multimap p) \multimap p \tag{3}$$

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. $p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n$. The greedy handshaking can be stated as:

$$\vdash \bigwedge_{i \in 1..n} \left((p_1 \wedge \dots \wedge p_{i-1} \wedge p_{i+1} \wedge \dots \wedge p_n) \multimap p_i \right) \multimap p_1 \wedge \dots \wedge p_n \tag{4}$$

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 $p \multimap q$ becomes effective, i.e. implies the promise q , when it is matched by a dual contract $q \multimap p$. Even more directly, $p \multimap q$ should be effective also in the case that the premise p is already true:

$$\vdash p \wedge (p \multimap q) \multimap q \tag{5}$$

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:

$$\vdash (p \multimap q) \multimap (p \multimap q) \tag{6}$$

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

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:

$$\vdash p \multimap \top \tag{7}$$

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

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

$$\vdash (p \rightarrow q) \wedge (q \rightarrow r) \rightarrow (p \rightarrow r) \quad (8)$$

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

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:

$$\text{Customer} : \text{bookFlight} \rightarrow \text{pay}$$

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:

$$\text{AirLine} : \text{pay} \rightarrow \text{bookFlight} \wedge \text{freeDrink}$$

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:

$$\vdash \text{AirLine} \rightarrow (\text{pay} \rightarrow \text{bookFlight})$$

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

$$\vdash \text{Customer} \rightarrow (\text{bookFlight} \wedge \text{freeDrink} \rightarrow \text{pay})$$

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).

$$\vdash (p \rightarrow q) \wedge (q \rightarrow q') \rightarrow (p \rightarrow q') \quad (9)$$

$$\vdash (p' \rightarrow p) \wedge (p \rightarrow q) \rightarrow (p' \rightarrow q) \quad (10)$$

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 :

$$\vdash q \rightarrow (p \rightarrow q) \quad (11)$$

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

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 $\{\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{s}, \dots\}$ of prime (atomic) formulae. Arbitrary PCL formulae are denoted with the letters p, q, r, s, \dots (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:*

$$p ::= \perp \mid \top \mid \mathbf{p} \mid \neg p \mid p \vee p \mid p \wedge p \mid p \rightarrow p \mid p \twoheadrightarrow p$$

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.*

$$\begin{array}{ll} \top \twoheadrightarrow \top & \text{[ZERO]} \\ (p \twoheadrightarrow p) \rightarrow p & \text{[FIX]} \\ (p' \rightarrow p) \rightarrow (p \twoheadrightarrow q) \rightarrow (q \rightarrow q') \rightarrow (p' \twoheadrightarrow q') & \text{[PREPOST]} \end{array}$$

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:*

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

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 [3].

Theorem 1 *PCL is consistent, i.e. $\not\vdash \perp$.*

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

$$\begin{array}{ll} \not\vdash (p \rightarrow q) \rightarrow (p \twoheadrightarrow q) & \not\vdash (p \twoheadrightarrow q) \rightarrow q \\ \not\vdash \perp \twoheadrightarrow p & \not\vdash ((q \rightarrow p) \rightarrow q) \rightarrow (p \twoheadrightarrow q) \end{array}$$

Note that if we augment our logic with the axiom of excluded middle, then $(p \rightarrow q) \leftrightarrow q$ 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 [30] with rules for the contractual implication connective \rightarrow .

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

$$\frac{\Gamma \vdash q}{\Gamma \vdash p \rightarrow q} \text{ [ZERO]} \quad \frac{\Gamma, p \rightarrow q, r \vdash p \quad \Gamma, p \rightarrow q, q \vdash r}{\Gamma, p \rightarrow q \vdash r} \text{ [FIX]} \quad \frac{\Gamma, p \rightarrow q, a \vdash p \quad \Gamma, p \rightarrow q, q \vdash b}{\Gamma, p \rightarrow q \vdash a \rightarrow b} \text{ [PREPOST]}$$

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 \rightarrow are not dual, unlike e.g. left/right rules for \wedge . Nevertheless, the structural approach of [30] can be adapted. Full details about proofs are in [3].

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 $\{\perp, \neg\}$, then $\not\vdash p \rightarrow \perp$.*

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 [34], the performance of our tool is acceptable for the examples presented in this paper. Our tool is available at [29].

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 \rightarrow 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 \perp ; it acts homomorphically on $\wedge, \vee, \rightarrow, \neg$; it satisfies $m(p \rightarrow q) = \mathcal{C}[m(p), m(q)]$ 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.*

$$\begin{aligned} m_0(p \multimap q) &= m_0(q) & m_1(p \multimap q) &= (m_1(q) \multimap m_1(p)) \multimap m_1(q) \\ m_2(p \multimap q) &= \neg\neg(m_2(q) \multimap m_2(p)) \multimap m_2(q) & m_3(p \multimap q) &= \neg(m_3(q) \multimap m_3(p)) \vee m_3(q) \\ m_4(p \multimap q) &= ((m_4(q) \multimap m_4(p)) \vee \mathbf{a}) \multimap m_4(q) \end{aligned}$$

where \mathbf{a} is any prime formula.

However, there can be no *sound* encodings, so \multimap 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 $\not\vdash p$.*

In [3] 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 (**refundS**). Also, the buyer promises that if he signs the preliminary contract, then either he will pay the stipulated price (**payB**), or he will not pay and lose the deposit (**refundB**).*

$$\text{Buyer} = ((\text{sellS} \vee \text{refundS}) \multimap \text{signB}) \wedge (\text{signB} \multimap (\text{payB} \vee (\neg \text{payB} \wedge \text{refundB})))$$

*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.*

$$\text{Seller} = ((\text{payB} \vee \text{refundB}) \multimap \text{signS}) \wedge (\text{signS} \multimap (\text{sellS} \vee (\neg \text{sellS} \wedge \text{refundS})))$$

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:

$$\vdash \text{Buyer} \wedge \text{Seller} \rightarrow \text{signB} \wedge \text{signS} \quad (12)$$

