Under consideration for publication in Math. Struct. in Comp. Science

cJoin: Join with communicating transactions[†]

ROBERTO BRUNI[1], HERNÁN MELGRATTI[2] and UGO MONTANARI[1]

[1] *Dipartimento di Informatica, Università di Pisa*
Largo Bruno Pontecorvo 3, I-56127 Pisa, Italy
Email: {ugo,bruni}@di.unipi.it
[2] *Departamento de Computación, FCEyN, Universidad de Buenos Aires - CONICET*
Pabellón I, Ciudad Universitaria, (C1428EGA) Buenos Aires, Argentina
Email: hmelgra@dc.uba.ar

Received 18 January 2011; Revised 28 October 2011

This paper proposes a formal approach to the design and programming of *Long Running Transactions* (LRT). We exploit techniques from process calculi to define cJoin, which is an extension of the Join calculus with few well-disciplined primitives for LRT. Transactions in cJoin are intended to describe the transactional interaction of several partners, under the assumption that any partner executing a transaction may communicate only with other transactional partners. In such case, the transactions run by any party are bound to achieve the same outcome (i.e., all succeed or all fail). Hence, a distinguishing feature of cJoin, called *dynamic joinability*, is that ongoing transactions can be merged to complete their tasks and when this happens either all succeed or all abort. Additionally, cJoin is based on compensations, i.e., partial executions of transactions are recovered by executing user-defined programs instead of providing automatic roll-back. The expressiveness and generality of cJoin is demonstrated by many examples addressing common programming patterns. The mathematical foundation is accompanied by a prototype language implementation, which is an extension of the JoCaml compiler.

Contents

[†] Research supported by the EU FET-GC2 IST-2004-16004 Integrated Project Sensoria, the Italian MIUR Project IPODS (PRIN 2008), the ANPCyT Project BID-PICT-2008-00319, and the UBACyT Project 20020090300122.

1. Introduction

The ultimate goal of *Service Oriented Architecture* (SOA) is to make it possible to develop new components and applications (now services) just by assembling existing ones. Many recent efforts, strongly pushed by large industrial consortia, have given birth to several (proposals for) programming/description languages tailored to the specification of web service integration, generally known as *web service composition languages* (WSCL), like XLANG (Thatte, 2001), WSFL (Leymann, 2001), WS-BPEL (BPEL, 2003), WS-CDL (WSCDL, 2004), WSCI (WSCI, 2002) and BPMN (BPMN, 2010). WSCLs address aggregation by following two complementary approaches:

— *Orchestration*: A composite service consists essentially of a unique program (usually known as *orchestrator*) that coordinates the execution of all components, while involved services are neither aware of the fact they are taking part in a larger process nor of its structure and goal. The application logic of a composite service relies on the orchestrator, which is responsible for interacting with (i.e., invoking) all involved components in the right order. For this reason, orchestration is appropriate for specifying intra-organisation (or private) processes, whose application details may be completely known and whose execution may be coordinated in a centralised way. Typical orchestration languages are XLANG, WSFL, and (executable processes of) WS-BPEL and BPMN.

— *Choreography*: Choreographies do not rely on a centralised coordinator since they are intended to facilitate the integration of business processes that spawn over different organisations. In this context, services are aware of the interaction protocol that underlies their composition, and thus of the way in which they should interact. For this reason, choreography languages allow for and focus on the definition of protocols that

parties should follow in order to achieve a common goal. There are two main approaches to define choreographies: (i) the *global model*, in which a protocol describes from a global perspective the messages exchanged by the parties, and (ii) the *interaction model* in which each service describes the temporal and logical dependencies among the messages it exchanges, i.e., a kind of interface definition. WS-CDL adopts the global model style, while WSCI and abstract processes of WS-BPEL are instances of the interaction model.

A common aspect considered by both orchestration and choreography styles is related to *long running transactions* (LRT), i.e., the possibility of executing some parts of a composed service atomically. Nevertheless, atomicity does not imply here the usual "all-or-nothing" property of database transactions, because perfect roll-back is unlikely in case of a fault. For example, the sending of a message cannot be undone. Consequently, LRTs often rely on a weaker notion of atomicity based on *compensations*. Compensations are ad hoc, user-programmed activities to be run when recovering from partial executions of LRTs arising after a fault or interruption because successful completion is no longer possible. For example, if some information has been sent and it is not longer valid after the fault, then a second message can be sent to the recipient.

Since most industrial standards lack rigourous foundations, many efforts have been spent to provide a formal basis to reason about LRTs in composition languages. As far as the orchestration of LRT is concerned, the first proposal that appeared in the literature is (to the best of our knowledge) StAC (Butler et al., 2002; Butler and Ferreira, 2004), which enriches an imperative language with primitives for installing, activating and removing compensations. After StAC, proposals such as (Butler et al., 2005b; Bruni et al., 2005; Bruni et al., 2011) have provided formal semantics for compensation languages, whose primitives are closer to real orchestration languages (see e.g. (Eisentraut and Spieler, 2009)).

A different line of research focuses on the formal definition of LRTs for interaction-based choreographies (Bocchi et al., 2003; Bruni et al., 2004; Lucchi and Mazzara, 2004; Laneve and Zavattaro, 2005; Caires et al., 2009; de Vries et al., 2010). Typically, such research thread consists of extending well-assessed mobile calculi with ad hoc constructs tailored to transactions and compensations. The enriched calculi exploit the communication primitives provided by, e.g., π-calculus (Milner et al., 1992) and join-calculus (Fournet et al., 1996), to model communication among parties. Hence, a composed service is described by a set of processes, any of them defining a particular partner of the complete system. In this way, any party declares the interface for proper composition with other partners. In this respect, transactional processes in those calculi are the formal counterpart of WSCI interfaces or BPEL abstract processes. Nevertheless, transactional calculi go beyond the scope of being just declarative definitions of service interfaces. In fact, they are aimed at providing an operational characterisation for business processes.

Consider the typical scenario in which a user books a room through a hotel reservation service. The ideal protocol followed by the two parties can be sketched as follow, by using an informal π-calculus-like notation:

$$\mathsf{Client} \equiv \mathit{request}!(\mathit{data}).\mathit{offer}?(\mathit{price}).\mathit{accept}!(\mathit{cc})$$
$$\mathsf{Hotel} \equiv \mathit{request}?(\mathit{details}).\mathit{offer}!(\mathit{rate}).\mathit{accept}?(\mathit{card})$$

We write $a!(v)$ for the sending of the message v on the communication channel a, and $a?(x)$ for receiving on the variable x some message sent on the channel a. The prefix symbol "$\_.\_$" must not be confused with the usual dot-notation from object-oriented language: it is used to establish the order in which actions must be executed. The client starts by sending a booking request to the hotel, which answers it with a rate offer. After receiving the offer, the client accepts it. This is the ideal protocol both parties should follow in order to accomplish the common goal. Nevertheless, there are several situations in which parties may be forced/inclined not to complete the execution of the protocol (e.g., the hotel has no available rooms for the requested day, or the client does not obtain acceptable rates). Clearly, just stopping the execution of the protocol may be not acceptable in most of the cases. Compensable transactions are designed to handle this kind of situations. In addition to the usual primitives of name passing calculi, transactional calculi provide a new kind of terms, generally of the form $[P : Q]$, involving: (i) a process P that is required to be executed until completion and (ii) the corresponding compensation Q to be executed in case P cannot complete successfully. Moreover, the cancellation of the transaction can be handled by making P reach a special process, usually denoted by *abort*. For example, when the hotel is unable to proceed with the order, it may abort the transaction and use the compensation to suggest an alternative hotel to the client (e.g., by sending the message $\mathit{alt}!(\mathit{hotel})$). Then, the description of the protocol could be improved as follow

$$\mathsf{Client} \equiv [\mathit{request}!(\mathit{data}).\mathit{offer}?(\mathit{price}).(\mathit{accept}!(\mathit{cc}) + \mathit{abort}) : \mathit{alt}?(h).Q]$$
$$\mathsf{Hotel} \equiv [\mathit{request}?(\mathit{details}).(\mathit{offer}!(\mathit{rate}).\mathit{accept}?(\mathit{card}) + \mathit{abort}) : \mathit{alt}!(\mathit{hotel})]$$

