Flat Committed Join in Join [1]

Roberto Bruni, Hernán Melgratti and Ugo Montanari

Dipartimento di Informatica, Università di Pisa, I-56127 Pisa, Italy,
{bruni,melgratt,ugo}@di.unipi.it

Abstract

Committed Join (cJoin) is an extension of Join with high-level primitives for programming dynamic nested negotiations with compensations. In this paper we show that *flat* cJoin processes (i.e. processes without sub-negotiations) can be encoded in the ordinary Join calculus by exploiting a distributed two-phase commit protocol. In particular, we first define a type system that singles out flat processes and prove subject reduction for it. Then, we show that all flat cJoin processes can be written in an equivalent canonical form, where a few elementary definition patterns are used. Finally, we show that canonical flat processes can be implemented in Join. It is worth noting that negotiation primitives are encoded as fully distributed agreements between all participants, thus avoiding a centralized coordinator.

1 Introduction

Recently, in the area of formal languages, there is a renewed interest from both Academic and Industrial research concerning the design of orchestration primitives for programming largely distributed and long-running decision processes [3,8,2,13]. The increasing number of applications in the area of e-commerce, web services choreography and orchestration patterns demands a rigorous mathematical presentation of such languages, to support formal analysis and verification.

Committed Join (cJoin) [5] is an extension of the Join calculus with primitives for handling distributed *negotiations* (also called *contracts*). Roughly, negotiations are processes that execute in a controlled environment until completion, when they commit and make their results observable to the rest of the system. Additionally, they can be explicitly aborted, in which case, suitable compensation programs can be activated to resume a locally consistent state. A distinctive feature of cJoin is that several negotiations can be merged during their execution into a larger one. This occurs when two or more participants to different negotiations communicate through special ports, called *merge names*. Interacting negotiations are bound together, and

[1] Research supported by the MSR Cambridge Project NAPI, by the FET-GC Project IST-2001-32747 AGILE, by the MIUR Project COFIN 2001013518 COMETA, and by the MURST-CNR 1999 Project, *Software Architectures on Cooperative WAN*.

thus they will jointly reach the same decision, i.e. if one of them eventually commits (resp. aborts) all will eventually commit (resp. abort). This is particularly interesting for designing multi-party negotiations, where independent participants can provide transactional services making explicit the ways in which parties can interact, and where the actual structure of a negotiation is discovered at runtime.

The approach of cJoin contrasts with approaches such as [3], where business processes are described as graphs that spawn across organization boundaries requiring all participants to be known statically. Moreover, partners cannot hide interactions with third parties that can influence the final decision.

Crucial points about the implementation of cJoin are: (1) the commit of interacting negotiations as a global decision, and (2) the number of participants and their identities are not known statically. We show that, for a significant fragment of cJoin, global decisions can be implemented in a fully distributed way by using the *distributed two phase commit* protocol (D2PC) proposed in [4] for implementing *zero-safe nets* [6] (a transactional extension of Petri nets). Note that the Join code written for the D2PC in the case of zero-safe nets can be imported and reused with minor modifications in the encoding of cJoin, giving evidence of its generality.

cJoin is much more expressive than zero-safe nets, as it retains the full expressive power of ordinary Join. The presence of compensations and of merge names increases the level of complexity of the encoding, making it far from trivial. Indeed, we restrict ourselves to consider cJoin processes that can be typed as *flat*, meaning that they will never generate nested negotiations. We show that flat processes form a sub-calculus of cJoin by proving the subject reduction property for them. Moreover, a suitable form of serializability is guaranteed to hold for flat cJoin. We show that the cJoin encoding of any zero-safe net is a flat process.

To facilitate the translation, we define the encoding of flat processes that are written in a suitable canonical form, where only a few elementary definition patterns are allowed. This can be done without loss of generality, as we show that any flat process can be transformed in an equivalent process in canonical form. The elementary definition patterns we consider are inspired by the basic shapes of transitions in zero-safe nets: they are obtained by imposing a strict bound on the number of messages that can be consumed / produced within a single reduction.

Although we show that Join is expressive enough to encode flat cJoin, i.e. that the new primitives for flat negotiations do not increase the expressivity of the language, we argue that the syntax of cJoin yields a separation of concerns that is difficult to achieve at the level of Join, thus cJoin facilitates programming and reasoning about distributed contracts. We conjecture that by further elaborating the encoding of flat processes one should be able to implement full cJoin in Join.

Structure of the paper. In § 2 we present the syntax and semantics of cJoin. In § 3 we define the type system for flat processes, prove subject reduction and show that the encoding of zero-safe nets presented in [5] yields flat processes. Moreover, we show that flat processes have equivalent canonical representatives that employ only elementary definition patterns. In § 4 we present a correct and complete distributed encoding of canonical flat cJoin processes in Join.

$$M,N ::= 0 \mid x\langle\vec{y}\rangle \mid M|N \qquad\qquad D,E ::= J \triangleright P \mid J \blacktriangleright P \mid D \wedge E$$
$$P,Q ::= M \mid abort \mid [P:Q] \mid P|Q \mid \mathbf{def}\, D \,\mathbf{in}\, P \qquad\qquad J,K ::= x\langle\vec{y}\rangle \mid J|K$$

Figure 1. cJoin Calculus Syntax.

$$dn(x\langle\vec{y}\rangle)=\{x\} \qquad dn(J|K)=dn(J)\cup dn(K) \qquad rn(x\langle\vec{y}\rangle)=\{\vec{y}\} \qquad rn(J|K)=rn(J)\cup rn(K)$$

$$dn_{\mathrm{o}}(D\wedge E)=dn_{\mathrm{o}}(D)\cup dn_{\mathrm{o}}(E) \qquad dn_{\mathrm{o}}(J\triangleright P)=dn(J) \qquad dn_{\mathrm{o}}(J\blacktriangleright P)=\emptyset$$
$$dn_{\mathrm{m}}(D\wedge E)=dn_{\mathrm{m}}(D)\cup dn_{\mathrm{m}}(E) \qquad dn_{\mathrm{m}}(J\triangleright P)=\emptyset \qquad dn_{\mathrm{m}}(J\blacktriangleright P)=dn(J)$$

$$fn(D\wedge E)=fn(D)\cup fn(E) \qquad fn(J\blacktriangleright P)=fn(J\triangleright P)=dn(J)\cup(fn(P)\setminus rn(J))$$
$$fn(0)=\emptyset \qquad fn(abort)=\emptyset \qquad fn(x\langle\vec{y}\rangle)=\{x\}\cup\{\vec{y}\} \qquad fn(P|Q)=fn(P)\cup fn(Q)$$
$$fn(\mathbf{def}\, D \,\mathbf{in}\, P)=(fn(P)\cup fn(D))\setminus dn(D) \qquad fn([P:Q])=fn(P)\cup fn(Q)$$

Figure 2. Defined, received, and free names.

2 Background

cJoin syntax. The Join calculus [10] is a *process description language* (PDL) with asynchronous name-passing communication and it has the same expressive power as the asynchronous π-calculus. *Committed* Join (cJoin) [5] is a conservative extension of Join with additional high-level primitives for programming dynamic nested negotiations with compensations. Like Join, cJoin relies on an infinite set of names $x,y,...,u,v,...$ to model communication channels and transmitted values. Name tuples are written $\vec{u}$. The syntax of cJoin is given in Figure 1. cJoin differs from Join because of the additional operators *abort*, $[P:Q]$ and $J \blacktriangleright P$.

Messages M can be either the inert process 0, the asynchronous emission $x\langle\vec{y}\rangle$ of message $\vec{y}$ on port x, or the parallel composition of messages $M|N$.

Processes P, can be plain messages, the special constant *abort* causing the abort of its enclosing negotiation, a negotiation $[P:Q]$, where P is the normal execution of the activity and Q is its compensation in case of abort, the parallel composition of processes $P|Q$, or a process **def** D **in** P equipped with local ports defined by D.

A *definition* D is a conjunction of ordinary and merge reaction rule, $J \triangleright P$ and $J \blacktriangleright P$ respectively, that associate *join-patterns* J with *guarded processes* P. Names introduced by the definition D of **def** D **in** P are bound in the whole process P as well as in the guarded processes contained in D. The sets of defined names dn, received names rn and free names fn are defined in Figure 2. In particular, we distinguish between defined *ordinary* names $dn_{\mathrm{o}}(D)$ and defined *merge* names $dn_{\mathrm{m}}(D)$ that are always assumed to be disjoint sets of names.