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

$$\vdash \text{Buyer} \wedge \text{Seller} \wedge \neg \text{payB} \rightarrow \text{refundB} \quad (13)$$

$$\vdash \text{Buyer} \wedge \text{Seller} \wedge \neg \text{sellS} \rightarrow \text{refundS} \quad (14)$$

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 `lock` \wedge `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.

$$\text{Buyer} = (\text{send} \vee \text{unlock}) \rightarrow (\text{lock} \wedge \text{offer})$$

$$\text{Seller} = \text{offer} \rightarrow ((\text{lock} \rightarrow \text{send}) \vee \text{abort})$$

$$\text{Bank} = (\text{lock} \wedge \text{abort}) \rightarrow \text{unlock}$$

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

$$\vdash (\text{Buyer} \wedge \text{Seller} \wedge \text{Bank}) \rightarrow (\text{send} \vee (\text{abort} \wedge \text{unlock}))$$

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 $\mathbf{g}_{i,j}$ for “ r_i gives a piece (of kind i) to r_j ”. Since retailers can use their own cutlery, we assume $\mathbf{g}_{i,i}$ to be true. Retailer r_i can start eating whenever $e_i = \bigwedge_j \mathbf{g}_{j,i}$. Instead, he provides the cutlery to others whenever $p_i = \bigwedge_j \mathbf{g}_{i,j}$.

Suppose that r_1 commits to a simple exchange with r_2 : they commit to $\mathbf{g}_{2,1} \rightarrow \mathbf{g}_{1,2}$ and $\mathbf{g}_{1,2} \rightarrow \mathbf{g}_{2,1}$, and the exchange takes place since $\mathbf{g}_{2,1} \wedge \mathbf{g}_{1,2}$ 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 $\mathbf{g}_{2,i} \wedge \mathbf{g}_{i,2}$ for all i . In particular, $\mathbf{g}_{i,2}$ 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 $\mathbf{g}_{2,1} \rightarrow \mathbf{g}_{1,2}$. Instead, the retailer r_1 could commit to a safer contract:

$$e_1 \rightarrow p_1 = \mathbf{g}_{1,1} \wedge \mathbf{g}_{2,1} \wedge \cdots \wedge \mathbf{g}_{n,1} \rightarrow \mathbf{g}_{1,1} \wedge \mathbf{g}_{1,2} \wedge \cdots \wedge \mathbf{g}_{1,n}$$

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 [3]).

$$\vdash \bigwedge_i (e_i \rightarrow p_i) \rightarrow \bigwedge_i e_i$$

3 The Calculus

We now define our calculus of contracting processes.

3.1 Syntax

We use a denumerable set of *names*, ranged over by n, m, \dots , and a denumerable set of *variables*, ranged over by x, y, \dots . 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. That is, distinct names represent distinct concrete objects, each one with its own identity. Instead, distinct variables can be bound to the same object, so they can be fused. When variable fusion happens, our calculus instantiates the involved variables to a specific name, as we will see.

We extend prime formulae of PCL with parameters: $\mathbf{p}(a, b)$. Note that we do not introduce quantifiers in PCL, which then remains a propositional logic. Indeed, the formula $\mathbf{p}(a, b)$ is still atomic from the point of view of the logic. We let letters c, d range over formulae, while letters u, v range over $\{\perp, \neg\}$ -free formulae. The syntax of our calculus follows.

$$\begin{aligned} \pi &::= \tau \mid \text{ask } c \mid \text{tell } u \mid \text{check } c \mid \text{fuse}_x c \mid \text{join}_x c && (\text{prefixes}) \\ P &::= u \mid \sum_{i \in I} \pi_i.P_i \mid (P|P) \mid (a)P \mid X(\vec{a}) && (\text{processes}) \end{aligned}$$

Processes P are mostly standard, and include the active constraint u , the sum of guarded processes $\sum_i \pi_i.P_i$, the parallel composition $P|P$, and the scope delimitation $(a)P$. We use a set of definitions $\{X_i(\vec{x}) \doteq P_i\}_i$ with the provision that each occurrence of X_j in P_k is guarded, i.e. it is behind some prefix.

We write 0 for the empty sum, and we use $+$ to merge sums:

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

Singleton sums are simply written $\pi.P$.

Prefixes π include τ (the standard silent operation as in CCS), as well as *ask*, *tell*, *check* carrying the same meaning as in Concurrent Constraints [33]. The prefix *ask* c causes a process to stop until the formula c is deducible from the context. Dually, the prefix *tell* u augments the context with the formula u . Note that, since we only allow negation-free formulae u here, the context will always be consistent, by Lemma 2. The prefix *check* c checks if c is consistent with the context. The other prefixes *fuse* _{x} c and *join* _{x} c drive the fusion of the variable x . The prefix *join* _{x} c instantiates x to any known name, provided that after the instantiation the formula c is satisfied. The prefix *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. If we use names to model session identifiers, then the difference between *join* _{x} c and *fuse* _{x} c becomes evident: a *join* _{x} c is used to join an already initiated session, while a *fuse* _{x} c is used to initiate a new session.

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:

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

That is, when a variable x is instantiated to a name, the prefixes *fuse* _{x} c and *join* _{x} c can no longer require the fusion of x , so they behave as a plain *ask* c .

3.2 Semantics

We now define the semantics of processes. Henceforth, we consider processes up-to alpha-conversion. The semantics is then given through a two-layered transition system (Fig. 3.2). The bottom layer is an LTS $\xrightarrow{\alpha}$ between processes, which provides a compositional semantics. Actions α are as follows, where C denotes a set of PCL formulae.

$$\alpha ::= \tau \mid C \mid C \vdash c \mid C \vdash_x^F c \mid C \vdash_x^J c \mid C \not\vdash \perp \mid (a)\alpha \quad (\text{actions})$$

The action τ represents an internal move. The action C is an advertisement of a set of active constraints. The action $C \vdash c$ is a *tentative* action, generated by a process attempting to fire an *ask* c prefix. This action also carries a set C as the collection of active constraints discovered so far. Similarly for $C \vdash_x^F c$ and *fuse* _{x} c , for $C \vdash_x^J c$ and *join* _{x} c , as well as for $C \not\vdash \perp$ and *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 [32]. 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 $(\vec{a} \cup \vec{b})$.