The above protocol allows also the client to abort the transaction after receiving an offer (for instance when the offer does not satisfy her expectations). Alternatively, the hotel may abort after receiving a request (for instance when no rooms are available). Clearly, more sophisticated protocols may be written to allow clients and hotels to abort at any moment. We illustrate the use of compensations by making the component Hotel generate the single message $\mathit{alt}!(\mathit{hotel})$ to provide the client with the information of an alternative hotel to contact running Q. More concretely, *hotel* could be a tuple of channel names $\mathit{request}', \mathit{offer}', \mathit{accept}'$ to contact another hotel and Q could be just a recursive instance of Client parametric on such channels.

Proposals in the literature differ mainly in the kinds of interactions allowed across transaction boundaries and the effects associated with the handshaking. Roughly, on one side of the design option we have completely permeable transactional scopes (as in πt-calculus (Bocchi et al., 2003)), where messages may freely cross transactional boundaries. For instance, a possible computation for the protocol can be described as follows

$$\begin{array}{rcl}
\mathsf{Client}|\mathsf{Hotel} & \to & [\mathit{offer}?(\mathit{price}).(\mathit{accept}!(\mathit{cc}) + \mathit{abort}) : \mathit{alt}?(h).Q] \\
& & \qquad\qquad | \; [\mathit{offer}!(\mathit{rate}).\mathit{accept}?(\mathit{card}) + \mathit{abort} : \mathit{alt}!(\mathit{hotel})] \\
& \to & [(\mathit{accept}!(\mathit{cc}) + \mathit{abort}) : \mathit{alt}?(h).Q] \mid [\mathit{accept}?(\mathit{card}) : \mathit{alt}!(\mathit{hotel})] \\
& \to & [0 : \mathit{alt}?(h).Q] \mid [0 : \mathit{alt}!(\mathit{hotel})] \\
& \to & 0
\end{array}$$

Messages in the above computation flows freely from one transaction to the others. The main drawback of such approach is that transactional scopes does not ensure all interacting transactions to reach the same result, i.e., some of them may commit even though others have failed. For instance, consider a client executing the following protocol

$$\mathsf{Client}' \equiv [\mathit{request}!(\mathit{data}).\mathit{offer}?(\mathit{price}).(\mathit{accept}!(\mathit{cc})|\mathit{abort}) : \mathit{alt}?(h).Q]$$

Analogously to the previous case, there is a computation leading the system to the following state

$$\mathsf{Client}'|\mathsf{Hotel} \to^* [(\mathit{accept}!(\mathit{cc})|\mathit{abort}) : \mathit{alt}?(h).Q] \mid [\mathit{accept}?(\mathit{card}) : \mathit{alt}!(\mathit{hotel})]$$

Then the system may evolve to

$$\mathsf{Client}'|\mathsf{Hotel} \to^* [\mathit{abort} : \mathit{alt}?(h).Q] \mid [0 : \mathit{alt}!(\mathit{hotel})]$$

At this point one transaction (that one from the client) can only abort by releasing the compensation Q, while the other (the hotel party) can only commit. Hence, the hotel has reserved a room that the client is not willing to book. It is true that we can write a different compensation for the client that contains the code needed for making the hotel cancel the reservation. For example, graceful termination mechanisms for closing dyadic sessions have been studied in (Boreale et al., 2008) and can be likely reused for transactions. Nevertheless, from our point of view, the fact that involved participants have no guaranties about the final outcome of the remaining transactional participants provides too weak a transactional mechanism for handling many common situations. Although stronger transactional properties may be ensured by programming ad hoc coordination code through compensations, suitable transactional primitives should relieve programmers from writing such kind of code.

In this paper we present *Committed Join* (cJoin), a calculus designed to ensure that all participants of the same transaction reach the same agreed outcome. This is achieved by making interacting transactional processes become part of the same larger transaction. The cJoin is an extension of the Join calculus (Fournet and Gonthier, 1996), which is a process calculus with asynchronous name-passing communication. We based our approach on the Join calculus rather than on other more popular process calculi, such as the π-calculus, because Join adheres to a locality principle that guarantees that extruded names cannot be used in input by the process who received them (they can only output values on such ports). This feature is crucial for deploying distributed implementations and it is not enforced in the full π-calculus. Moreover, it allows to obtain precise characterisations of transaction termination and atomic joining of multiple transactions, which are missing from most alternative proposals in the literature. Another advantage is that

the operational semantics rules are quite simple and compact when compared to other transactional calculi.

The process $\mathsf{Client}'|\mathsf{Hotel}$ behaves in cJoin as follows. When both transactions communicate through the port *request* for the first time they are merged in a unique larger transaction, whose transactional process and compensation correspond respectively to the parallel composition of the residuals of the original transactions and to the parallel composition of the original compensations, as shown below

$$\begin{array}{rcl} \mathsf{Client}'|\mathsf{Hotel} & \to & [\mathit{offer}?(\mathit{price}).(\mathit{accept}!(\mathit{cc}) \mid \mathit{abort}) \\ & & \quad \mid (\mathit{offer}!(\mathit{rate}).\mathit{accept}?(\mathit{card}) + \mathit{abort}) : \mathit{alt}?(h).Q \mid \mathit{alt}!(\mathit{hotel})] \end{array}$$

From this moment on, the system may evolve as usual. In particular, assuming the hotel sends an offer (1) and the client sends the confirmation (2), the system moves as follows

$$\begin{array}{rcll} & \to & [\mathit{accept}!(\mathit{cc}) \mid \mathit{abort} \mid \mathit{accept}?(\mathit{card}) : \mathit{alt}?(h).Q \mid \mathit{alt}!(\mathit{hotel})] & (1) \\ & \to & [\mathit{abort} : \mathit{alt}?(h).Q \mid \mathit{alt}!(\mathit{hotel})] & (2) \end{array}$$

In this case both original transactions are bound together and none of them has already committed, therefore the abort condition reached by the client causes the hotel transaction to be compensated as well. In this way, transactional scopes of cJoin ensure that all parties of a transaction commit (resp. abort) only when all other parties commit (resp. abort), although each party is responsible for defining its own compensation. Note that the transactional primitive in cJoin relieves programmers from coding protocols needed to agree on a common result for a distributed transaction, while leaving to the programmer the responsibility for defining suitable compensations to recover aborted transactions. Though no automatic roll-back mechanism is provided, it is obvious that restoring the initial process upon the abort can be straightforwardly programmed by recursive definitions like $P \equiv [Q : P]$, easy to implement in cJoin syntax.