CHAM. The operational semantics of cJoin is given in the reflexive CHAM style [10], where states (called *solutions*) are finite multisets of terms (called *molecules*), and computations are multiset rewrites. Multisets are written as $m_1,\ldots,m_n$. We usually abbreviate $m_1,\ldots,m_n$ with $\otimes_i m_i$. Solutions can be structured in a hierarchical way by using the operator *membrane* $\{[.]\}$, grouping a solution into a molecule. Transformations are described by a set of *chemical rules*, which can be of two different kinds: *heating / cooling* (or *structural*) rules $\rightleftharpoons$ for syntactical rearrangements of molecules in a solution, and *reaction* rules $\rightarrow$ for basic computation steps. Rules

$$m ::= P \mid D \mid \llcorner P \lrcorner \mid \{[S]\} \qquad S ::= m \mid m, S$$

Figure 3. Syntax of cJoin molecules and solutions.

only address the part of the solution that actually moves and can be applied at any level in the hierarchy. Molecules m and solutions S for cJoin are in Figure 3.

Note that processes and definitions are molecules. Additionally, molecules having the form $\llcorner Q \lrcorner$ denote compensations that are frozen inside a solution and that will not be executed unless their negotiation aborts. To reason up-to structural equivalence, we shall overload $\rightarrow$ to denote also sequences $\rightleftharpoons^* \rightarrow \rightleftharpoons^*$.

Operational semantics of cJoin. The chemical rules for cJoin are given in Figure 4. The first five chemical rules are the ordinary ones for Join. Rule STR-NULL states that 0 can be added or removed from any solution. Rules STR-JOIN and STR-AND stand for the associativity and commutativity of $|$ and $\wedge$. STR-DEF denotes the activation of a local definition, which implements a static scoping discipline by properly renaming defined ports by *globally fresh* names. We write the substitution of names $x_1 \dots x_n$ by $y_1 \dots y_n$ as $\sigma = \{^{y_1 \cdots y_n}/_{x_1 \dots x_n}\}$, with $dom(\sigma) = \{x_1, \dots, x_n\}$ and $range(\sigma) = \{y_1, \dots, y_n\}$. We indicate with σ_N an injective substitution σ such that $dom(\sigma) = N$. We require newly defined names to be globally fresh, which means fresh w.r.t the implicit context in which the rule is applied. The reaction RED describes the application of an active definition $J \triangleright P$ to messages $J\sigma$ matching the pattern J (for a suitable substitution σ, with $dom(\sigma) = rn(J)$). The instance of J is consumed and replaced by a new instance $P\sigma$ of the guarded process P.

Rule STR-CONT states that a term denoting a contract corresponds to a sub-solution consisting of two molecules: the process P and its compensation Q, which is frozen (because the operator $\llcorner . \lrcorner$ forbids the enclosed process to compute). At commit time, the local resources M produced inside a negotiation are released via the rule COMMIT, which can be executed only when all internal computations have finished. At commit time, private definitions of a contract can be discarded, because neither the messages that are being released contain those names nor they could have been extruded previously. After commit, its compensation procedure $\llcorner Q \lrcorner$ is useless and can be discarded as well. The abortion of a negotiation is handled by the rule ABORT, which releases Q whenever *abort* is present in the solution.

Interactions among negotiations are dealt with MERGE, which consumes messages from different contracts and creates a larger negotiation by combining the definitions and messages of the original ones with a new instance of the guarded process $P\sigma$, where $dom(\sigma) = rn(J_1 | \dots | J_n)$. Name clashes are avoided because we assume that STR-DEF generates globally fresh names. The compensation for the joint negotiation is the parallel composition of all the original compensations.

Example 2.1 *Mailing list.* Consider a data structure that allows to send atomically a message to a list of subscribers (in the sense that it is either sent to all or to none). Such structure can be defined as $\mathsf{ML} \equiv \mathit{MailingList}\langle k \rangle \triangleright \mathsf{MLDef}$, where:

$$\mathsf{MLDef} \equiv \textbf{def}\ \mathsf{List}\ \textbf{in}\ k\langle add, tell, close \rangle \mid l\langle nil \rangle$$

$$
\begin{array}{lrcl}
\text{STR-NULL} & 0 & \rightleftharpoons & \\
\text{STR-JOIN} & P \mid Q & \rightleftharpoons & P, Q \\
\text{STR-AND} & D \wedge E & \rightleftharpoons & D, E \\
\text{STR-DEF} & \mathbf{def}\ D\ \mathbf{in}\ P & \rightleftharpoons & D\sigma_{dn(D)}, P\sigma_{dn(D)}\ (range(\sigma_{dn(D)})\ \text{ globally fresh}) \\
\text{RED} & J \triangleright P, J\sigma & \to & J \triangleright P, \sigma \\
\text{STR-CONT} & [P : Q] & \rightleftharpoons & \{[P, \llcorner Q \lrcorner]\} \\
\text{COMMIT} & \{[M|\mathbf{def}\ D\ \mathbf{in}\ 0, \llcorner Q \lrcorner]\} & \to & M \\
\text{ABORT} & \{[abort|P, \llcorner Q \lrcorner]\} & \to & Q \\
\text{MERGE} & J_1|\ldots|J_n \blacktriangleright P, \bigotimes_i \{[J_i\sigma, S_i, \llcorner Q_i \lrcorner]\} & \to & J_1|\ldots|J_n \blacktriangleright P, \{[\bigotimes_i S_i, P\sigma, \llcorner Q_1|\ldots|Q_n \lrcorner]\}
\end{array}
$$

Figure 4. Operational semantics of cJoin.

$$
\begin{array}{rl}
\mathsf{List}\ \equiv & nil\langle v,w\rangle \blacktriangleright w\langle\rangle \\
\wedge & l\langle y\rangle \mid add\langle x\rangle \triangleright \mathbf{def}\ z\langle v,w\rangle \blacktriangleright x\langle v\rangle \mid y\langle v,w\rangle\ \mathbf{in}\ l\langle z\rangle \\
\wedge & l\langle y\rangle \mid tell\langle v\rangle \triangleright [\mathbf{def}\ z\langle\rangle \triangleright 0\ \mathbf{in}\ y\langle v,z\rangle \mid l\langle y\rangle : l\langle y\rangle] \\
\wedge & l\langle y\rangle \mid close\langle\rangle \triangleright 0
\end{array}
$$

A new mailing list is created by sending a message to the port *MailingList*. Since cJoin adheres to the "continuation passing" style of programming, the content of the message sent to *MailingList* is a continuation port k, which expects information about the newly created mailing list. The creation of a new list defines five fresh ports *nil, l, add, tell* and *close*: three of them (namely *add*, *tell*, and *close*) will be used to interact with the list from "outside" and will be sent to the port k as the outcome of the creation. The remaining two ports will never be extruded. They denote the empty list (*nil*) and the actual state of the list (l).

Once a list is created, a new subscriber can be added by sending a message *add* with the name x of the port where it will be listening to for new messages. In this case, the list is modified by installing z (on top of it), a forwarder of messages to x.

The port *tell* is used to send a message v to the list. When *tell* is received a new negotiation identified by a fresh name z is generated, and the state of the structure is put inside the negotiation, therefore all other activities, such as adding or closing are blocked until the negotiation ends. Inside the negotiation, the message v is sent to the forwarder at the top of the list y with the identifier of the negotiation z. Note that each forwarder sends the message to the corresponding subscriber and to the following forwarder in the list. This is repeated until *nil* is reached, when a message to the identifier of the transaction is sent. The firing rule $z\langle\rangle \triangleright 0$ consumes the last local name and the contract commits by releasing all the messages addressed to the subscribers and the state of the list. Then the list is ready to serve new requests.

3 Flat cJoin

Flat transactions were introduced in database community as a basic mechanism to assure atomic execution of composed activities. The term flat specifies that the activities forming a transaction are basic actions, such as read and write, but they cannot be transactions themselves. Similarly, we define a sub-calculus of cJoin, called *flat* cJoin, where negotiations cannot be nested. In this section we characterize flat processes as well-typed terms and we show that any cJoin process can be written in an equivalent canonical form.

3.1 A type system for flat cJoin

We single out flat processes of cJoin with the type system in Figure 5. It takes the set $T = \{\Box_0, \Box_1, \Box_2\}$ of types and uses the following type judgements:

$\vdash P : \Box_0$	The constructor of negotiations $[\_ : \_]$ does not appear at all in P.
$\vdash P : \Box_1$	P does not contain active negotiations but can activate flat contracts.
$\vdash P : \Box_2$	P can have or generate flat negotiations but not nested ones.
$\vdash D : \Box_0$	D does not contain constructors for negotiations.
$\vdash D : \Box_1$	D can contain or initiate flat negotiations but not nested ones.

Rules (SUB-P) and (SUB-D) stand for the sub-type order $\Box_0 < \Box_1 < \Box_2$. We say that a process P (resp. a definition D) is *well-typed* if $\vdash P : \Box_2$ (resp. $\vdash D : \Box_1$).