The first three lines of the rules in Fig. 3.2 handle the base cases of our semantics. The rule TAU allows a τ prefix to fire. The rules ASK, CHECK, FUSE, and JOIN simply generate the corresponding 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.

$$\begin{array}{c}
\tau.P \xrightarrow{\tau} P \text{ [TAU]} \quad \text{ask } c.P \xrightarrow{\emptyset \vdash c} P \text{ [ASK]} \quad \text{tell } u.P \xrightarrow{\tau} u|P \text{ [TELL]} \\
\text{check } c.P \xrightarrow{\{c\} \not\vdash \perp} P \text{ [CHECK]} \quad \text{fuse}_x c.P \xrightarrow{\emptyset \vdash_x^F c} P \text{ [FUSE]} \quad \text{join}_x c.P \xrightarrow{\emptyset \vdash_x^J c} P \text{ [JOIN]} \\
u \xrightarrow{\{u\}} u \text{ [CONSTR]} \quad \sum_i \pi_i.P_i \xrightarrow{\emptyset} \sum_i \pi_i.P_i \text{ [IDLESUM]} \\
\frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})C'} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(CUC')} P'|Q'} \text{ [PARCONSTR]} \quad \frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})(C' \vdash c)} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(CUC' \vdash c)} P'|Q'} \text{ [PARASK]} \\
\frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})(C' \vdash_x^F c)} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(CUC' \vdash_x^F c)} P'|Q'} \text{ [PARFUSE]} \quad \frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})(C' \vdash_x^J c)} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(CUC' \vdash_x^J c)} P'|Q'} \text{ [PARJOIN]} \\
\frac{P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})(C' \not\vdash \perp)} Q'}{P|Q \xrightarrow{(\vec{a}\vec{b})(CUC' \not\vdash \perp)} P'|Q'} \text{ [PARCHECK]} \quad \frac{P \xrightarrow{\tau} P'}{P|Q' \xrightarrow{\tau} P'|Q'} \text{ [PARTAU]} \\
\frac{\pi_j.P_j \xrightarrow{\alpha} P'}{\sum_i \pi_i.P_i \xrightarrow{\alpha} P'} \text{ [SUM]} \quad \frac{P\{\vec{a}/\vec{x}\} \xrightarrow{\alpha} P'}{X(\vec{a}) \xrightarrow{\alpha} P'} \text{ if } X(\vec{x}) \doteq P \text{ [DEF]} \\
\frac{P \xrightarrow{\alpha} P'}{(a)P \xrightarrow{\alpha} (a)P'} \text{ if } a \notin \alpha \text{ [DEL]} \quad \frac{P \xrightarrow{\alpha} P'}{(a)P \xrightarrow{(a)\alpha} P'} \text{ [OPEN]} \\
\frac{P \xrightarrow{(\vec{a})(C' \vdash c)} P'}{P \xrightarrow{\tau} (\vec{a})P'} \text{ if } C \vdash c \text{ [CLOSEASK]} \quad \frac{P \xrightarrow{(xn\vec{a})(C' \vdash_x^J c)} P'}{P \xrightarrow{\tau} (n\vec{a})P'\sigma} \text{ if } \begin{array}{l} C\sigma \vdash c\sigma, \\ \sigma = \{n/x\} \end{array} \text{ [CLOSEJOIN]} \\
\frac{P \xrightarrow{(x\vec{y}\vec{a})(C' \vdash_x c)} P'}{P \xrightarrow{\tau} (n\vec{a})P'\{n/x\vec{y}\}} \text{ if } \begin{array}{l} C \vdash_{x\vec{y}}^n c, \\ n \text{ fresh} \end{array} \text{ [CLOSEFUSE]} \\
\frac{P \xrightarrow{\tau} P'}{P \mapsto P'} \text{ [TOPTAU]} \quad \frac{P \xrightarrow{(\vec{a})(C \not\vdash \perp)} P'}{P \mapsto (\vec{a})P'} \text{ if } C \not\vdash \perp \text{ [TOPCHECK]}
\end{array}$$

Figure 1: The transition system for our calculus. Symmetric rules for $+$, $|$ are omitted. The rules 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 next three lines of rules handle parallel composition. 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 SUM , DEF are quite standard: they handle external choice and (possibly recursive) definitions, respectively. The rules DEL , OPEN handle delimitation. As usual, when a is not mentioned in an action, we can propagate the

action across a delimitation (a) using DEL . The rule OPEN instead allows for scope extrusion. Note that OPEN has no side conditions: 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 (\bar{a}) can be brought back to the process level, stopping the scope extrusion. The rule CLOSEJOIN is similar to CLOSEASK , except it also allows the variable x to be instantiated with 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 also causes the instantiation of variables to names. Unlike CLOSEJOIN , however, it cannot instantiate x to an existing name; rather, the name n here is a completely 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 $C \vdash_{x\vec{y}}^n c$, to be defined in a while (Def. 5).

Summing up, the LTS given by $\xrightarrow{\alpha}$ allows for the generation of tentative actions for ask , join and fuse (rules ASK, JOIN, FUSE), which can then be converted to $\xrightarrow{\tau}$ whenever enough constraints are discovered (rules CLOSE^*). Then, the τ action can be propagated towards the top level (rule PARTAU). A prefix $\text{check } c$, instead, 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 \succ , layered over the $\xrightarrow{\alpha}$ relation. The reduction \succ 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 \succ is only applied to the top level.

Definition 5 (Local Minimal Fusion) *The relation $C \vdash_{\vec{z}}^n c$ holds whenever:*

$$\exists C' \subseteq C. (C' \{n/\vec{z}\} \vdash c \{n/\vec{z}\} \wedge \nexists \vec{w} \subsetneq \vec{z}. C' \{n/\vec{w}\} \vdash c \{n/\vec{w}\})$$

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.