Another important issue addressed in this paper is *transaction serializability*. Not to be confused with object serializability, it is a way for ensuring the correctness of reasoning at different levels of abstractions, in which transactions become atomic reductions when seen at the abstract level. Let us consider a set of n transactions $\{T_i \mid 1 \leq i \leq n\}$, each consisting of several activities to be carried out. Their concurrent execution $T_1 \parallel \ldots \parallel T_n$ can interleave the activities from different transactions and it is said to be *serializable* if there exists a sequence $T_{i_1};\ T_{i_2}; \ldots ; T_{i_n}$ that executes all transactions one at a time (without interleaving their steps) and produces the same result (Bernstein et al., 1987). More generally, in the case of nested transactions, each T_i could involve recursively several sub-transactions $T_{i,1}, ..., T_{i,n_1}$ among the activities to be carried out, whose execution is possibly interleaved with those of other transactions and of their sub-transactions. Serializability is important because it allows to reason about the behaviour of a system by considering one transaction at a time, at any given level of nesting. Transaction serializability is generally difficult to achieve in other proposals where communication is allowed across transactions. Here we show that, for a large class of cJoin processes, called *shallow processes*, serializability is guaranteed by construction because if two separately initiated transactions interact, then their scopes are merged together as part of the same transaction, i.e. after merge they cannot commit or abort independently.

For the prototype implementation of cJoin we rely on available distributed implementation of Join. In fact, the primitives of Join have been exploited in the design of Jocaml (Conchon and Le Fessant, 1999), an extension of the *Objective Caml*, a functional language with support of object-oriented and imperative paradigms, and Polyphonic $C^{\#}$ (Benton et al., 2002) (later Cω) that extends $C^{\#}$ with asynchronous methods and synchronisation patterns, called *chords*. We take advantage of this fact for extending JoCaml with transactional primitives. The resulting language, called *transactional* JoCaml (t-JoCaml), adds to JoCaml the possibility of writing programs that should execute as compensable transactions in the style of cJoin transactions.

Paper Outline. After introducing some preliminaries (Section 2) we give the syntax and semantics of cJoin (Section 3) and describe several examples illustrating the main features of cJoin (Section 4). By exploiting the strategy used for implementing cJoin transactions in Join itself, as summarised in Section 5, in Section 6 we describe t-JoCaml as an extension of JoCaml. We remark that t-JoCaml actually implements a sub-calculus of cJoin (called *flat*) in which transactions cannot be nested (see Section 3.2). Section 6.2.2 describes the corresponding extension of the JoCaml compiler we have realised. Section 7 shows how transaction serializability can be achieved in cJoin. To conclude we compare our proposal against several approaches appeared in the literature and we present some final remarks (Section 8).

Preliminary studies on cJoin have been presented at IFIP-TCS 2004–IFIP 18th World Computer Congress, 3rd International Conference on Theoretical Computer Science, and COMETA 2003–Workshop of the COMETA Project on Computational Metamodels, after which several other proposals of transactional process calculi emerged in the literature. Yet the features of cJoin remained quite peculiar and this work integrates previous studies with new perspectives in the area of service-oriented programming and business processes, most notably the well-disciplined use of compensations.

2. Preliminaries

In this section we report on the operational semantics of the Join calculus as a chemical abstract machine, by following the presentation of (Fournet and Gonthier, 1996).

2.1. *The Chemical Abstract Machine*

The semantics of the Join calculus relies on the *reflexive chemical abstract machine* (CHAM). In a CHAM (Berry and Boudol, 1992) computation states S (called *solutions*) are finite multisets of terms m (called *molecules*), and computations are multiset rewrites. Multisets are denoted by $m_1, \ldots, m_n$ and abbreviated with $\oplus_i\ m_i$. Solutions can be structured in a hierarchical way by using the operator *membrane* $\{\!|\cdot|\!\}$ to group a solution S into a molecule $\{\!|S|\!\}$. (In (Berry and Boudol, 1992) molecules can be built also with the constructor *airlock*, but it is not needed in our presentation.)

Transformations are described by a set of *chemical rules*, which specify how solutions react. In a CHAM there are two different kinds of chemical rules: *heating / cooling* rules $S \rightleftharpoons S'$ representing syntactical rearrangements of molecules in a solution, and *reac-*

$$\frac{m_1, \ldots, m_k \to m'_1, \ldots, m'_l \ \in \text{ set of CHAM rules}}{m_1\sigma, \ldots, m_k\sigma \to m'_1\sigma, \ldots, m'_l\sigma}\ (\text{REACTION LAW}) \qquad \frac{S \to S'}{S, S'' \to S', S''}\ (\text{CHEMICAL LAW}) \qquad \frac{S \to S'}{\{\![S]\!\} \to \{\![S']\!\}}\ (\text{MEMBRANE LAW})$$

Figure 1. CHAM laws.

tion rules $S \to S'$. Heating / cooling rules are analogous to the axioms for structural congruence in process calculi, and thus called also *structural* rules. Structural rules are reversible: a solution obtained by applying a cooling rule can be heated back to the original state, and vice versa. Reaction rules, on the other hand, cannot be undone. Rules can carry formal parameters to be matched against actual parameters in the redex and substituted in the right-hand side.

The laws governing CHAM computations are in Figure 1 (we give them for reaction rules, but they are applicable to heating / cooling rules as well):

— *Reaction law*: Given a rule, an instance of its left-hand-side can be replaced by the corresponding instance of the right-hand-side. The substitution σ replaces the formal parameters with the actual parameters by matching the solution against the left-hand side of the rule.
— *Chemical law*: Reactions can be applied in every larger solution
— *Membrane Law*: Reactions may occur at any level in the hierarchy of solutions

Note that CHAM's heating / cooling / reaction rules have no premises and are purely local. They specify only the part of the solution that actually changes. Moreover, since solutions are multisets, not overlapping rules can be applied concurrently.

2.2. *The Join calculus*

The Join calculus relies on an infinite set of names $x, y, u, v, \ldots$. Name tuples are written $\vec{u}$. Join processes, definitions and patterns are in Figure 2(a). A *process* is either the inert process 0, the asynchronous emission $x\langle\vec{y}\rangle$ of message $\vec{y}$ on port x, the process **def** D **in** P equipped with local ports defined by D, or a parallel composition of processes $P|Q$. A *definition* is a conjunction of elementary reactions $J \triangleright P$ that associate *join-patterns* J with *guarded processes* P. Names defined by D in **def** D **in** P are bound in P and in all the guarded processes contained in D. The sets of defined names dn, received names rn and free names fn are in Figure 2(b).

Example 1. Consider the processes $Q =$ **def** $proxy\langle y\rangle \triangleright server\langle proxy, y\rangle$ **in** $proxy\langle a\rangle$ and $P = server\langle proxy, b\rangle|Q$. Roughly, Q defines a local port *proxy* such that when a message on *proxy* arrives with any content y then the name *proxy* is extruded on (the elsewhere defined, free port) *server* together with y. Intuitively, the "local proxy" forwards to the "public server" each message tagged with its "origin" (i.e., the name of the proxy). The initial state of Q carries a message on *proxy* whose content is a. The process P places Q in a context that includes a message to port *server* with content $\langle proxy, b\rangle$. Then, $fn(Q) = \{server, a\}$ and $fn(P) = \{server, a, b, proxy\}$. Therefore the name *server* has common meaning in P and Q, while the symbol *proxy* denotes, by accident, different

$$
\begin{array}{lrcl}
(\textsc{proc}) & P, Q & ::= & 0 \mid x\langle \vec{y} \rangle \mid \mathbf{def}\ D\ \mathbf{in}\ P \mid P|Q \\
(\textsc{def}) & D, E & ::= & J \triangleright P \mid D \wedge E \\
(\textsc{pat}) & J, K & ::= & x\langle \vec{y} \rangle \mid J|K
\end{array}
$$