Clearly, the inert process 0, the emission of a message $x\langle\vec{y}\rangle$ and the constant *abort* do not contain constructors for negotiations, and are typed $\Box_0$. By rule (PAR), the parallel composition $P|Q$ can be typed $\Box_i$ if both P and Q type $\Box_i$. Consequently, the type of $P|Q$ corresponds to the greatest of the lower types that can be assigned to P and Q. In fact, considering P and Q well-typed, if P contains an active negotiation (i.e., $\vdash P : \Box_2$), independently of the structure of Q, the process $P|Q$ contains an active contract (i.e., $\vdash P|Q : \Box_2$). Rule (NEG) prevents nesting by stating that $[P : Q]$ can be typed $\Box_2$ only when P does not have negotiations (i.e., $\vdash P : \Box_0$). Instead, the compensation Q can use negotiations in definitions. This will not compromise flat condition because compensations execute at the top-level and not inside the negotiations they are originated from. Rule (DEF) combines the typing of definitions and processes. Note that **def** D **in** P can be typed $\Box_0$ only if neither D nor P use constructors for negotiations, i.e. if both have type $\Box_0$. Instead, it can be typed $\Box_1$ when negotiations appear only in definitions (D or those contained in P). Finally, if **def** D **in** P types $\Box_2$, its active negotiations appear in P, which therefore types $\Box_2$.

By rule (CONJ), a conjuction of definitions is typed $\Box_i$ only when both subterms type $\Box_i$. By rules (ORD) and (ORD-0), an ordinary definition $J \triangleright P$ is well-typed when its guarded processes P is well-typed. Moreover, it has type $\Box_0$ if P does not contain constructors for negotiations (i.e., $\vdash P : \Box_0$). Differently, a merge rule is well-typed only if P has type $\Box_0$ (rule (MERGE)). This is required in order to avoid nesting, because the instances of P will execute inside a negotiation.

Example 3.1 *Well-typed terms.* Consider the mailing list process introduced in Example 2.1. Several subterms and their types are below:

$$P_1 \equiv \textbf{def}\ z\langle\rangle \triangleright 0\ \textbf{in}\ y\langle v,z\rangle \mid l\langle y\rangle \qquad P_2 \equiv [P_1 : l\langle y\rangle]$$
$$D_1 \equiv l\langle y\rangle \mid tell\langle v\rangle \triangleright P_2 \qquad D_2 \equiv l\langle y\rangle \mid close\langle\rangle \triangleright 0$$

$$\vdash P_1 : \Box_0 \qquad \vdash P_2 : \Box_2 \qquad \vdash D_1 : \Box_1 \qquad \vdash D_2 : \Box_0 \qquad \vdash D_1 \wedge D_2 : \Box_1$$

$$
\begin{array}{ccccc}
\text{(Sub-P)} & \text{(Sub-D)} & \text{(Zero)} & \text{(Mess)} & \text{(Abort)} \\
\dfrac{\vdash P : \Box_i}{\vdash P : \Box_j}\; i<j & \dfrac{\vdash D : \Box_0}{\vdash D : \Box_1} & \vdash 0 : \Box_0 & \vdash x\langle y\rangle : \Box_0 & \vdash abort : \Box_0
\end{array}
$$

$$
\begin{array}{ccc}
\text{(Par)} & \text{(Neg)} & \text{(Def)} \\
\dfrac{\vdash P : \Box_i \quad \vdash Q : \Box_i}{\vdash P|Q : \Box_i} & \dfrac{\vdash P : \Box_0 \quad \vdash Q : \Box_1}{\vdash [P : Q] : \Box_2} & \dfrac{\vdash D : \Box_i \quad \vdash P : \Box_j}{\vdash \mathbf{def}\ D\ \mathbf{in}\ P : \Box_{max(i,j)}}
\end{array}
$$

$$
\begin{array}{cccc}
\text{(Conj)} & \text{(Ord-0)} & \text{(Ord)} & \text{(Merge)} \\
\dfrac{\vdash D : \Box_i \quad \vdash E : \Box_i}{\vdash D \wedge E : \Box_i} & \dfrac{\vdash P : \Box_0}{\vdash J \triangleright P : \Box_0} & \dfrac{\vdash P : \Box_i}{\vdash J \triangleright P : \Box_1} & \dfrac{\vdash P : \Box_0}{\vdash J \blacktriangleright P : \Box_0}
\end{array}
$$

Figure 5. Flat cJoin Typing.

Moreover, $\vdash \mathsf{MLDef} : \Box_1$ (it does not have active negotiations but can initiate them), and also $\vdash \mathsf{ML} : \Box_1$.

Example 3.2 *Counterexample.* The term $\mathbf{def}\ x\langle\rangle \blacktriangleright [P:0]\ \mathbf{in}\ [\mathbf{def}\ D\ \mathbf{in}\ x\langle\rangle : 0]$ is not well-typed because it has a merge definition whose guarded process is a negotiation (rule (Merge) cannot be applied because $\not\vdash x\langle\rangle \blacktriangleright [P:0] : \Box_0$). In fact, it reduces to $\mathbf{def}\ x\langle\rangle \blacktriangleright [P:0]\ \mathbf{in}\ [\mathbf{def}\ D\ \mathbf{in}\ [P:0]:0]$ when $x \notin dn(D)$, which has nested contracts.

Proposition 3.3 (Join processes are $\Box_0$) *Let P be a Join process, then* $\vdash P : \Box_0$.

Lemma 3.4 (Subject Reduction for $\Box_0$) *Let $P : \Box_0$. If $P \to^* P'$ then $P' : \Box_0$.*

The following result assures that flat processes do not introduce nesting.

Theorem 3.5 (Subject Reduction for $\Box_2$) *Let $P : \Box_2$. If $P \to^* P'$ then $P' : \Box_2$.*

Subject reduction does not hold for $\Box_1$. Consider $P \equiv \mathbf{def}\ x\langle\rangle \triangleright [Q : Q']\ \mathbf{in}\ x\langle\rangle$, where $\vdash Q : \Box_0$ and $\vdash Q' : \Box_1$. Although $\vdash P : \Box_1$, P reduces to $P' \equiv \mathbf{def}\ x\langle\rangle \triangleright [Q : Q']\ \mathbf{in}\ [Q : Q']$, which can be typed $\Box_2$ but not $\Box_1$.

Definition 3.6 [Flat cJoin] Let P be a cJoin process. P is *flat* iff $\vdash P : \Box_2$. Flat cJoin is the sub-calculus of all flat processes.

In [5] we defined the class of shallow processes and proved a serializability result for them, meaning that disjoint negotiations cannot interfere with each other (unless they are merged). Although the definition of shallow processes is not reported here, it is trivial to check that flat processes are also shallow.

Corollary 3.7 (Serializability) *Any flat process P is shallow and thus serializable.*

3.2 Zero Safe Nets and cJoin.

Zero-safe nets (ZS *nets*) [6] have been introduced to model serializable transactions in concurrent systems. They support multiway transactions, i.e. with several entry and exit points and a statically unknown number of participants. Recently, they have been used in [4] to encode short-running transactions of Microsoft Biztalk®,

(FIRING)
$$\frac{S+Z\,[\rangle\, S'+Z' \in T}{(S+S'',Z+Z'') \rightarrow_T (S'+S'',Z'+Z'')}$$

(STEP)
$$\frac{(S_1,Z_1) \rightarrow_T (S_1',Z_1')\quad (S_2,Z_2) \rightarrow_T (S_2',Z_2')}{(S_1+S_2,Z_1+Z_2) \rightarrow_T (S_1'+S_2',Z_1'+Z_2')}$$

(CONCATENATION)
$$\frac{(S_1,Z) \rightarrow_T (S_1',Z'')\quad (S_2,Z'') \rightarrow_T (S_2',Z')}{(S_1+S_2,Z) \rightarrow_T (S_1'+S_2',Z')}$$

(CLOSE)
$$\frac{(S,\emptyset) \rightarrow_T (S',\emptyset)}{(S,\emptyset) \Rightarrow_T (S',\emptyset)}$$

Figure 6. Operational semantics of ZS nets (+ denotes multiset union).

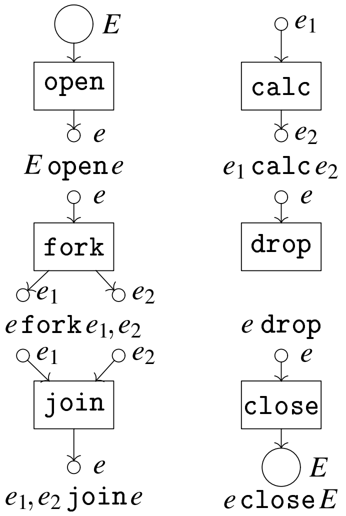
a commercial workflow management system [13]. However, ZS nets are not suitable to model interesting aspects such as name mobility, programmable compensations and nesting, which are the main features of cJoin.