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:

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

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 $\mathbf{p}(x)$ we must exploit the implication,

$$\begin{aligned}
P|0 &\equiv P & P|Q &\equiv Q|P & P|(Q|R) &\equiv (P|Q)|R \\
P+0 &\equiv P & P+Q &\equiv Q+P & P+(Q+R) &\equiv (P+Q)+R \\
(a)(P|Q) &\equiv P|(a)Q & \text{if } a \notin \text{free}(P) & & (a)(b)P &\equiv (b)(a)Q
\end{aligned}$$

Figure 2: Structural equivalence.

so fusing x with y is mandatory. Further, the hypothesis $\mathfrak{q}(z) \vee \mathfrak{s}$ can only be discharged through $\mathfrak{q}(y)$, 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 $\mathfrak{q}(z) \vee \mathfrak{s}$ can now be discharged through \mathfrak{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:

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

where, with some abuse of notation, C' stands for the parallel composition of its constraints. In P , when dealing with the process $(x)(y)(z)(\text{fuse}_x c.P \mid C')$, 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 $C \vdash_{\vec{z}}^n c$ 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 $C \vdash_{xyz}^n c$ and $C \vdash_{xy}^n c$ 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.*

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

3.3 Examples

We illustrate the semantics through some examples.

Example 4 (Handshaking) *Recall the handshaking example from Sect. 1.1:*

$$\begin{aligned}
\text{Alice} &= (x) (\text{tell } \mathfrak{b}(x) \rightarrow \mathfrak{a}(x). \text{fuse}_x \mathfrak{a}(x). \text{lendAirplane}) \\
\text{Bob} &= (y) (\text{tell } \mathfrak{a}(y) \rightarrow \mathfrak{b}(y). \text{fuse}_y \mathfrak{b}(y). \text{lendBike})
\end{aligned}$$

A possible trace is the following:

$$\begin{aligned}
& \text{Alice} \mid \text{Bob} \\
& \xrightarrow{\tau} (x) (\text{b}(x) \rightarrow \text{a}(x) \mid \text{fuse}_x \text{a}(x). \text{lendAirplane}) \mid \text{Bob} \\
& \xrightarrow{\tau} (x) (\text{b}(x) \rightarrow \text{a}(x) \mid \text{fuse}_x \text{a}(x). \text{lendAirplane}) \mid \\
& \quad (y) (\text{a}(y) \rightarrow \text{b}(y) \mid \text{fuse}_y \text{b}(y). \text{lendBike}) \\
& \xrightarrow{\tau} (n) (\text{b}(n) \rightarrow \text{a}(n) \mid \text{lendAirplane}\{n/x\} \mid \\
& \quad \text{a}(n) \rightarrow \text{b}(n) \mid \text{ask b}(n). \text{lendBike}\{n/y\}) \\
& \xrightarrow{\tau} (n) (\text{b}(n) \rightarrow \text{a}(n) \mid \text{lendAirplane}\{n/x\} \mid \\
& \quad \text{a}(n) \rightarrow \text{b}(n) \mid \text{lendBike}\{n/y\})
\end{aligned}$$

In step one, we use TELL, PARTAU, DEL to fire the prefix $\text{tell b}(x) \rightarrow \text{a}(x)$. Dually, in step two, we fire the prefix $\text{tell a}(y) \rightarrow \text{b}(y)$ through the same rules. Step three is the crucial one. There, the prefix $\text{fuse}_x \text{a}(x)$ is fired through rule FUSE . Through rules CONSTR, PARFUSE , we discover the active constraint $c_a = \text{b}(x) \rightarrow \text{a}(x)$. We use rule OPEN to obtain the action $(x)\{c_a\} \vdash \text{a}(x)$ for the Alice part. For the Bob part, we use rule CONSTR to discover $c_b = \text{a}(y) \rightarrow \text{b}(y)$, 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 $(x, y)\{c_a, c_b\} \vdash \text{a}(x)$. 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 $\{c_a, c_b\} \vdash_{x,y}^n \text{a}(x)$. Note that the instantiation causes $\text{fuse}_y \text{b}(y)$ to transform into $\text{ask b}(n)$, so that it can be fired in the last step.

To better understand the role of contractual implication in this example, consider the following alternative handshaking which makes no use of \rightarrow .

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

It is straightforward to see that $P' = \text{Alice}' \mid \text{Bob}'$ has analogous behaviour to $P = \text{Alice} \mid \text{Bob}$, since the handshaking is still performed. However, the processes P and P' behave quite differently in the presence of a third kid, Carl.

$$\text{Carl} = (z) \text{fuse}_z \text{a}(z). 0$$

The system $P' \mid \text{Carl}$ allows Carl to fire his own prefix, by fusing x with z . So, Carl will be allowed to play with Alice's airplane, even if Alice is not given anything back. Indeed, after the instantiation, Alice will be stuck on an $\text{ask b}(n)$ prefix. Technically, this happens because $\{\text{a}(x)\} \vdash_{x,z}^n \text{a}(z)$ holds. In this case, Bob will be stuck as well: he can not play with Alice's airplane since Carl took it. By contrast, the system $P \mid \text{Carl}$ 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 provided. Since Carl provides no bike, he is not allowed to use the airplane. Instead, Bob can successfully exchange his bike with Alice's airplane.

Example 5 (Dining retailers) We reuse the formulae of Ex. 3, augmenting each prime formula with a parameter x .

$$\begin{aligned}
e_i(x) &= \wedge_j \mathbf{g}_{j,i}(x) & p_i(x) &= \wedge_j \mathbf{g}_{i,j}(x) \\
R_i &= (x) (\text{tell } e_i(x) \rightarrow p_i(x). \text{fuse}_x p_i(x). \parallel_j \text{give}_{i,j})
\end{aligned}$$

As discussed in Ex. 3, we have that $\wedge_i e_i(n) \rightarrow p_i(n)$ is enough to entail $p_i(n)$ for all i . So, all the fuse prefixes are fired, and all the $\text{give}_{i,j}$ 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 6 (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.

$$\begin{aligned} s(x) &= \text{order}(x) \wedge (\text{pay}(x) \vee \text{insurance}(x)) \rightarrow \text{ship}(x) \\ S &= (x)\text{tell } s(x).\text{fuse}_x \text{ship}(x).(S \mid \text{doShip}(x)) \end{aligned}$$