(a) Syntax

$$
\begin{array}{rcl@{\qquad}rcl}
(\textsc{Free}) & & & & & \\
fn(0) & = & \emptyset & fn(x\langle \vec{y} \rangle) & = & \{x\} \cup \{\vec{y}\} \\
fn(\mathbf{def}\ D\ \mathbf{in}\ P) & = & (fn(P) \cup fn(D)) \backslash dn(D) & fn(P|Q) & = & fn(P) \cup fn(Q) \\
fn(J \triangleright P) & = & dn(J) \cup (fn(P) \backslash rn(J)) & fn(D \wedge E) & = & fn(D) \cup fn(E) \\
(\textsc{Defined}) & & & & & \\
dn(J \triangleright P) & = & dn(J) & dn(D \wedge E) & = & dn(D) \cup dn(E) \\
dn(x\langle \vec{y} \rangle) & = & \{x\} & dn(J|K) & = & dn(J) \cup dn(K) \\
(\textsc{Received}) & & & & & \\
rn(x\langle \vec{y} \rangle) & = & \{\vec{y}\} & rn(J|K) & = & rn(J) \cup rn(K)
\end{array}
$$

(b) Free, Defined and Received names

$$
\begin{array}{ll}
(\textsc{str-null}) & 0 \leftrightharpoons \\
(\textsc{str-join}) & P \mid Q \leftrightharpoons P, Q \\
(\textsc{str-and}) & D \ \wedge \ E \leftrightharpoons D, E \\
(\textsc{str-def}) & \mathbf{def}\ D\ \mathbf{in}\ P \leftrightharpoons D\sigma_{dn(D)}, P\sigma_{dn(D)} \qquad (range(\sigma_{dn(D)}) \text{ globally fresh}) \\
(\textsc{red}) & J \triangleright P, J\sigma \rightarrow J \triangleright P, P\sigma
\end{array}
$$

(c) Semantics

Figure 2. Join Calculus.

ports in $server\langle proxy, b \rangle$ and Q: a free (elsewhere defined) port in the former and a private port in the latter. Moreover, letting $D = proxy\langle y \rangle \triangleright server\langle proxy, y \rangle$, we have $fn(D) = \{proxy, server\}$, $dn(D) = \{proxy\}$ and $rn(D) = \{y\}$.

The semantics of the Join calculus relies on the *reflexive* CHAM. It is called reflexive because active reaction rules are represented by molecules present in solutions, which are activated dynamically. Molecules, generated by $m \ ::= \ P \ \mid \ D$, correspond to terms of the Join calculus denoting processes or definitions. The chemical rules are shown in Figure 2(c). Rule STR-NULL states that 0 can be added or removed from any solution. Rules STR-JOIN and STR-AND implies the associativity and commutativity of $\mid$ and $\wedge$, because $\_\,,\_$ is such. STR-DEF denotes the activation of a local definition, which implements a static scoping discipline by properly renaming defined ports by *globally fresh* names. A name x is *fresh* w.r.t. a process P (resp. a definition D) if $x \notin fn(P)$ (resp. $x \notin fn(D)$). Moreover, x is fresh w.r.t. a solution s if it is fresh w.r.t. every term in s. A set of names X is fresh if every name in X is such. We write the substitution of names $x_1 \ldots x_n$ by names $y_1 \ldots y_n$ as $\sigma = \{^{y_1 \ldots y_n}/_{x_1 \ldots x_n}\}$, with $dom(\sigma) = \{x_1, \ldots, x_n\}$ and $range(\sigma) = \{y_1, \ldots, y_n\}$. We indicate with σ_N an injective substitution σ such that $dom(\sigma) = N$. When we require names to be globally fresh, we mean that they must be different from all other names appearing in the enclosing context.

Example 2. Consider the process P in the previous example. A solution s containing only P, i.e., $s = \{\!|P|\!\}$, may be heated as follows. First, the two parallel agents are separated by obtaining $\{\!|server\langle proxy, b\rangle, \mathbf{def}\ proxy\langle y\rangle \triangleright server\langle proxy, y\rangle\ \mathbf{in}\ proxy\langle a\rangle|\!\}$. Now the second molecule contains a definition of a local port *proxy* that is different from the homonymous free port *proxy* in the first molecule $server\langle proxy, b\rangle$. Hence, when using STR-DEF for separating the local definition from the corresponding process, the local definition of *proxy* must be renamed by using a fresh name, say $proxy_1$, obtaining the solution $s' = \{\!|server\langle proxy\rangle, proxy_1\langle y\rangle \triangleright server\langle proxy_1, y\rangle, proxy_1\langle a\rangle|\!\}$.

Finally, RED describes the use of an active reaction rule $(J \triangleright P)$ to consume messages forming an instance of J (for a suitable substitution σ, with $dom(\sigma) = rn(J)$), and produce a new instance $P\sigma$ of its guarded process P.

Example 3. By applying rule (RED) to the solution s' from Example 2 for $\sigma = \{^a/_y\}$, we get $s' \to \{\!|server\langle proxy, b\rangle, proxy_1\langle y\rangle \triangleright server\langle proxy_1, y\rangle, server\langle proxy_1, a\rangle|\!\}$. Note that the local port $proxy_1$ has been extruded on the free channel *server*.

Remark 2.1. For π-calculus enthusiasts, the Join calculus can be easily grasped by considering the main differences enforced by the syntax, namely: (i) like in the asynchronous π-calculus, only output particles are allowed, not output prefixes; (ii) input prefixes are encoded as definitions and joint inputs are allowed (i.e. more than one message can be consumed atomically); (iii) all definitions are persistent, as if they were prefixed by the replication operator; (iv) the definition construct **def** D **in** P is also binding all defined names in D to ensure the unique receptor property and favour distributed implementation (names can still be extruded, but the processes that receive them can only output on them); (v) the programming style is continuation passing, in the sense that output prefixes can be encoded by including in the message a fresh continuation name k where the acknowledge of the receipt must be sent to activate the output-prefixed process. For example, the process P from Example 1 can be understood as the π-calculus process $\overline{s}\langle p, b\rangle \mid (\nu p)(\ \overline{p}a \mid !p(y).\overline{s}\langle p, y\rangle\ )$, where for brevity initial letters of port names are used.

We shall write $P \equiv Q$ when $P \leftrightharpoons^* R$, for $\leftrightharpoons^*$ the reflexive and transitive closure of the relation $\leftrightharpoons$. Moreover, we abuse the notation by allowing one step reductions up to $\equiv$, i.e. writing $P \to Q$ when $P \leftrightharpoons^* \to \leftrightharpoons^* Q$. We write $P \to^n Q$ for $n \geq 0$ if there exist $n+1$ processes $P_0, ..., P_n$ such that $P \equiv P_0 \to P_1 \to ... \to P_{n-1} \to P_n \equiv Q$. Finally, we write $P \to^* Q$ if $P \to^n Q$ for some n.

Note that by exploiting heating and cooling rules, it is always possible to move definitions around, possibly after some remaining of their defined ports. In particular, we remark that **def** D **in** $P \equiv P|$**def** D **in** 0 whenever $fn(P) \cap dn(D) = \emptyset$ and furthermore that (**def** D_1 **in** P_1)|(**def** D_2 **in** P_2) $\equiv$ **def** D_1 **in** (P_1|**def** D_2 **in** P_2) $\equiv$ **def** $D_1 \wedge D_2$ **in** $(P_1|P_2)$ if $dn(D_1) \cap (dn(D_2) \cup fn(P_2)) = dn(D_2) \cap (dn(D_1) \cup fn(P_1)) = \emptyset$.