Analogously to Petri nets, ZS nets rely on *places* (i.e. repositories of resources, messages), *tokens* (i.e. instances of places), *markings* U (i.e. multisets of place) and *transitions* $U[\rangle U'$ (i.e. basic activities to fetch and produce multisets of tokens). However, the places of ZS nets are partitioned into ordinary and transactional ones (called *stable* and *zero*, respectively). Correspondingly, markings U can be seen as pairs (S,Z) with $U = S+Z$, where S and Z are the multisets of stable and zero resources, respectively. Tokens in zero places are transient data belonging to some ongoing negotiation, while tokens in stable places model committed decisions achieved via negotiations, which start from and lead to *stable markings* (i.e. multisets of stable places). The key point is that stable tokens produced inside a negotiation are made available only at commit time, when no zero tokens are left.

The operational semantics of ZS nets is defined by the two relations $\Rightarrow_T$ and $\rightarrow_T$ (indexed by the set of transitions T) in Figure 6. Rules FIRING and STEP are the ordinary ones for Petri nets. The rule CONCATENATION composes zero tokens in series but stable tokens in parallel, hence stable tokens produced by the first step cannot be consumed by the second step. A negotiation $(S,\emptyset) \Rightarrow_T (S',\emptyset)$ is a concatenation of steps from a stable marking to a stable markings (rule CLOSE).

In the literature, ZS nets have been already encoded in Join [4] (via a distributed two-phase commit protocol for establishing the end of a negotiation) and in cJoin [5] (almost straightforwardly, taking advantage of the additional negotiation primitives). We briefly recall the latter encoding $[\![\_]\!]_{\mathsf{cJ}}$ because:

- Without loss of generality, both encodings are defined for ZS nets made with the basic shapes in Figure 7(a) (which are as expressive as the general nets), for E any stable place and e, e_1, e_2 any zero places. We use mnemonic names like E `open` e to denote a transition $E\,[\rangle\, e$ that can spawn a fresh local negotiation and e `fork` e_1, e_2 to denote a transition $e\,[\rangle\, e_1+e_2$ that can create parallel threads within a running negotiation. Basic shapes are analogous to the elementary definition patterns we shall consider when encoding flat cJoin in Join.
- ZS nets do not have programmable compensations. The encoding $[\![\_]\!]_{\mathsf{cJ}}$ shows that suitable default compensations can just restore the initial state of the negotiation. As an original result, in Proposition 3.8 below we prove that ZS nets are encoded as flat processes.



(a) Basic shapes of ZS nets.

$$[\![E\ \texttt{open}\ e]\!]_{\mathsf{cJ}} ::= E\langle\rangle \triangleright [\mathbf{def}\ z\langle\rangle \triangleright 0\ \mathbf{in}\ e\langle z\rangle : E\langle\rangle]$$
$$[\![e\ \texttt{fork}\ e_1, e_2]\!]_{\mathsf{cJ}} ::= e\langle z\rangle \blacktriangleright e_1\langle z\rangle | e_2\langle z\rangle$$
$$[\![e_1, e_2\ \texttt{join}\ e]\!]_{\mathsf{cJ}} ::= e_1\langle z_1\rangle | e_2\langle z_2\rangle \blacktriangleright e\langle z_1\rangle$$
$$[\![e_1\ \texttt{calc}\ e_2]\!]_{\mathsf{cJ}} ::= e_1\langle z\rangle \blacktriangleright e_2\langle z\rangle$$
$$[\![e\ \texttt{drop}\ ]\!]_{\mathsf{cJ}} ::= e\langle z\rangle \blacktriangleright 0$$
$$[\![e\ \texttt{close}\ E]\!]_{\mathsf{cJ}} ::= e\langle z\rangle \blacktriangleright E\langle\rangle$$

$$[\![E]\!]_{\mathsf{cJ}} ::= E\langle\rangle$$
$$[\![S_1 + S_2]\!]_{\mathsf{cJ}} ::= [\![S_1]\!]_{\mathsf{cJ}} | [\![S_2]\!]_{\mathsf{cJ}}$$

(b) Translation of basic shapes and markings.

Figure 7. Encoding of ZS Nets in cJoin.

Encoding ZS nets in cJoin. The translation $[\![\_]\!]_{\mathsf{cJ}}$ in Figure 7(b) associates a cJoin definition (resp. message) with each basic shape of transitions (resp. stable marking). Places are seen as ports and tokens as messages. Tokens in stable places carry no value, while tokens in zero places carry the identifier of the transaction they belong to. For T the set of transitions and S the initial marking of the ZS nets, we let $[\![T]\!]_{\mathsf{cJ}} = \bigwedge_{t\in T}[\![t]\!]_{\mathsf{cJ}}$ and then take the cJoin process **def** $[\![T]\!]_{\mathsf{cJ}}$ **in** $[\![S]\!]_{\mathsf{cJ}}$, which consists of the translation of the initial marking S in the environment containing all the definitions associated with transitions in T. Transitions whose pre-sets contain zero places are translated as merge definitions, otherwise as ordinary Join definitions.

We shortly discuss a few peculiarities of the encoding (details are in [5]). The translation of a transition of the form E `open` e is a cJoin definition that can open a new negotiation containing the definition of a fresh name z (the identifier of the transaction) together with the message $e\langle z\rangle$, and whose default compensation is the only stable resource $E\langle\rangle$. The dummy definition $z\langle\rangle \triangleright 0$ is a convenient way to define a local identifier for the negotiation and has no computational meaning. In fact, no message will ever be produced on port z. The port e corresponds to the homonymous zero place and it is a name defined externally via merge definitions (originated from those transitions in T fetching from place e), which can be used to compute inside negotiations and even merge them via the reaction MERGE of cJoin. For example, two disjoint negotiations with local tokens in e_1 and e_2 can be merged by firing a transition e_1, e_2 `join` e, i.e. by executing the MERGE reaction for $e_1\langle z_1\rangle | e_2\langle z_2\rangle \blacktriangleright e\langle z_1\rangle$. Note that the identifiers z_1 and z_2 become then equivalent identifiers for the same larger negotiation. The key point is that when stable messages $E\langle\rangle$ are released inside a negotiation, e.g., by firing e `close` E, then they cannot be fetched before the negotiation commits, because all the rules that can consume them are ordinary ones and float outside the negotiation boundaries.

While the correctness and completeness of the encoding can be found in [5], here we state the following original result based on the type system in Figure 5.

$count(0) = 1$	$count(x\langle\vec{u}\rangle) = 1$	$count(P\vert Q) = count(P) + count(Q)$
$count(abort) = 1$	$count([P : Q]) = count(P)$	$count(\mathbf{def}\ D\ \mathbf{in}\ P) = count(P)$

Figure 8. Definition of *count*(*P*).

OPEN	$x\langle\vec{v}\rangle \triangleright P$	&	$\vdash P : \Box_2$	&	$count(P) = 1$
ORD-MOV	$x\langle\vec{u}\rangle \triangleright P$	&	$\vdash P : \Box_1$	&	$count(P) \leq 2$
MERGE-MOV	$x\langle\vec{u}\rangle \blacktriangleright P$	&	$\vdash P : \Box_0$	&	$count(P) \leq 2$
ORD-JOIN	$x\langle\vec{v}\rangle\vert y\langle\vec{w}\rangle \triangleright P$	&	$\vdash P : \Box_1$	&	$count(P) = 1$
MERGE-JOIN	$x_1\langle\vec{v}_1\rangle\vert\ldots\vert x_n\langle\vec{v}_n\rangle \blacktriangleright P$	&	$\vdash P : \Box_0$	&	$count(P) = 1$

Figure 9. Definitions in canonical form

Proposition 3.8 (The cJoin encoding of ZS nets is flat) $\vdash \mathbf{def}\ [\![T]\!]_{\mathsf{cJ}}\ \mathbf{in}\ [\![S]\!]_{\mathsf{cJ}} : \Box_2$.

3.3 A canonical form for flat processes

As done with ZS nets, we will restrict our attention to processes built with some basic shapes to simplify the definition of the encoding of flat cJoin into Join. In particular, we forbid definitions to consume and produce messages freely. The auxiliary function *count* in Figure 8 counts the atomic agents present in a process.

Definition 3.9 [Canonical Form] Let *P* be a flat process, *P* is in canonical form if any definition in *P* satisfies one of the conditions in Figure 9.

It is worth noting that these conditions match with the basic shapes of ZS nets. By (OPEN), a reaction that creates a new negotiation consumes exactly one message and produces only one agent inside the new negotiation. Rule (ORD-JOIN) assures that a synchronization consumes two messages and produces exactly a new agent. Differently, rule (MERGE-JOIN) allows to join several negotiations simultaneously. Moreover, a join cannot spawn directly a new negotiation (a task left to (OPEN)). Finally, rules (ORD-MOV) and (MERGE-MOV) are instances of transitions `calc`, `fork`, and `close` (with `drop` as a particular case) of ZS nets.