The process S above is recursive, so that many orders can be 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.

$$\begin{aligned} i(x) &= \text{premium}(x) \rightarrow \text{insurance}(x) \\ I &= (x)\text{tell } i(x).\text{fuse}_x \text{insurance}(x). \\ & \quad (I \mid \tau.\text{check } \neg\text{pay}(x).(\text{refundS}(x) \mid \text{debtCollect}(x))) \end{aligned}$$

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:

$$\begin{aligned} b_0(x) &= \text{ship}(x) \rightarrow \text{order}(x) \wedge \text{pay}(x) \\ B_0 &= (x)\text{tell } b(x).\text{receive}(x) \end{aligned}$$

A buyer can also pay later, if provides insurance:

$$\begin{aligned} b_1(x) &= \text{ship}(x) \rightarrow \text{order}(x) \wedge \text{premium}(x) \\ B_1 &= (x)\text{tell } b(x).(\text{receive}(x) \mid \tau.\text{tell } \text{pay}(x)) \end{aligned}$$

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

$$b_2(x) = \text{order}(x) \wedge \text{pay}(x) \quad B_2 = B_0$$

Finally, interaction with a malicious buyer is possible:

$$b_3(x) = \text{order}(x) \wedge \text{premium}(x) \quad B_3 = B_0$$

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 7 (Automated Judge) Consider an online market, where buyers and sellers trade items. The contract of a buyer is to pay for an item, provided that the seller promises to send it; dually, the contract of a seller is to send an item, provided that the buyer pays. A buyer first issues her contract, then waits until discovering she has to pay, and eventually proceeds with the process *CheckOut*. At this point, the buyer may either abort the transaction (process *NoPay*), or actually pay the item, by issuing the constraint $\text{paid}(x)$. After the item has been paid, the buyer may wait for the item to be sent ($\text{ask sent}(x)$), or possibly open a dispute with the seller ($\text{tell dispute}(x)$).

$$\begin{aligned} \text{Buyer} &= (x) (\text{tell send}(x) \rightarrow \text{pay}(x). \text{fuse}_x \text{pay}(x). \text{CheckOut}) \\ \text{CheckOut} &= \tau. \text{NoPay} + \tau. \text{tell paid}(x). (\tau. \text{tell dispute}(x) + \text{ask sent}(x)) \end{aligned}$$

The behaviour of the seller is dual: issue the contract, wait until she has to send, and then proceed with the process *Ship*. There, either choose not to send, or send the item and then wait for the payment or open a dispute.

$$\begin{aligned} \text{Seller} &= (y) (\text{tell pay}(y) \rightarrow \text{send}(y). \text{fuse}_y \text{send}(y). \text{Ship}) \\ \text{Ship} &= \tau. \text{NoSend} + \tau. \text{tell sent}(y). (\tau. \text{tell dispute}(y) + \text{ask pay}(y)) \end{aligned}$$

To automatically resolve disputes, the process *Judge* may 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 join_z primitive, which binds the variable z to the name of the session established between buyer and seller. If the obligation $\text{pay}(z)$ is found, but the item has not been actually paid (i.e. $\text{check } \neg \text{paid}(z)$ passes), then the buyer is convicted (modelled by $\text{jailBuyer}(z)$, not further detailed). Similarly, if the obligation $\text{send}(z)$ has not been supported by a corresponding $\text{sent}(z)$, then the seller is convicted.

$$\begin{aligned} \text{Judge} &= (z) (\text{join}_z (\text{pay}(z) \wedge \text{dispute}(z)). \text{check } \neg \text{paid}(z). \text{jailBuyer}(z) \mid \\ &\quad \text{join}_z (\text{send}(z) \wedge \text{dispute}(z)). \text{check } \neg \text{sent}(z). \text{jailSeller}(z)) \end{aligned}$$

A possible trace is the following:

$$\begin{aligned}
& \underline{Buyer} \mid \underline{Seller} \mid \underline{Judge} \\
& \xrightarrow{\tau} (x) (\text{send}(x) \rightarrow \text{pay}(x) \mid \text{fuse}_x \text{pay}(x). \text{CheckOut}) \mid \underline{Seller} \mid \underline{Judge} \\
& \xrightarrow{\tau} (x) (\text{send}(x) \rightarrow \text{pay}(x) \mid \underline{\text{fuse}_x \text{pay}(x)}. \text{CheckOut}) \mid \\
& \quad (y) (\text{pay}(y) \rightarrow \text{send}(y) \mid \text{fuse}_y \text{send}(y). \text{Ship}) \mid \underline{Judge} \\
& \xrightarrow{\tau} (n) (\text{send}(n) \rightarrow \text{pay}(n) \mid \text{CheckOut}\{n/x\} \mid \\
& \quad \text{pay}(n) \rightarrow \text{send}(n) \mid \underline{\text{ask send}(n)}. \text{Ship}\{n/y\}) \mid \underline{Judge} \\
& \xrightarrow{\tau} (n) (\text{send}(n) \rightarrow \text{pay}(n) \mid \text{CheckOut}\{n/x\} \mid \text{pay}(n) \rightarrow \text{send}(n) \mid \underline{\text{Ship}\{n/y\}}) \mid \underline{Judge} \\
& \xrightarrow{\tau} (n) (\text{send}(n) \rightarrow \text{pay}(n) \mid \text{CheckOut}\{n/x\} \mid \text{pay}(n) \rightarrow \text{send}(n) \mid \text{NoSend}) \mid \underline{Judge} \\
& \xrightarrow{\tau} (n) (\text{send}(n) \rightarrow \text{pay}(n) \mid \underline{\text{tell paid}(n)}. (\tau. \text{tell dispute}(n) + \text{ask sent}(n)) \mid \\
& \quad \text{pay}(n) \rightarrow \text{send}(n) \mid \text{NoSend}) \mid \underline{Judge} \\
& \xrightarrow{\tau} (n) (\text{send}(n) \rightarrow \text{pay}(n) \mid \text{paid}(n) \mid \underline{\tau. \text{tell dispute}(n)} + \text{ask sent}(n) \mid \\
& \quad \text{pay}(n) \rightarrow \text{send}(n) \mid \text{NoSend}) \mid \underline{Judge} \\
& \xrightarrow{\tau} (n) (\text{send}(n) \rightarrow \text{pay}(n) \mid \text{paid}(n) \mid \underline{\text{tell dispute}(n)} + \text{ask sent}(n) \mid \\
& \quad \text{pay}(n) \rightarrow \text{send}(n) \mid \text{NoSend}) \mid \underline{Judge} \\
& \xrightarrow{\tau} (n) (\text{send}(n) \rightarrow \text{pay}(n) \mid \text{paid}(n) \mid \text{dispute}(n) \mid \text{pay}(n) \rightarrow \text{send}(n) \mid \text{NoSend}) \mid \underline{Judge} \\
& \xrightarrow{\tau} (n) (\text{send}(n) \rightarrow \text{pay}(n) \mid \text{paid}(n) \mid \text{dispute}(n) \mid \text{pay}(n) \rightarrow \text{send}(n) \mid \text{NoSend} \mid \\
& \quad \underline{\text{check } \neg \text{sent}(n)}. \text{jailSeller}(n) \mid \dots) \\
& \xrightarrow{\tau} (n) (\text{jailSeller}(n) \mid \dots)
\end{aligned}$$