In Sections 3 and 4, where several examples of cJoin processes are given, we shall use the following syntactic sugar for the case where different decisions can be taken after receiving a certain message tuple: we write $J \triangleright P + Q$ in place of $J \triangleright P \wedge J \triangleright Q$.

$$
\begin{array}{lrcl}
(\text{MESS}) & M, N & ::= & 0 \mid x\langle \vec{y} \rangle \mid M|N \\
(\text{PROC}) & P, Q & ::= & M \mid \mathbf{def}\ D\ \mathbf{in}\ P \mid P|Q \mid [P:Q] \mid \mathit{abort} \\
(\text{DEF}) & D, E & ::= & J \triangleright P \mid D \wedge E \mid J \blacktriangleright P \\
(\text{PAT}) & J, K & ::= & x\langle \vec{y} \rangle \mid J|K
\end{array}
$$

Figure 3. Syntax of cJoin.

3. Committed Join

In order to handle LRTs we extend the syntax of the Join calculus as shown in Figure 3. For convenience we introduce the syntactical category M of processes without definitions, i.e., a parallel composition of messages. In addition to cJoin processes, we add terms $[P : Q]$ to denote transactions, where P is the transactional process and Q is its compensation, i.e., the process to be executed when P aborts. A transactional process is executed in isolation until reaching either a commit state or an abort condition. If P commits, the obtained result is delivered to the outside of the transaction. Otherwise, the compensation Q is activated. The abort decision is caused by the presence of the special basic process *abort*.

A new kind of definitions $J \blacktriangleright P$, called *merge definitions*, is introduced to specify the possible interactions among transactions (inter-transaction communication). Merge definitions allow the consumption of messages produced in the scope of different transactions by joining all participants in a unique larger LRT. Note that all ongoing transactions that want to merge are treated uniformly: they will issue a request to $J \blacktriangleright P$ by producing an instance of a particle in J and when a full instance $J\sigma$ can be formed out of those particles all the transactions containing those particles are merged. Thus $J \blacktriangleright P$ acts as some kind of *message board* where merge advertises are posted by transactions. This is different w.r.t. an "asymmetric" merge discipline, where one message in a transaction is received by an input running in a different transaction causing the merge. In fact, this kind of permeability of transaction scopes is not allowed in cJoin. Moreover, cJoin definitions can be used to create transactions dynamically. For instance, by firing $J \triangleright [P : Q]$ a new instance of the transaction P with compensation Q is activated. Perfect roll-back can thus be programmed just by writing definitions like $J \triangleright [P : J]$. In fact, if an instance $J\sigma$ of J is consumed to produce $[P\sigma : J\sigma]$ such that $P\sigma$ will abort, then the original messages $J\sigma$ are restored as a compensation. Notably, when two or more of such transactions are merged into a unique transaction, then the overall compensation becomes just the parallel composition of their perfect roll-backs.

The sets of defined names dn, received names rn and free names fn are defined in Figure 4. In particular, we distinguish between defined *ordinary* names $dn_{\mathrm{o}}(D)$ and defined *merge* names $dn_{\mathrm{m}}(D)$ that, as a general well-formedness discipline, are always assumed to be disjoint sets of names.

(PROC)

$$\begin{aligned} fn(0) &= \emptyset & fn(x\langle\vec{y}\rangle) &= \{x\} \cup \{\vec{y}\} \\ fn(P|Q) &= fn(P) \cup fn(Q) & fn(abort) &= \emptyset \\ fn(\textbf{def } D \textbf{ in } P) &= (fn(P) \cup fn(D)) \backslash dn(D) & fn([P:Q]) &= fn(P) \cup fn(Q) \end{aligned}$$

(DEF)

$$\begin{aligned} fn(J \triangleright P) &= dn(J) \cup (fn(P) \backslash rn(J)) & fn(D \wedge E) &= fn(D) \cup fn(E) \\ fn(J \blacktriangleright P) &= dn(J) \cup (fn(P) \backslash rn(J)) \\ dn_{\mathrm{o}}(J \triangleright P) &= dn(J) & dn_{\mathrm{o}}(D \wedge E) &= dn_{\mathrm{o}}(D) \cup dn_{\mathrm{o}}(E) \\ dn_{\mathrm{o}}(J \blacktriangleright P) &= \emptyset \\ dn_{\mathrm{m}}(J \triangleright P) &= \emptyset & dn_{\mathrm{m}}(D \wedge E) &= dn_{\mathrm{m}}(D) \cup dn_{\mathrm{m}}(E) \\ dn_{\mathrm{m}}(J \blacktriangleright P) &= dn(J) \end{aligned}$$

(PAT)

$$\begin{aligned} rn(x\langle\vec{y}\rangle) &= \{\vec{y}\} & rn(J|K) &= rn(J) \cup rn(K) \\ dn(x\langle\vec{y}\rangle) &= \{x\} & dn(J|K) &= dn(J) \cup dn(K) \end{aligned}$$

Figure 4. Free, defined and received names.

3.1. *Operational Semantics*

The operational semantics of cJoin is given in the reflexive CHAM style. Molecules m and solutions S for cJoin are as follow

$$\begin{aligned} m &::= P \mid D \mid \llcorner P \lrcorner \mid \{\!|S|\!\} \\ S &::= m \mid m, S \end{aligned}$$

Processes and definitions are molecules. Terms $\llcorner Q \lrcorner$ denote compensations, i.e., frozen processes to be activated only when the corresponding transaction aborts. Molecules $\{\!|S|\!\}$ stands for running transactions.

The chemical rules for cJoin are given in Figure 5. The first five chemical rules are the ordinary ones for Join. Rule (STR-TRANS) states that a term denoting a transaction corresponds to a sub-solution consisting of two molecules: the transactional process P and its compensation Q, which is frozen (the operator $\llcorner . \lrcorner$ forbids the enclosed process to compute because there is no rule for applying reductions inside it).

A transaction can commit only when all internal computations have finished. This situation can be characterised as the special pattern $\{\!|M|\textbf{def } D \textbf{ in } 0, \llcorner Q \lrcorner|\!\}$, comprising some messages M, the definition D for computing inside the transaction and the so-far installed compensation Q. Since the scope of D is the nil process, it is evident that at commit time, private definitions of the transactional process can be discarded, because neither the messages that are being released contain those names nor they could have been extruded previously. On the other hand, if defined names of D were present in some of the messages inside the transaction, then the transactional activity would not yet be complete and the commit would not take place. When a transaction commit (rule (COMMIT)), the local resources M produced inside a transaction are released as outcome. After commit, its compensation procedure $\llcorner Q \lrcorner$ is useless and it is discarded as well.

The abortion of a transaction is handled by the rule (ABORT), which releases Q whenever *abort* is present in the solution.

Interactions among transactions are dealt with (MERGE), where we let the notation $\Pi_i J_i$ and $\Pi_i Q_i$ denote, respectively, the parallel composition of messages $J_1|...|J_n$ and the parallel composition of processes $Q_1|...|Q_n$. We also recall that the notation $\otimes_i m_i$ denotes multisets of molecules and solutions. The rule (MERGE) consumes messages from different transactions and creates a larger transaction by combining the definitions and messages of the original ones with a new instance of the guarded process $P\sigma$, where $dom(\sigma) = rn(J_1|\ldots|J_n)$. Name clashes are avoided because we assume that STR-DEF generates globally fresh names. The compensation for the joint transaction is the parallel composition of all the original compensations.