Proposition 3.10 *Let P be a flat process. P can be written as an equivalent canonical flat process.*

Example 3.11 The process MLDef in Example 2.1 is not in canonical form. In fact, the definition $\mathsf{Tell} \equiv l\langle y\rangle \mid tell\langle v\rangle \triangleright [\mathbf{def}\ z\langle\rangle \triangleright 0\ \mathbf{in}\ y\langle v,z\rangle \mid l\langle y\rangle : l\langle y\rangle]$ is a `join` that creates a negotiation with two internal messages. It can be rewritten as

$$\begin{aligned}\mathsf{Tell}' \equiv\ & l\langle y\rangle \mid tell\langle v\rangle \triangleright a\langle y,v\rangle\\ & \wedge\ a\langle y,v\rangle \triangleright [\mathbf{def}\ z\langle\rangle \triangleright 0\ \mathbf{in}\ \ \mathbf{def}\ b\langle\rangle \triangleright y\langle v,z\rangle \mid l\langle y\rangle\ \mathbf{in}\ b\langle\rangle : l\langle y\rangle]\end{aligned}$$

where *a* and *b* are fresh names. Note that $\vdash \mathsf{Tell}' : \Box_1$. Its first rule is an ORD-JOIN, while the second is an OPEN. In fact, the process contained in the negotiation has type $\Box_0$ and the count of emitted messages is 1 (i.e. $b\langle\rangle$). The definitions appearing inside the contract are in canonical form, actually they corresponds to ORD-MOV: $z\langle\rangle \triangleright 0$ is a `drop` and $b\langle\rangle \triangleright y\langle v,z\rangle \mid l\langle y\rangle$ is a `fork`.

TOP-LEVEL PROCESSES

$$
\begin{array}{lll}
[\![0]\!]_{\mathsf{S},\mathsf{B}} & = 0 & \\
[\![x\langle\vec{u}\rangle]\!]_{\mathsf{S},\mathsf{B}} & = x^{\mathsf{s}}\langle\vec{u}\rangle & \textit{if } x \notin \mathsf{B} \;\&\; \vec{u} \in \mathsf{S} \\
[\![x\langle\vec{u}\rangle]\!]_{\mathsf{S},\mathsf{B}} & = x^{\mathsf{b}}\langle\vec{u}\rangle & \textit{if } x \notin \mathsf{B} \;\&\; \vec{u} \in \mathsf{B} \\
[\![x\langle\vec{u}\rangle]\!]_{\mathsf{S},\mathsf{B}} & = 0 & \textit{if } x \in \mathsf{B} \\
[\![P|Q]\!]_{\mathsf{S},\mathsf{B}} & = [\![P]\!]_{\mathsf{S},\mathsf{B}} | [\![Q]\!]_{\mathsf{S},\mathsf{B}} & \\
[\![abort]\!]_{\mathsf{S},\mathsf{B}} & = 0 & \\
[\![\mathbf{def}\, D \,\mathbf{in}\, P]\!]_{\mathsf{S},\mathsf{B}} & = \mathbf{def}\, [\![D]\!]^{0}_{\mathsf{S}',\mathsf{B}'} \,\mathbf{in}\, [\![P]\!]_{\mathsf{S}',\mathsf{B}'} & \mathsf{S}' = \mathsf{S} \uplus dn_{\mathsf{o}}(D) \;\&\; \mathsf{B}' = \mathsf{B} \uplus dn_{\mathsf{m}}(D) \\
[\![[P:Q]]\!]_{\mathsf{S},\mathsf{B}} & = \mathbf{def}\, \mathsf{D} \wedge cmp\langle\rangle \triangleright [\![Q]\!]_{\mathsf{S},\mathsf{B}} \,\mathbf{in}\, state\langle\{cmp\}\rangle \mid [\![P]\!]^{put,abt,\{lock\}}_{\mathsf{S},\mathsf{B}} &
\end{array}
$$

PROCESSES IN A NEGOTIATION

$$
\begin{array}{lll}
[\![0]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = p\langle\ell,\emptyset,\emptyset\rangle & \\
[\![x\langle\vec{u}\rangle]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = p\langle\ell,\emptyset,\{x^{\mathsf{s}}\langle\vec{u}\rangle\}\rangle & \textit{if } x \in \mathsf{S} \;\&\; \vec{u} \in \mathsf{S} \\
[\![x\langle\vec{u}\rangle]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = p\langle\ell,\emptyset,\{x^{\mathsf{b}}\langle\vec{u}\rangle\}\rangle & \textit{if } x \in \mathsf{S} \;\&\; \vec{u} \in \mathsf{B} \\
[\![x\langle\vec{u}\rangle]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = 0 & \textit{if } x \in \mathsf{S} \;\&\; \vec{u} \notin (\mathsf{S}\cup\mathsf{B}) \\
[\![x\langle\vec{u}\rangle]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = x^{\mathsf{s}}\langle\vec{u},p,a,\ell\rangle & \textit{if } x \notin \mathsf{S}\cup\mathsf{B} \;\&\; \vec{u} \in \mathsf{S} \\
[\![x\langle\vec{u}\rangle]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = x^{\mathsf{b}}\langle\vec{u},p,a,\ell\rangle & \textit{if } x \notin \mathsf{S}\cup\mathsf{B} \;\&\; \vec{u} \in \mathsf{B} \\
[\![x\langle\vec{u}\rangle]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = x^{\mathsf{z}}\langle\vec{u},p,a,\ell\rangle & \textit{if } x \notin \mathsf{S}\cup\mathsf{B} \;\&\; \vec{u} \notin \mathsf{S}\cup\mathsf{B} \\
[\![x\langle\vec{u}\rangle]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = x^{\mathsf{s}}_{s}\langle\vec{u},p,a,\ell\rangle & \textit{if } x \in \mathsf{B} \;\&\; \vec{u} \in \mathsf{S} \\
[\![x\langle\vec{u}\rangle]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = x^{\mathsf{b}}_{s}\langle\vec{u},p,a,\ell\rangle & \textit{if } x \in \mathsf{B} \;\&\; \vec{u} \in \mathsf{B} \\
[\![x\langle\vec{u}\rangle]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = x^{\mathsf{z}}_{z}\langle\vec{u},p,a,\ell\rangle & \textit{if } x \in \mathsf{B} \;\&\; \vec{u} \notin \mathsf{S}\cup\mathsf{B} \\
[\![abort]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = a\langle\rangle & \\
[\![\mathbf{def}\, D \,\mathbf{in}\, P]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & = \mathbf{def}\, [\![D]\!]^{1}_{\mathsf{S},\mathsf{B}} \,\mathbf{in}\, [\![P]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}} & \textit{if } count(P) = 1 \\
[\![\mathbf{def}\, D \,\mathbf{in}\, P]\!]^{(p_1,p_2),(a_1,a_2),\ell}_{\mathsf{S},\mathsf{B}} & = \mathbf{def}\, [\![D]\!]^{1}_{\mathsf{S},\mathsf{B}} \,\mathbf{in}\, [\![P]\!]^{(p_1,p_2),(a_1,a_2),\ell}_{\mathsf{S},\mathsf{B}} & \textit{if } count(P) = 2 \\
[\![P \mid Q]\!]^{(p_1,p_2),(a_1,a_2),\ell}_{\mathsf{S},\mathsf{B}} & = [\![P]\!]^{p_1,a_1,\ell}_{\mathsf{S},\mathsf{B}} \mid [\![Q]\!]^{p_2,a_2,\ell}_{\mathsf{S},\mathsf{B}} & \textit{if } count(P) = count(Q) = 1
\end{array}
$$

Figure 10. Encoding of canonical flat processes.

Proposition 3.12 (The encoding of ZS is in canonical form) *Let $N = (T,S)$ be a* ZS *net, then* **def** $[\![T]\!]_{\mathsf{cJ}}$ **in** $[\![S]\!]_{\mathsf{cJ}}$ *is already in canonical form.*

4 Encoding flat cJoin in Join.

In this section we describe the encoding in Join of canonical flat cJoin processes. As we are interested in computations that start from and lead to consistent states, we restrict our attention to processes that start without active negotiations, that is canonical flat cJoin processes that additionally type $\square_1$. For simplicity, the encoding relies on Join calculus extended with the data type *SET*, for finite sets and the standard operations of emptyset $\emptyset$, union $\cup$, and difference $\setminus$.

Processes are encoded by considering two sets of names: S denoting a set of ordinary names and B containing merge names, which are used to decide whether a free name in P is an ordinary or a merge one. Therefore, the encoding is well-defined only when $fn(P) \subseteq \mathsf{S}\cup\mathsf{B}$ and $\mathsf{S}\cap\mathsf{B} = \emptyset$.

Definition 4.1 [Encoding]. The Join process associated to a canonical flat cJoin process P with type $\square_1$ is $[\![P]\!]_{fn(P),\emptyset}$ (see Figure 10).

Top-level processes. The function $[\![P]\!]_{\mathsf{S},\mathsf{B}}$ defines the encoding for top-level processes. Note that the emission of a message in a stable name x is translated as a message on x^{s} or x^{b} considering whether the parameters $\vec{u}$ are ordinary or merge names. Ports x^{s} or x^{b} are introduced by the encoding of definitions presented below. For simplicity we assume all names in $\vec{u}$ are either ordinary or merge, but the presentation can be extended by using a different port for any possible combination.