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 line, where they can pick anything they want. After the meal is over, they are no longer allowed to return to the buffet.

We model this scenario through the following processes:

$$\text{Buffet} = (x) (\text{pasta}(x) \mid \text{chicken}(x) \mid \text{cheese}(x) \mid \text{fruit}(x) \mid \text{cake}(x))$$

$$\text{Bob} = (x) \text{fuse}_x \text{pasta}(x) \wedge \text{chicken}(x). \text{SatiatedB}$$

$$\text{Carl} = (x) \text{fuse}_x \text{pasta}(x). \text{fuse}_x \text{chicken}(x). \text{SatiatedC}$$

The Buffet above can interact with either Bob or Carl, and make them satiated. 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:

$$\text{Buffet} \mid \text{Carl} \rightarrow^* \text{SatiatedC} \mid P$$

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,

$$\bigoplus_{i \in I} p_i = (o)(z)(z_i)_{i \in I} (r(o, z) \mid \parallel_{i \in I} r(o, z_i) \rightarrow p_i)$$

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 $r(o, z_i)$. 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

$$(o)(m) \left((z_i)_{i \in I \setminus J} (\|_{i \in I \setminus J} r(o, z_i) \rightarrow p_i) \parallel_{i \in J} (r(o, m) | r(o, m) \rightarrow p_i) \right) | Q'$$

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 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:

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

The new specification actually enforces the desired buffet policy:

$$\text{Buffet}' | \text{Carl} \not\rightarrow^* \text{SatiatedC} | P$$

4 Expressive power

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.

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

Each $\text{fuse}_x p(n, x)$ instantiates a variable x such that $p(n, x)$ holds. Of course, the same x can not 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.

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

Process $\text{New}(n, v)$ initializes the cell having name n and initial value v . Process $\text{Get}(n, y)$ recovers v by fusing it with y : the procedure is destructive so the cell is re-created. Process $\text{Set}(n, v)$ 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 [22]. For illustration, we only consider \mathbf{p} -tagged pairs here.

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

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 \mathbf{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 \mathbf{fuse} prefix. Then, we use \mathbf{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 can easily encode the synchronous π -calculus in our calculus. Our encoding, shown below, preserves parallel composition, and maps name restriction to name delimitation, as one might desire.

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

where x, y are fresh. 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 $\mathbf{msg}(y, z)$.

The encoding above satisfies the requirements of [24]. 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_π for reduction in π ,

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

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

$$\begin{aligned}
[(\nu m)(n\langle m \rangle.P|n(z).Q)] &\mapsto^* \\
(m)(o)(\mathbf{msg}(o, m)|[P]|n(n, o)|[Q]\{m/y\}) &\sim (m)[P|Q\{m/y\}]
\end{aligned}$$

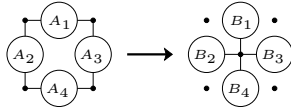
since the name o is fresh and the constraints $\text{msg}(o, m), \text{in}(n, o)$ do not affect the behaviour of P, Q . To see this, consider the inputs and outputs occurring in P, Q . Indeed, in the encoding of inputs, the fuse_x prefix will instantiate x to a fresh name, hence not with o . On the other 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 $\text{msg}(o, m)$ 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 an operator \oplus on contracts is defined 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 [17]. Before dealing with the general case, we introduce our encoding through a simple example.

Example 9 Consider the following rewriting rule:



Above, $A_1 \dots A_4$ 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 $B_1 \dots B_4$, 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 $A_1 \dots A_4$. 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 and \ominus , respectively.

$$a_i(n, m, x) = f_{i \oplus 1}(x, m) \wedge s_{i \ominus 1}(x, n) \multimap f_i(x, n) \wedge s_i(x, m) \quad (15)$$

An intuitive interpretation of f, s is as follows: $f_i(x, n)$ states that n is the first name of some process $A_i(n, -)$ which is about to apply the rule. Similarly for $s_i(x, m)$ 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.

$$A_i(n, m) = (x)\text{tell } a_i(n, m, x).\text{fuse}_x f_i(x, n) \wedge s_i(x, m). B_i(x)$$

Our PCL logic enables the wanted transition:

$$P = \parallel_i A_i(n_i, n_{i\oplus 1}) \rightsquigarrow^* (m) \parallel_i B_i(m)$$

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 6 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 $\text{tag}(e)$ and an ordered tuple of vertices (e_1, \dots, e_k) where $e_j \in V_G$. The tag $\text{tag}(e)$ uniquely determines the arity k .

Definition 7 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. 8 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 8 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 $e, e' \in E_{G_i} \cup E_{H_i}$
- $\text{tag}(e) = \text{tag}(\sigma(e))$ 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 $J \rightarrow K$ holds iff, for some embedding σ , we have

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

Note that the assumption $V_{G_i} \subseteq V_{H_i}$ of Def. 7 ensures that $V_J \subseteq V_K$, so no dangling hyperedges are created through rewriting.