The rule (MERGE) is quite general, as it can be used to join atomically an unbounded number of ongoing transactions. To help the understanding, we show a few particular instances of the rule (MERGE), which will be used frequently in the rest of this paper.

The first case is that of a trivial merge, where a unique transaction is involved. It is interesting to show it just to clarify that rule (MERGE) can be applied without merging transactions. This kind of merge rules works as global definitions that can be used in any transaction of the system.

(TRIV-MERGE)
$$(x\langle y\rangle \blacktriangleright P), \{\!|x\langle v\rangle, S, \llcorner Q \lrcorner|\!\} \quad \rightarrow \quad (x\langle y\rangle \blacktriangleright P), \{\!|S, P\{^{v}/_{y}\}, \llcorner Q \lrcorner|\!\}$$

The second case is that of a two-way merge, which will be very useful in our examples.

(TWO-MERGE)
$$\begin{array}{l}(x_1\langle y_1\rangle | x_2\langle y_2\rangle \blacktriangleright P),\\ \{\!|x_1\langle v_1\rangle, S_1, \llcorner Q_1 \lrcorner|\!\},\\ \{\!|x_2\langle v_2\rangle, S_2, \llcorner Q_2 \lrcorner|\!\}\end{array} \quad \rightarrow \quad (x_1\langle y_1\rangle | x_2\langle y_2\rangle \blacktriangleright P), \{\!|S_1, S_2, P\{^{v_1,v_2}/_{y_1,y_2}\}, \llcorner Q_1 | Q_2 \lrcorner|\!\}$$

Note that, as a degenerate case of rule (TWO-MERGE) we also have:

(TWO-MERGE-DEG)
$$\begin{array}{l}(x_1\langle y_1\rangle | x_2\langle y_2\rangle \blacktriangleright P),\\ \{\!|x_1\langle v_1\rangle, x_2\langle v_2\rangle, S, \llcorner Q \lrcorner|\!\}\end{array} \quad \rightarrow \quad (x_1\langle y_1\rangle | x_2\langle y_2\rangle \blacktriangleright P), \{\!|S, P\{^{v_1,v_2}/_{y_1,y_2}\}, \llcorner Q \lrcorner|\!\}$$

The following proposition states that Join is a sub-calculus of Join.

Proposition 3.1. Join is a sub-calculus of cJoin.

Proof. It is obvious from the syntax that any Join process is also a cJoin process. It remains to show that: (1) for any Join processes P and Q, if $P \rightarrow Q$ in Join, then $P \rightarrow Q$ in cJoin; and (2) for any Join processes P, if $P \rightarrow Q$ in cJoin, then Q is a Join process and $P \rightarrow Q$ in Join. Both implications follow straightforwardly from the fact that the chemical rules of cJoin can be partitioned in two sets: one consisting exactly of the chemical rules of Join, and the other containing structural rules involving non-Join operators on both sides and reaction rules involving non-Join operators in the left-hand side. □

We are now ready to revisit the hotel booking problem described in Section 1 and show how it can be modelled in cJoin.

$$
\begin{array}{lrcl}
(\textsc{str-null}) & 0 & \leftrightharpoons & \\
(\textsc{str-join}) & P \mid Q & \leftrightharpoons & P, Q \\
(\textsc{str-and}) & D \wedge E & \leftrightharpoons & D, E \\
(\textsc{str-def}) & \mathbf{def}\ D\ \mathbf{in}\ P & \leftrightharpoons & D\sigma_{dn(D)}, P\sigma_{dn(D)} \\
 & & & \quad {\scriptstyle (range(\sigma_{dn(D)})\ globally\ fresh)} \\
(\textsc{red}) & J \triangleright P, J\sigma & \to & J \triangleright P, P\sigma \\
(\textsc{str-trans}) & [P : Q] & \leftrightharpoons & \{\!|P, \llcorner Q \lrcorner|\!\} \\
(\textsc{commit}) & \{\!|M|\mathbf{def}\ D\ \mathbf{in}\ 0, \llcorner Q \lrcorner|\!\} & \to & M \\
(\textsc{abort}) & \{\!|abort|P, \llcorner Q \lrcorner|\!\} & \to & Q \\
(\textsc{merge}) & & & \\
 & (\Pi_i J_i \blacktriangleright P), \otimes_i \{\!|J_i\sigma, S_i, \llcorner Q_i \lrcorner|\!\} & \to & (\Pi_i J_i \blacktriangleright P), \{\!|\otimes_i S_i, P\sigma, \llcorner \Pi_i Q_i \lrcorner|\!\}
\end{array}
$$

Figure 5. Operational semantics of cJoin.

$$
\begin{array}{lcl}
\mathsf{HB} & \equiv & \mathbf{def}\ \ \mathit{HotelSrv}\langle r\rangle \mid \mathit{HotelReq}\langle d, \kappa\rangle \ \blacktriangleright\ r\langle d, \kappa\rangle \\
 & & \mathbf{in}\ \ \mathsf{H} \mid \mathsf{C} \\
 & & \\
\mathsf{H} & \equiv & [\mathbf{def}\ \ \mathit{request}\langle \mathit{details}, \kappa\rangle \ \triangleright\ \kappa\langle \mathit{price}, \mathit{accept}\rangle + \mathit{abort} \\
 & & \quad \wedge\ \mathit{accept}\langle \mathit{cc}\rangle \ \triangleright\ 0 \\
 & & \quad \mathbf{in}\ \ \mathit{HotelSrv}\langle \mathit{request}\rangle : \mathit{alt}\langle \mathit{hotel}\rangle] \\
 & & \\
\mathsf{C} & \equiv & [\mathbf{def}\ \ \mathit{offer}\langle \mathit{rate}, \kappa\rangle \ \triangleright\ \kappa\langle \mathit{card}\rangle + \mathit{abort} \\
 & & \quad \mathbf{in}\ \ \mathit{HotelReq}\langle \mathit{data}, \mathit{offer}\rangle : Q]
\end{array}
$$

Figure 6. Hotel Booking

Example 4. Process HB in Figure 6 shows a possible modelling for the hotel booking application, where H describes the behaviour of the hotel service while C models the protocol followed by the client. There are two main differences with the description given in Section 1. First, we adhere to the continuation-passing style for enabling communication among different processes. Note that channels *request* and *offer* carry on one extra parameter (the continuation) that identifies the channel κ where to communicate next. Second, the system is not just the parallel composition of the two parties, but it also contains a merge definition that allows the communication among the two transactions. In fact, the two parties do not start by communicating directly and the first interaction takes place indirectly. Note that C starts by sending the message $\mathit{HotelReq}\langle \mathit{data}, \mathit{offer}\rangle$ to a merge channel. Similarly, H sends $\mathit{HotelSrv}\langle \mathit{request}\rangle$. While neither C nor H can complete their transactions in isolation, these two messages together enable the merge definition that forwards the request from the client to the hotel and joins both transactions, producing the following state

$$
\begin{array}{rcll}
\mathsf{HB} & \equiv & \textbf{def} & \mathit{HotelSrv}\langle r\rangle \mid \mathit{HotelReq}\langle d,\kappa\rangle \blacktriangleright r\langle d,\kappa\rangle \\
& & \textbf{in} & [\textbf{def}\ \mathit{request}\langle \mathit{details},\kappa\rangle \triangleright \kappa\langle \mathit{price},\mathit{accept}\rangle & (1)\\
& & & \qquad\qquad + \mathit{abort} & (2)\\
& & & \quad \wedge\ \mathit{accept}\langle \mathit{cc}\rangle \triangleright 0 & (3)\\
& & & \quad \wedge\ \mathit{offer}\langle \mathit{rate},\kappa\rangle \triangleright \kappa\langle \mathit{card}\rangle & (4)\\
& & & \qquad\qquad + \mathit{abort} & (5)\\
& & & \textbf{in}\ \mathit{request}\langle \mathit{data},\mathit{offer}\rangle : \mathit{alt}\langle \mathit{hotel}\rangle \mid Q]
\end{array}
$$