A top-level message $x\langle\vec{u}\rangle$ on a merge name ($x \in \mathsf{B}$) lives outside a negotiation and cannot be consumed. Moreover, it is not observable because x is a defined name. Consequently, it is useless and encoded as the inert process 0. Analogously for *abort*, which is meaningless outside contracts.

Note that S and B are updated when encoding a top-level process with local definitions, i.e. to S' and B' when defining $[\![\mathbf{def}\ D\ \mathbf{in}\ P]\!]_{\mathsf{S},\mathsf{B}}$. In this case, both D and P are encoded by taking into account $dn(D)$. We use $\_ \uplus \_$ to denote the union of disjoint sets. (Note that defined names can always be renamed with fresh ones.)

When a negotiation is translated into Join, it is associated with a new coordinator D (Figure 12), which will monitor the execution of the contract. As P will run as part of a negotiation, it is encoded as $[\![P]\!]^{put,abt,\{lock\}}_{\mathsf{S},\mathsf{B}}$ where put, abt, $lock \in dn_{\mathsf{o}}(\mathsf{D})$. We can safely assume that P initiates with a unique thread because we are translating canonical processes with type $\Box_1$, and therefore negotiations $[P : Q]$ appear in definitions with $count(P) = 1$. The compensation Q is encoded as a top-level process, which is activated with a message on the local port cmp. As cmp is used only to initialize the state of the coordinator ($state\langle\{cmp\}\rangle$), the message $cmp\langle\rangle$ is emitted only when the coordinator (and consequently the contract) aborts.

Processes in negotiations. The auxiliary encoding $[\![\_]\!]^{p,a,\ell}_{\mathsf{S},\mathsf{B}}$ describes the implementation of a thread being monitored by a manager D that defines channels p and a for receiving commit or abort confirmations. The set ℓ collects the references to known parties in the same negotiation (called *synchronization set*). The inert process 0 in a negotiation means thread completion and it is translated as $p\langle\ell,\emptyset,\emptyset\rangle$ to notify that it is ready to commit. The message contains ℓ to inform D about known parties.

The encoding of a message $x\langle\vec{u}\rangle$ requires a case analysis on the different kinds of names involved in it. When the message is sent to a free name or to an ordinary name defined at the top-level ($x \in S$) there are two different cases. If the arguments $\vec{u}$ are not local names, e.g. $\vec{u} \in \mathsf{S}$, then the thread is attempting to close the negotiation by releasing $x\langle\vec{u}\rangle$. Hence it is encoding as a commit notification $p\langle\ell,\emptyset,\{x^{\mathsf{s}}\langle\vec{u}\rangle\}\rangle$. Note that $x^{\mathsf{s}}\langle\vec{u}\rangle$ will be released if the negotiation finally commits.

Instead, when the arguments are names defined in a contract, the negotiation can enter in a stall situation unless other participants abort the whole contract. In fact such message cannot be consumed before commit, which is required to enable the commit of the contract (COMMIT requires all local names not to appear in messages). The stall situation is encoded with 0, in this way the thread finishes without notifying its coordinator neither commit nor abort, and the coordinator will be blocked (unless one of its parties aborts).

On the other hand, a name x defined in a negotiation is encoded by using three different ports: x^{z}, x^{b} and x^{s} to handle different types of parameters, i.e., local, merge and top-level. Similarly, merge names are encoded taking into account the type of their parameters, but they also should consider that a negotiation can finish when the received names are not local. Port x_z^{k} (with $\mathsf{k} \in \{\mathsf{z},\mathsf{b},\mathsf{s}\}$) is used to encode the behavior of a merge name that receives names of type k and continues the execution of the negotiation. Instead, port x_s^{k} allows also the possibility of committing a contract even when the message is not consumed. Note that the emission on x is translated as a message that carries the values p, a and ℓ for interacting with the manager. (A thoughtful discussion about encoding merge definitions is below).

The constant process *abort* is translated into a message $a\langle\rangle$ that informs the manager about the abort. The translation of a process **def** D **in** P involves the translation of D and P. When $count(P) = 1$, P is encoded by using the same coordinator assigned to the whole process.We remark that the sets of variables S and B are not updated in this case, because D introduces just local names. Also, D is encoded with $[\![\_]\!]^1_{\mathsf{S},\mathsf{B}}$ and not with $[\![\_]\!]^0_{\mathsf{S},\mathsf{B}}$, which is used only for top-level definitions.

The encoding of the parallel execution $P|Q$ requires information about two different coordinators: two ports p_1 and p_2 for notifying the commit, and two ports a_1 and a_2 for aborting. Then, P is encoded by using p_1, a_1 and Q using p_2, a_2. Similarly for $[\![\textbf{def}\ D\ \textbf{in}\ P]\!]_{\mathsf{S},\mathsf{B}}$ when $count(P) = 2$. The generation of different coordinators is due to the encoding of `fork` definitions described below.

Definitions. The encoding of definitions is in Figure 11. We recall that $[\![\_]\!]^0_{\mathsf{S},\mathsf{B}}$ is for top-level definitions, while $[\![\_]\!]^1_{\mathsf{S},\mathsf{B}}$ is for definitions inside negotiations. In both cases the encoding of a conjunction $D \wedge E$ is the obvious one. The translation of a top-level definition of the port x creates two new ports x^{s} and x^{b}, which handle ordinary and merge parameters respectively. Such ports are associated to different translations of the guarded process P: x^{s} considers $\vec{u}$ as ordinary names and x^{b} as merge names. We recall that, for simplicity, we assume all names in $\vec{u}$ being of the same kind. We also commit in Figure 11 the encoding of a join, which generates four different rules: one for each combination of argument types.

The definition in a contract of $x\langle\vec{u}\rangle \triangleright P$ where $count(P) = 1$ is translated into three rules. Each rule introduces a new port x^{k} ($\mathsf{k} \in \{\mathsf{z},\mathsf{b},\mathsf{s}\}$) to handle a particular kind of received names $\vec{u}$. Port x^{z} receives local names, x^{b} merge names, and x^{s} top-level names. Additionally, the new ports x^{k} have as parameters p, a and ℓ because the encoding in Figure 10 needs such information to contact the manager of the contract where the message x belongs to. In fact, the guarded process P must be encoded w.r.t. the values p, a, ℓ of the manager of the fetched message on x^{k}.

Similarly, a `fork` is encoded with three rules (Figure 11 shows only the rule for x^{z}) but the guarded process P is translated by using two new coordinators D_1 and D_2. Ports put_i, abt_i and $lock_i$ are defined names of the new coordinators D_i, while p and a are the channels associated to the thread that forks (they are retrieved from the message on x). Channels $lock_i$ are added to the participant list ℓ, which will be common to both new threads. For simplicity, we close the original thread (and create two new ones) instead of reusing it. The compensations for the new threads are the channels necessary to abort the other two participants.

The remaining shape for ordinary definitions is a `join` $x\langle\vec{u}\rangle|y\langle\vec{v}\rangle \triangleright P$ where two

DEFINITIONS

$$[\![D \wedge E]\!]^i_{\mathsf{S,B}} = [\![D]\!]^i_{\mathsf{S,B}} \wedge [\![E]\!]^i_{\mathsf{S,B}} \qquad \textit{for } i = 1,2$$

$$[\![x\langle\vec{u}\rangle \triangleright P]\!]^0_{\mathsf{S,B}} = x^{\mathsf{s}}\langle\vec{u}\rangle \triangleright [\![P]\!]_{\mathsf{S}\uplus\{\vec{u}\},\mathsf{B}} \wedge x^{\mathsf{b}}\langle\vec{u}\rangle \triangleright [\![P]\!]_{\mathsf{S,B}\uplus\{\vec{u}\}} \qquad \text{(remaining patterns omitted)}$$

$$[\![x\langle\vec{u}\rangle \triangleright P]\!]^1_{\mathsf{S,B}} = x^{\mathsf{z}}\langle\vec{u},p,a,\ell\rangle \triangleright [\![P]\!]^{p,a,\ell}_{\mathsf{S,B}} \wedge x^{\mathsf{b}}\langle\vec{u},p,a,\ell\rangle \triangleright [\![P]\!]^{p,a,\ell}_{\mathsf{S,B}\uplus\{\vec{u}\}} \wedge x^{\mathsf{s}}\langle\vec{u},p,a,\ell\rangle \triangleright [\![P]\!]^{p,a,\ell}_{\mathsf{S}\uplus\{\vec{u}\},\mathsf{B}} \qquad \textit{if } count(P) = 1$$