We now proceed to encode graph rewriting in our calculus. To simplify our encoding, we make a mild assumption: we require each G_i to be a *connected* hypergraph. Then, encoding a generic hypergraph J is performed in a compositional way: we assign a unique name n to each vertex in V_J , and then build a parallel composition of processes $A_{\text{tag}(e)}(\vec{n})$, one for each hyperedge e in E_J , where $\vec{n} = (n_1, \dots, n_k)$ identifies the adjacent vertices. Note that since the behaviour of an hyperedge e depends on its tag, only, we index A with $t = \text{tag}(e)$.

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 $\mathbf{p}_{e,h}(x, n)$ 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 $\mathbf{p}_{e,h}(x, n)$ in the definition of A_t . The process $A_t(\vec{n})$ promises $\mathbf{p}_{e,1}(x, n_1), \dots, \mathbf{p}_{e,k}(x, n_k)$ (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:

$$a_e(x, \vec{n}) = \bigwedge_{\substack{1 \leq h \leq k \\ \bar{e} \in E_{G_i} \\ \bar{e}_h = e_h}} \mathbf{p}_{\bar{e},h}(x, n_h) \rightsquigarrow \bigwedge_{1 \leq h \leq k} \mathbf{p}_{e,h}(x, n_h) \quad (16)$$

Note that in the previous example 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 \mathbf{f}_i and \mathbf{s}_i in (15) are rendered as $\mathbf{p}_{e_i,1}$ and $\mathbf{p}_{e_i,2}$ in (16). Also the operators $\oplus 1$ and $\ominus 1$, used in (15) to choose neighbours, are generalized in (16) through the condition $\bar{e}_h = e_h$.

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 $\mathbf{p}_{e,h}(x, n_h)$ 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 $\text{tag}(e1) = \text{tag}(\bar{e}1) = t1$ and $\text{tag}(\bar{e}2) = \text{tag}(e2) = t2$. Each process thus advertises two contracts, for instance:

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

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 perform the fusion, say the first one:

$$(m)(a_{e1}(m, n) | a_{\bar{e}1}(m, n) | \text{Rewrite}_{e1} | a_{e2}(m, n) | a_{\bar{e}2}(m, n) | \text{Fusion}_{t2} \{m/x\})$$

Above m is a fresh name. Note however that the process $\text{Fusion}_{t2} \{m/x\}$ can still proceed with the *other* rewriting, since the substitution above can not disable a prefix which was enabled before. So, we can end up with $\text{Rewrite}_{\bar{e}2}$, 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.

$$A_t(\vec{n}) = (x) \left(\bigoplus_{\text{tag}(e)=t} a_e(x, \vec{n}) \mid \sum_{\text{tag}(e)=t} \text{fuse}_x \bigwedge_{1 \leq h \leq k} p_{e,n}(x, n_h). B_e(x, \vec{n}) \right)$$

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: $A_t(\vec{n}) \mid \dots \mid A_{t'}(\vec{n}')$. 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, \dots, 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 $p_{e,n}$. 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. $\text{tell}_{\text{vert}_h}(m, n)$, and retrieved through the corresponding $\text{join}_y \text{vert}_h(m, y)$. 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.

Our encoding $[-]$ satisfies *operational correspondence*:

$$\begin{aligned} J \rightarrow^* K &\implies [J] \mapsto^* \sim [K] \\ [J] \mapsto^* P &\implies \exists K. J \rightarrow^* K \wedge P \mapsto^* \sim [K] \end{aligned}$$

where \sim is \mapsto -bisimilarity. The encoding also preserves *termination*, *divergence* and *success*. Hence, it satisfies the criteria of [24], when the requirements on names are adapted to vertices.

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 [8, 9, 10, 27]. For instance, in [9] 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 paid 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:

$$Buyer = (\text{ship} \vee \text{refund}) \rightarrow \text{pay} \qquad Seller = \text{pay} \rightarrow (\text{ship} \vee \text{refund})$$

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

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

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 in the contracts in [9] 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. [11, 4, 7, 5]. 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 [19]. 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 [7]. 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:

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

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:

$$\textit{Buyer} = (\textit{unavailable} \rightarrow (\textit{refund} \wedge \textit{15\%discount})) \rightarrow \textit{pay}$$

Several logics for modelling contracts have been introduced over the years, taking inspiration and extending e.g. classical [15], modal [14, 1, 20], intuitionistic [2], deontic [31, 21], default [23] and defeasible logics [25]. 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 [3] 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:

$$\begin{aligned} \text{Seller} = \forall \text{item}, \text{cust}, \text{addr} : \\ \text{pay}(\text{item}, \text{cust}, \text{addr}) \rightarrow \text{ship}(\text{item}, \text{addr}) \end{aligned} \quad (17)$$

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.

$$\text{Bob} = \text{ship}(\text{drill}, \text{bobAddress}) \rightarrow \text{pay}(\text{drill}, \text{Bob}, \text{bobAddress})$$

Joining the two contracts above will yield the intended agreement:

$$\text{Seller} \wedge \text{Bob} \rightarrow \text{pay}(\text{drill}, \text{Bob}, \text{bobAddress}) \wedge \text{ship}(\text{drill}, \text{bobAddress})$$

Principals. We will consider extending our logic with a *says* modality, similarly to [20]. This will enable us to write, e.g. *Alice says* ($\mathbf{b} \rightarrow \mathbf{a}$) to represent the fact that Alice has issued a contract where she promises to lend her airplane, provided that she borrows a bike (note that the binding between principals and contracts is represented *outside* the logic PCL). Contract agreements shall then come in a richer flavour, because of the binding between the contracting parties and their inferred duties, which is now revealed. Back to our first example of Sect. 1.1, one could expect an handshaking of the following form:

$$\text{Alice says } (\mathbf{b} \rightarrow \mathbf{a}) \wedge \text{Bob says } (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow \text{Alice says } \mathbf{a} \wedge \text{Bob says } \mathbf{b}$$

from which it is clear that the duty of Alice is that of lending her airplane, and the duty of Bob is that of lending his bike. This additional information can be exploited by a third party (a sort of “automated” judge) which has to investigate the responsibilities of various parties, in the unfortunate case that a contract is not respected. For instance, if our automated judge is given the evidence that Alice's airplane has never been lent to Bob, from the above he will infer that $(\text{Alice says } \mathbf{a}) \wedge \neg \mathbf{a}$. From this, our judge will be able to infer that Alice has not respected her contract (and possibly punish her), which is modelled by the formula *Alice says* \perp .