At this moment the hotel receives the request from the client and it can (nondeterministically) decide whether to make an offer (reaction (1)) or to abort (reaction (2)). We remind that the notation $J \triangleright P + Q$ is syntactic sugar for $J \triangleright P \wedge J \triangleright Q$. If the hotel aborts, then both transactions are aborted and the corresponding compensations (i.e., $\mathit{alt}\langle \mathit{hotel}\rangle \mid Q$) are activated. Otherwise, the hotel can produce the message $\mathit{offer}\langle \mathit{price},\mathit{accept}\rangle$ to send an offer to the client who, in turn, may decide whether to accept it (reaction (4)) or to abort the conversation (rule (5)). The abortion from the client is handled analogously to the previous case. If the client accepts the offer, then it generates the message $\mathit{accept}\langle \mathit{card}\rangle$, which enables the reaction rule (3). When reaction (3) is fired all messages inside the transaction are consumed, and hence the transaction commits (i.e., both parties have successfully finished).

Another interesting example is the modelling of a mailing-list manager with all-or-nothing delivery of messages to subscribers.

Example 5 (Mailing list). Consider a data structure that allows to send atomically a message to a list of subscribers (in the sense that the same message is either sent to all or to none). Such structure can be defined as $\mathsf{ML} = \mathit{MailingList}\langle k\rangle \triangleright \mathsf{MLDef}$, where:

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

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

transaction identified by a fresh name z is generated, and the state of the structure is put inside the transaction, therefore all other activities, such as adding or closing are blocked until the transaction ends. Inside the transaction, the message v is sent to the forwarder at the top of the list y with the identifier of the transaction 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 transaction 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. The following process Sys subscribes two users, *Alice* and *Bob* to the mailing list, and sends the message *News*.

$$\mathsf{Emp} \equiv \mathit{employees}\langle a, t, c\rangle \triangleright a\langle \mathit{Alice}\rangle \mid a\langle \mathit{Bob}\rangle \mid t\langle \mathit{News}\rangle$$

$$\mathsf{Sys} \equiv \mathbf{def}\ \mathsf{ML} \wedge \mathsf{Emp}\ \mathbf{in}\ \mathit{MailingList}\langle \mathit{employees}\rangle$$

A possible computation of the process Sys is shown in Figure 7. In this particular computation, both subscriptions take place before the emission of the message, nevertheless the process does not fix this priority and consequently messages could be consumed in a different order. For simplicity, we abbreviate chemical solutions by omitting definitions present in successive solutions, though we usually write only those definitions involved in the reduction step. Inside solutions, we underline the fired reaction rule and the consumed messages that matched the corresponding pattern. The phase List creation instantiates a new mailing list by defining the fresh ports *add*, *tell* and *close*, which are sent to the port *employee*. The second phase (Subscriptions) adds the names *Alice* and *Bob* to the created mailing list. Phase Distribution of *News* generates a new transaction that produces (in a sequential way) the copies of the name *News* to be sent to any subscriber of the list. Nevertheless those messages are not released until the phase Commit takes place. Only when the transaction commits, the generated messages are atomically sent to subscribers.

3.2. *Flat transactions*

Nesting is a useful mechanism for programming transactions, e.g. when a transaction can succeed even when certain sub-activities fail. In the area of databases, nested transactions have been studied since (Moss, 1981). Contrary to other languages proposed in the literature that do not support nesting (e.g., Webπ_∞ (Lucchi and Mazzara, 2004) and $\rho\pi$ (Lanese et al., 2010a)), cJoin syntax allows for proper nested transactions, like in $[[P : P'] : [Q : Q']]$. Nevertheless, many common situations that would involve nested transactions can be modelled in cJoin without nesting by exploiting dynamic merge and message-passing communication, as shown in the multi-way transaction example (see Section 4.2).

This section introduces *flat* cJoin, which is a sub-calculus of cJoin without nested transactions. The following sections will show that flat cJoin is expressive enough for modelling several common programming patterns (Section 4) and, in addition, how it can be implemented (Sections 5 and 6). In fact, as it will be clear later, the syntactic restrictions imposed on flat cJoin help us to encode flat cJoin back to Join and to extend

Initial Soup: $\{\![\ \underline{\mathsf{Sys}}\ ]\!\} \rightleftharpoons \{\![\mathit{MailingList}\langle k\rangle \triangleright \mathsf{MLDef},\ \mathsf{Emp},\ \mathit{MailingList}\langle \mathit{employees}\rangle\ ]\!\}$

List creation:

$\{\![\ \underline{\mathit{MailingList}\langle k\rangle \triangleright \mathsf{MLDef}},\ \mathsf{Emp},\ \underline{\mathit{MailingList}\langle \mathit{employees}\rangle}\ ]\!\} \to$

$\{\![\mathit{MailingList}\langle k\rangle \triangleright \mathsf{MLDef},\ \mathsf{Emp},\ \underline{\mathsf{MLDef}\{^{\mathit{employees}}/_k\}}\ ]\!\} \rightleftharpoons$

$\{\![\mathit{MailingList}\langle k\rangle \triangleright \mathsf{MLDef},\ \underline{\mathsf{Emp}},\ \mathsf{List}\{^{\mathit{employees}}/_k\},\ \underline{\mathit{employees}\langle \mathit{add}, \mathit{tell}, \mathit{close}\rangle},\ l\langle \mathit{nil}\rangle]\!\} \to$

$\{\![l\langle y\rangle \mid \mathit{add}\langle x\rangle \triangleright \ldots,\ \ldots,\ l\langle \mathit{nil}\rangle,\ \mathit{add}\langle \mathit{Alice}\rangle,\ \mathit{add}\langle \mathit{Bob}\rangle,\ \mathit{tell}\langle \mathit{News}\rangle]\!\}$

Subscriptions:

$\{\![\underline{l\langle y\rangle \mid \mathit{add}\langle x\rangle \triangleright \ldots},\ \ldots,\ \underline{l\langle \mathit{nil}\rangle},\ \underline{\mathit{add}\langle \mathit{Alice}\rangle},\ \mathit{add}\langle \mathit{Bob}\rangle,\ \mathit{tell}\langle \mathit{News}\rangle]\!\} \to$

$\{\![\underline{l\langle y\rangle \mid \mathit{add}\langle x\rangle \triangleright \ldots},\ \ldots,\ z_A\langle v, w\rangle \blacktriangleright \mathit{Alice}\langle v\rangle | \mathit{nil}\langle v, w\rangle,\ \underline{l\langle z_A\rangle},\ \underline{\mathit{add}\langle \mathit{Bob}\rangle},\ \mathit{tell}\langle \mathit{News}\rangle]\!\} \to$