$$\begin{aligned}[\![x\langle\vec{u}\rangle \triangleright P]\!]^1_{\mathsf{S,B}} = x^{\mathsf{z}}\langle\vec{u},p,a,\ell\rangle \triangleright \textbf{def } \mathsf{D}_1 \wedge \mathsf{D}_2 \textbf{ in } & [\![P]\!]^{(put_1,put_2),(abt_1,abt_2),\{lock_1,lock_2\}\cup\ell}_{\mathsf{S,B}} \\ & \mid p\langle \ell \cup \{lock_1,lock_2\},\{abt_1,abt_2\},\emptyset\rangle \\ & \mid state_1\langle\{abt_2,a\}\rangle \mid state_2\langle\{abt_1,a\}\rangle \\ \wedge\ x^{\mathsf{b}}\langle\vec{u},p,a,\ell\rangle \triangleright \ldots \ \wedge\ x^{\mathsf{s}}\langle\vec{u},p,a,\ell\rangle \triangleright \ldots & \qquad \textit{if } count(P) = 2\end{aligned}$$

$$\begin{aligned}[\![x\langle\vec{u}\rangle|y\langle\vec{v}\rangle \triangleright P]\!]^1_{\mathsf{S,B}} = {} & x^{\mathsf{z}}\langle\vec{u},p_1,a_1,\ell_1\rangle|y^{\mathsf{z}}\langle\vec{v},p_2,a_2,\ell_2\rangle \triangleright \\ & \textbf{def } \mathsf{D} \textbf{ in } p_1\langle\ell_1\cup\ell_2\cup\{lock\},\{abt,a_2\},\emptyset\rangle \mid [\![P]\!]^{put,abt,\ell_1\cup\ell_2\cup\{lock\}}_{\mathsf{S,B}} \\ & \mid p_2\langle\ell_1\cup\ell_2\cup\{lock\},\{abt,a_1\},\emptyset\rangle \mid state\langle\{a_1,a_2\}\rangle \\ & \wedge\ x^{\mathsf{z}}\langle\vec{u},p_1,a_1,\ell_1\rangle|y^{\mathsf{b}}\langle\vec{v},p_2,a_2,\ell_2\rangle \triangleright \ldots \wedge \ldots \qquad \textit{if } count(P) = 1\end{aligned}$$

$$\begin{aligned}[\![x\langle\vec{u}\rangle \blacktriangleright P]\!]^0_{\mathsf{S,B}} = {} & \textstyle\bigwedge_{\mathsf{k=s,b}} (x^{\mathsf{k}}_s\langle\vec{u},p,a,\ell\rangle \triangleright p\langle\ell,\emptyset,\emptyset\rangle \ \wedge\ x^{\mathsf{k}}_s\langle\vec{u},p,a,\ell\rangle \triangleright x^{\mathsf{k}}_z\langle\vec{u},p,a,\ell\rangle) \\ & \wedge x^{\mathsf{s}}_z\langle\vec{u},p,a,\ell\rangle \triangleright [\![P]\!]^{p,a,\ell}_{\mathsf{S}\uplus\{\vec{u}\},\mathsf{B}} \ \wedge\ x^{\mathsf{b}}_z\langle\vec{u},p,a,\ell\rangle \triangleright [\![P]\!]^{p,a,\ell}_{\mathsf{S,B}\uplus\{\vec{u}\}} \\ & \wedge x^{\mathsf{z}}_z\langle\vec{u},p,a,\ell\rangle \triangleright [\![P]\!]^{p,a,\ell}_{\mathsf{S,B}} \qquad \textit{if } count(P) = 1\end{aligned}$$

$$\begin{aligned}[\![x\langle\vec{u}\rangle \blacktriangleright P]\!]^0_{\mathsf{S,B}} = {} & \textstyle\bigwedge_{\mathsf{k=s,b}} (x^{\mathsf{k}}_s\langle\vec{u},p,a,\ell\rangle \triangleright p\langle\ell,\emptyset,\emptyset\rangle \ \wedge\ x^{\mathsf{k}}_s\langle\vec{u},p,a,\ell\rangle \triangleright x^{\mathsf{k}}_z\langle\vec{u},p,a,\ell\rangle) \\ & x^{\mathsf{z}}_z\langle\vec{u},p_1,a_1,\ell\rangle \triangleright \textbf{def } \mathsf{D}_1 \wedge \mathsf{D}_2 \textbf{ in } [\![P]\!]^{(put_1,put_2),(abt_1,abt_2),\{lock_1,lock_2\}\cup\ell}_{\mathsf{S,B}} \\ & \qquad \mid p\langle\ell\cup\{lock_1,lock_2\},\{abt_1,abt_2\},\emptyset\rangle \\ & \qquad \mid state_1\langle\{abt_2,a\}\rangle \mid state_2\langle\{abt_1,a\}\rangle \\ & x^{\mathsf{s}}_z\langle\vec{u},p_1,a_1,\ell\rangle \triangleright \ldots \wedge \ldots \qquad \textit{if } count(P) = 2\end{aligned}$$

$$\begin{aligned}[\![x_1\langle\vec{u}_1\rangle \mid \ldots \mid x_n\langle\vec{u}_n\rangle \blacktriangleright P]\!]^0_{\mathsf{S,B}} = {} & \textstyle\bigwedge_{\mathsf{k=s,b}} \bigwedge_i x^{\mathsf{k}}_{i_s}\langle\vec{u}_i,p,a,\ell\rangle \triangleright p\langle\ell,\emptyset,\emptyset\rangle \quad \wedge \\ & \textstyle\bigwedge_{\mathsf{k=s,b}} \bigwedge_i x^{\mathsf{k}}_{i_s}\langle\vec{u}_i,p,a,\ell\rangle \triangleright x^{\mathsf{k}}_{i_z}\langle\vec{u}_i,p,a,\ell\rangle \\ & \wedge x^{\mathsf{z}}_{1_z}\langle\vec{u}_1,p_1,a_1,\ell_1\rangle|\ldots|x^{\mathsf{z}}_{n_z}\langle\vec{u}_n,p_n,a_n,\ell_n\rangle \triangleright \\ & \qquad \textbf{def } \mathsf{D} \textbf{ in } state\langle\textstyle\bigcup_i\{a_i\}\rangle \mid [\![P]\!]^{put,abt,\bigcup_i\ell_i\cup\{lock\}}_{\mathsf{S,B}} \mid \\ & \qquad \textstyle\prod_j p_j\langle\bigcup_i\ell_i\cup\{lock\},\bigcup_i\{a_i\}\cup\{abt\},\emptyset\rangle \\ & \wedge\ \ldots \qquad \textit{if } count(P) = 1\end{aligned}$$

$$[\![J \blacktriangleright P]\!]^1_{\mathsf{S,B}} = \top$$

Figure 11. Encoding of canonical flat definitions.

different threads are synchronized and only one of them remains active ($count(P) = 1$). The translation states that the execution of a join ends both threads (messages to p_i), and encodes the guarded process P with a new coordinator D. The participant list for the three threads is $\ell_1 \cup \ell_2 \cup \{lock\}$. In this case, the omitted rules correspond to the different combinations of ports associated to x and y.

The last rules encode merge definitions, whose basic shapes are similar to ordinary definitions, consequently they are translated analogously. The main difference is that merge names have a non-deterministic behavior, because a negotiation can commit also when it contains messages addressed to a merge name or it can wait until those messages are consumed. Therefore, a merge name x is encoded with five different ports: x^{k}_z encode the waiting behavior (i.e., the negotiation will not commit until the message is consumed), and x^{s}_s and x^{b}_s allow both behavior because they can choose non-deterministically either to commit or to wait. Note that mes-

$$
\begin{array}{rrcl}
\mathsf{D} \equiv & state\langle A\rangle \,|\, put\langle \ell, A', C\rangle & \triangleright & commit\langle \ell \setminus \{lock\}, \ell, \{lock\}, C, A \cup A'\rangle \\
\wedge & state\langle A\rangle \,|\, abt\langle\rangle & \triangleright & failed\langle\rangle \,|\, \mathsf{release}\langle A\rangle \\
\wedge & commit\langle \{l\} \cup \ell, \ell', \ell'', C, A\rangle & \triangleright & commit\langle \ell, \ell', \ell'', C, A\rangle \,|\, l\langle \ell', lock, abt\rangle \\
\wedge & commit\langle \ell, \ell', \ell'', C, A\rangle \,|\, lock\langle \ell''', l, a\rangle & \triangleright & commit\langle \ell \cup (\ell''' \setminus \ell'), \ell' \cup \ell''', \ell'' \cup \{l\}, C, A \cup \{a\}\rangle \\
\wedge & commit\langle \emptyset, \ell, \ell, C, A\rangle & \triangleright & \mathsf{release}\langle C\rangle \\
\wedge & commit\langle \emptyset, \ell', \ell'', C, A\rangle \,|\, abt\langle\rangle & \triangleright & failed\langle\rangle \,|\, \mathsf{release}\langle A\rangle \\
\wedge & failed\langle\rangle \,|\, put\langle \ell, A', C\rangle & \triangleright & failed\langle\rangle \,|\, \mathsf{release}\langle A'\rangle \\
\wedge & failed\langle\rangle \,|\, lock\langle \ell, l, a\rangle & \triangleright & failed\langle\rangle \,|\, a\langle\rangle \\
\wedge & failed\langle\rangle \,|\, abt\langle\rangle & \triangleright & failed\langle\rangle
\end{array}
$$

Figure 12. The encoding of coordinators.

sages sent to merge names that are not used inside a negotiation are discarded when the thread commits, because they are useless outside contracts. In the encoding of a generalized `join` (with n participants) we abbreviate $D_1 \wedge \ldots \wedge D_n$ with $\bigwedge_i D_i$ and $P_1|\ldots|P_n$ with $\prod_i P_i$. In this case all threads are finished and the guarded process P is encoded using a new coordinator.