This extension will have some additional benefits, especially when putting it at work in insecure environments populated by attackers. Indeed, an attacker could maliciously issue a “fake” contract, where he makes a promise that he cannot actually implement, e.g. because the promised duty can only be performed by another party. For instance, consider again the contract (17). Assume that an attacker wants to maliciously exploit the seller contract, in order to receive a free item, and make the unaware customer Bob pay for it. To do that, the attacker issues the following contract:

$$\text{FakeBob} = \text{ship}(10\text{Kdiamond}, \text{fakeAddress}) \rightarrow \text{pay}(10\text{Kdiamond}, \text{Bob}, \text{fakeAddress})$$

Joining the seller and the attacker contracts will then cause an unwelcome situation for Bob, who is due to pay for a 10K diamond, which will be shipped to

the attacker's address:

$$Seller \wedge FakeBob \rightarrow \text{pay}(10K\text{diamond}, \text{Bob}, \text{fakeAddress}) \wedge \text{ship}(10K\text{diamond}, \text{fakeAddress})$$

To cope with this situation, we could require that each contract p is signed by the principal A who issues it, i.e. it has the form $A \text{ says } p$. Revisiting our example with this trick, in the safe case that Bob himself has ordered the item, we would expect to deduce:

$$Seller \wedge Bob \rightarrow Bob \text{ says } \text{pay}(\text{drill}, \text{Bob}, \text{bobAddress})$$

In this case, we have a successful transaction, because Bob is stating that he will pay for his drill. Instead, joining the seller and the attacker contracts produces:

$$Seller \wedge FakeBob \rightarrow FakeBob \text{ says } \text{pay}(10K\text{diamond}, \text{Bob}, \text{fakeAddress})$$

Now, it is easy to realize that someone has attempted a fraud, because the principal who has signed the contract (FakeBob) is different from that who is due to pay (Bob).

To exploit this extension also in our calculus, it suffices e.g. to constrain the $\text{tell } u$ prefix to formulae u of the form $A \text{ says } u'$, where A is the name of the principal running the prefix.

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 $\text{p}(t)$ tells the point in time where the “event” p occurs:

$$Seller(t) = \forall t' : (\text{pay}(t) \wedge \text{return}(t') \wedge t' < t + 10) \rightarrow \exists t'' < t' + 21 : \text{refund}(t'')$$

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:

$$Buyer(t) = (\forall t' : \text{return}(t') \wedge t' < t + 10 \rightarrow \exists t'' < t' + 21 : \text{refund}(t'')) \rightarrow \text{pay}(t)$$

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.

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

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 [18], to more recent approaches on temporal extensions of authorization logics like [16].

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 [13] 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 [26], 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 [37].

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 [3]. 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 logic for contracts. Technical Report DISI-09-034, DISI - Università di Trento, 2009.
- [4] Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calculus for long running transactions. In *Proc. FMOODS*, 2003.
- [5] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. POPL*, 2005.
- [6] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *Proc. ESOP*, 2007.
- [7] Michael J. Butler, C. A. R. Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, 2004.
- [8] Samuele Carpineti and Cosimo Laneve. A basic contract language for web services. In *Proc. ESOP*, 2006.
- [9] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.
- [10] Giuseppe Castagna and Luca Padovani. Contracts for mobile processes. In *Proc. CONCUR*, volume 5710 of *Lecture Notes in Computer Science*. Springer, 2009.
- [11] 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.
- [12] 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.
- [13] Ricardo Corin, Pierre-Malo Deniérou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5):573–636, 2008.
- [14] Aspassia Daskalopulu and Tom Maibaum. Towards electronic contract performance. In *Proc. 12th International Workshop on Database and Expert Systems Applications*, 2001.
- [15] Hasan Davulcu, Michael Kifer, and I.V. Ramakrishnan. CTR-S: A logic for specifying contracts in semantic web services. In *Proc. WWW04*, 2004.

- [16] Henry DeYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time. In *Proc. CSF*, 2008.
- [17] 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.
- [18] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.
- [19] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD Conference*, 1987.
- [20] Deepak Garg and Martín Abadi. A modal deconstruction of access control logics. In *Proc. FoSSaCS*, pages 216–230, 2008.
- [21] 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.
- [22] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
- [23] Georgios K. Giannikis and Aspasia Daskalopulu. The representation of e-contracts as default theories. In *New Trends in Applied Artificial Intelligence*, 2007.
- [24] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. In *Proc. CONCUR*, 2008.
- [25] Guido Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14(2-3), 2005.
- [26] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990.
- [27] Luca Padovani. Contract-based discovery and adaptation of web services. In *SFM*, 2009.
- [28] Joachim Parrow and Björn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *LICS*, 1998.
- [29] The PCL web site. <http://www.disi.unitn.it/~zunino/PCL>.
- [30] Frank Pfenning. Structural cut elimination - I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, 2000.
- [31] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *Proc. FMOODS*, 2007.
- [32] Davide Sangiorgi and David Walker. *The π -calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

- [33] Vijay Saraswat, Prakash Panangaden, and Martin Rinard. Semantic foundations of concurrent constraint programming. In *Proc. POPL*, pages 333–352. ACM, 1991.
- [34] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.
- [35] Anne Troelstra and Dirk van Dalen. *Constructivism in Mathematics, vol. 1*. North-Holland, 1988.
- [36] 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.
- [37] Jianying Zhou. *Non-repudiation in Electronic Commerce*. Artech House, 2001.