Finally, when a merge name x is defined more than once in a conjunction, redundant definitions for x_s^{k} are introduced. However, redundant definitions do not change the behavior of a process. Additionally, merge definitions are useless when appearing inside negotiations, because no sub-negotiations exist that can be merged. Hence, we omit their translation (the special symbol $\top$ denotes this fact).

Coordinator. Coordinators D in Figure 12, which are reused with minor variations from the encoding of ZS nets in Join [4], implement the D2PC, a variant of the ordinary two-phase commit protocol, where the role of the coordinator is played by all participants (it differs from the *decentralized* 2PC [1] because in D2PC the number of participants and their names are not statically fixed). We use the operation release which takes a set of messages and delivers them.

Roughly, the channel *state* records the messages that must be released in case of abort: (i) the channel that activates the compensation of the negotiation; and (ii) the list of ports abt_i of known participants. The commit protocol starts upon emission of the message $put\langle \ell, A', C\rangle$ (via a `join`, or `close`, or `drop`), which triggers a *commit* message (first rule of D). Each participant can also abort when it receives the message *abt*, which changes the modality of the coordinator to $failed\langle\rangle$ and releases the abort notification to any other known participant.

During the commit phase, messages on *commit* carry values $\langle \ell, \ell', \ell'', C, A\rangle$:

- ℓ records the set of known participants that must still be contacted;
- ℓ' stores the synchronization set of the thread (i.e. the list of known participants involved in the same transaction), which is typically augmented during the D2PC with the synchronization sets of other participants;
- ℓ'' records the parties who have already sent their consensus for commit;
- Sets C and A store the messages to be released in case of successful and unsuccessful completion, respectively.

The D2PC is based on the following steps performed by every participant:

(i) **first phase**. The participant sends a request to every thread in its own synchronization set (third rule of D). The message contains known participants.

(ii) **second phase**. The participant collects the messages sent by other parties and updates its own synchronization set (fourth rule of D). A request will be also sent to the new items in the synchronization set (by repeating (i) for them).

(iii) When the synchronization set is transitively closed, the commit protocol terminates locally and C is released (fifth rule of D).

(iv) If the participant transits in the state *failed*, it releases A, i.e. the compensation and the abort messages to known parties.

In the rest of this section we discuss the correctness and completeness of our encoding. Given a Join process P, $norm(P)$ denotes the process obtained by the repeated application of definitions in coordinators D until termination, i.e., completing the executions of the D2PC protocol. $norm(P)$ is defined for any P because the D2PC algorithm always terminates [4]. Moreover, we say $norm(P)$ stable, when it does not contain messages to ports *state*, i.e., all instances of the D2PC have finished either with the commit or abort of their participants. Hence, definitions in coordinators will never be used, and therefore they can be removed (for instance, as part of a garbage collection process). Moreover, when a negotiation aborts, $norm(P)$ can also contain messages sent by aborted negotiations (e.g. a negotiation sends $x\langle\vec{u}, p_i, a_i, \ell_i\rangle$ and then aborts), which can also be removed. We use $\underline{norm(P)}$ to denote the process obtained by removing garbage from a stable $norm(P)$. Note that $\underline{norm(P)}$ is *well-defined* only when all negotiations have finished.

The following results state that our encoding is correct and complete. We use the symbol $\approx$ to denote weak barbed bisimilarity [12].

Theorem 4.2 (Correctness) *Let P be a canonical flat process and $\vdash P : \Box_1$. If $P \rightarrow^*_{cJ} P'$ with $\vdash P' : \Box_1$, then $\exists Q$ s.t. $[\![P]\!]_{fn(P),\emptyset} \rightarrow^*_J Q$, and $\underline{norm(Q)} \approx [\![P']\!]_{fn(P'),\emptyset}$.*

Theorem 4.3 (Completeness) *Let P be a canonical flat process and $\vdash P : \Box_1$. If $[\![P]\!]_{fn(P),\emptyset} \rightarrow^*_J Q$ such that $\underline{norm(Q)}$ is well-defined, then $\exists P'$ s.t. $P \rightarrow^*_{cJ} P'$ and $[\![P']\!]_{fn(P'),\emptyset} \approx \underline{norm(Q)}$.*

Concluding remarks

cJoin is a conservative extension of the Join calculus coming equipped with few primitives for programming dynamic multi-party negotiations and their compensations. In this paper we show that flat cJoin processes can be implemented in Join in a fully distributed way. The result is achieved by first defining a type system for flat processes and proving the subject reduction property for it, then providing a canonical representative of flat processes that employs a few elementary definition patterns. Finally, it is shown that canonical representatives can be encoded in Join.

By Proposition 3.12, the encoding of ZS nets in cJoin produces processes in canonical form, which can therefore be encoded in Join by exploiting the implementation described in Section 4. We conjecture that the resulting encoding

$[\![\mathbf{def}\ [\![D]\!]_{\mathsf{cJ}}\ \mathbf{in}\ [\![P]\!]_{\mathsf{cJ}}]\!]$ is just a slightly redundant version of the direct translation in [4], but we leave as future work to spell out the formal details and proofs.

Finally, the results presented here suggest that full cJoin, including nested negotiations and compensations, can be modeled back in ordinary Join by further elaborating on the D2PC, but we leave this as a challenging future work.

References

[1] Bernstein, P., V. Hadzilacos and N. Goodman, "Concurrency, Control and Recovery in Database Systems," Addison-Wesley Longman, 1987.

[2] Bocchi, L., C. Laneve and G. Zavattaro, *A calculus for long-running transactions*, in: *Proceedings of FMOODS 2003*, Lect. Notes in Comput. Sci. (2003), to appear.

[3] *Business Process Execution Language for Web Services (*BPEL-WS*) ver.1.1*, (2003). `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel`.

[4] Bruni, R., C. Laneve and U. Montanari, *Orchestrating transactions in join calculus*, in: L. Brim, P. Jancar, M. Kretinsky and A. Kucera, editors, *Proceedings of CONCUR 2002*, Lect. Notes in Comput. Sci. **2421** (2002), pp. 321–336.

[5] Bruni, R., H. Melgratti and U. Montanari, *Nested commits for mobile calculi: extending Join* (2003), submitted manuscript. Available at the URL `http://www.di.unipi.it/~melgratt/publications/cjoin.ps`

[6] Bruni, R. and U. Montanari, *Zero-safe nets: Comparing the collective and individual token approaches*, Inform. and Comput. **156** (2000), pp. 46–89.

[7] Bruni, R. and U. Montanari, *Transactions and zero-safe nets*, in: H. Ehrig, G. Juhás, J. Padberg and G. Rozenberg, editors, *Advances in Petri Nets: Unifying Petri Nets*, Lect. Notes in Comput. Sci. **2128**, Springer Verlag, 2001 pp. 380–426.

[8] Butler, M., M. Chessell, C. Ferreira, C. Griffin, P. Henderson and D. Vines, *Extending the concept of transaction compensation*, IBM Systems Journal **41** (2002), pp. 743–758.

[9] Fournet, C., "The Join-Calculus: a Calculus for Distributed Mobile Programming," Ph.D. thesis, Ecole Polytechnique (1998).

[10] Fournet, C. and G. Gonthier, *The reflexive chemical abstract machine and the Join calculus*, in: *Proceedings of POPL'96* (1996), pp. 372–385.

[11] Fournet, C., G. Gonthier, J.-J. Lévy, L. Maranget and D. Rémy, *A calculus of mobile agents*, in: U. Montanari and V. Sassone, editors, *Proceedings of CONCUR'96*, Lect. Notes in Comput. Sci. **1119** (1996), pp. 406–421.

[12] Fournet, C. and C. Laneve, *Bisimulations in the join-calculus*, Theoret. Comput. Sci. **266** (2001), pp. 569–603.

[13] Roxburgh, U., *Biztalk orchestration: Transactions, exceptions, and debugging*, Microsoft BizTalk Server Technical Articles (2001). Available at the URL `http://msdn.microsoft.com/library/en-us/dnbiz/html/btsorch.asp?_r=1`.