$\{\![l\langle y\rangle \mid \mathit{tell}\langle v\rangle \triangleright \ldots,\ \ldots\ , z_B\langle v, w\rangle \blacktriangleright \mathit{Bob}\langle v\rangle | z_A\langle v, w\rangle,\ l\langle z_B\rangle,\ \mathit{tell}\langle \mathit{News}\rangle]\!\}$

Distribution of *News*:

$\{\![\underline{l\langle y\rangle \mid \mathit{tell}\langle v\rangle \triangleright \ldots}, \ldots, z_B\langle v, w\rangle \blacktriangleright \mathit{Bob}\langle v\rangle | z_A\langle v, w\rangle, \underline{l\langle z_B\rangle}, \underline{\mathit{tell}\langle \mathit{News}\rangle}]\!\} \to$

$\{\![\underline{z_B\langle v, w\rangle \blacktriangleright \mathit{Bob}\langle v\rangle | z_A\langle v, w\rangle}, \ldots, \{\![z\langle\rangle \triangleright 0, \underline{z_B\langle \mathit{News}, z\rangle}, l\langle z_B\rangle, \llcorner l\langle z_B\rangle \lrcorner]\!\}]\!\} \to$

$\{\![\underline{z_A\langle v, w\rangle \blacktriangleright \mathit{Alice}\langle v\rangle | \mathit{nil}\langle v, w\rangle}, \ldots, \{\![z\langle\rangle \triangleright 0, \mathit{Bob}\langle \mathit{News}\rangle, \underline{z_A\langle \mathit{News}, z\rangle}, l\langle z_B\rangle, \llcorner l\langle z_B\rangle \lrcorner]\!\}]\!\} \to$

$\{\![\underline{\mathit{nil}\langle v, w\rangle \blacktriangleright w\langle\rangle}, \ldots, \{\![z\langle\rangle \triangleright 0, \mathit{Bob}\langle \mathit{News}\rangle, \mathit{Alice}\langle \mathit{News}\rangle, \underline{\mathit{nil}\langle \mathit{News}, z\rangle}, l\langle z_B\rangle, \llcorner l\langle z_B\rangle \lrcorner]\!\}]\!\} \to$

$\{\![\ldots, \{\![\underline{z\langle\rangle \triangleright 0}, \mathit{Bob}\langle \mathit{News}\rangle, \mathit{Alice}\langle \mathit{News}\rangle, \underline{z\langle\rangle}, l\langle z_B\rangle, \llcorner l\langle z_B\rangle \lrcorner]\!\}]\!\} \to$

$\{\![\ldots, \{\![z\langle\rangle \triangleright 0, \mathit{Bob}\langle \mathit{News}\rangle, \mathit{Alice}\langle \mathit{News}\rangle, l\langle z_B\rangle, \llcorner l\langle z_B\rangle \lrcorner]\!\}]\!\}$

Commit:

$\{\![\ldots,\ \{\![\underline{z\langle\rangle \triangleright 0},\ \mathit{Bob}\langle \mathit{News}\rangle,\ \mathit{Alice}\langle \mathit{News}\rangle,\ l\langle z_B\rangle,\ \llcorner l\langle z_B\rangle \lrcorner]\!\}\ ]\!\} \rightleftharpoons$

$\{\![\ldots,\ \underline{\{\![\mathit{Bob}\langle \mathit{News}\rangle,\ \mathit{Alice}\langle \mathit{News}\rangle,\ l\langle z_B\rangle,\ \mathbf{def}\ z\langle\rangle \triangleright 0\ \mathbf{in}\ 0,\ \llcorner l\langle z_B\rangle \lrcorner]\!\}}\ ]\!\} \to$

$\{\![\ldots,\ \mathit{Bob}\langle \mathit{News}\rangle,\ \mathit{Alice}\langle \mathit{News}\rangle,\ l\langle z_B\rangle\ ]\!\}$

Figure 7. A possible computation of the Mailing list example

existing distributed implementation of Join to implement flat cJoin. Flat cJoin is defined through the following type system involving the set $T = \{\Box_0, \Box_1, \Box_2\}$ of types and the following type judgements:

- $\vdash P : \Box_0$ The transaction primitive $[\_ : \_]$ does not appear in P at all.
- $\vdash P : \Box_1$ Transactions may appear in P, but only inside definitions. P does not have active transactions but it may create them after some reductions.
- $\vdash P : \Box_2$ P has active flat transactions or may create them after some reductions.
- $\vdash D : \Box_0$ D does not contain transactions.
- $\vdash D : \Box_1$ D may contain flat transactions.

Definition 3.2 (Flat (or well-typed) definitions and processes). A definition D is said *flat* or *well-typed* if $\vdash D : \Box_1$ in the type system shown in Figure 8. Similarly, a process P is said *flat* or *well-typed* if $\vdash P : \Box_2$.

$$\text{(Sub-P)}\ \frac{\vdash P : \Box_i}{\vdash P : \Box_j}\ i<j \qquad \text{(Sub-D)}\ \frac{\vdash D : \Box_0}{\vdash D : \Box_1} \qquad \text{(Zero)}\ \vdash 0 : \Box_0 \qquad \text{(Mess)}\ \vdash x\langle y\rangle : \Box_0 \qquad \text{(Abort)}\ \vdash \mathit{abort} : \Box_0$$

$$\text{(Par)}\ \frac{\vdash P : \Box_i \qquad \vdash Q : \Box_i}{\vdash P|Q : \Box_i} \qquad \text{(Trans)}\ \frac{\vdash P : \Box_0 \qquad \vdash Q : \Box_1}{\vdash [P : Q] : \Box_2} \qquad \text{(Def)}\ \frac{\vdash D : \Box_i \qquad \vdash P : \Box_j}{\vdash \mathbf{def}\ D\ \mathbf{in}\ P : \Box_{max(i,j)}}$$

$$\text{(Conj)}\ \frac{\vdash D : \Box_i \qquad \vdash E : \Box_i}{\vdash D \wedge E : \Box_i} \qquad \text{(Ord-0)}\ \frac{\vdash P : \Box_0}{\vdash J \triangleright P : \Box_0} \qquad \text{(Ord)}\ \frac{\vdash P : \Box_i}{\vdash J \triangleright P : \Box_1} \qquad \text{(Merge)}\ \frac{\vdash P : \Box_0}{\vdash J \blacktriangleright P : \Box_0}$$

Figure 8. Flat cJoin Typing.

We comment on the typing rules in Figure 8. Rules (Sub-P) and (Sub-D) stand for the sub-type order $\Box_0 < \Box_1 < \Box_2$. Clearly, the inert process 0, the emission of a message $x\langle\vec{y}\rangle$ and the constant *abort* do not contain transactions and, hence, they have type $\Box_0$ (Rules Zero, Mess, Abort). The parallel composition $P|Q$ has type $\Box_i$ if both P and Q are typed $\Box_i$ (rule (Par)). Rule (Trans) prevents nesting by stating that $[P : Q]$ is typed $\Box_2$ only when P has no transactions (i.e., $\vdash P : \Box_0$). Note that the process $P \equiv [\mathbf{def}\ a\langle\rangle \triangleright [P_1 : Q_1]\ \mathbf{in}\ a\langle\rangle : Q]$ is not typable. Although P is not a nested transaction, P may evolve to a nested transaction as follows $P \to [\mathbf{def}\ a\langle\rangle \triangleright [P_1 : Q_1]\ \mathbf{in}\ [P_1 : Q_1] : Q]$. Contrastingly, a compensation may contain transactions as part of its definition. For instance, $[x\langle y\rangle \mid \mathit{abort} : \mathbf{def}\ a\langle\rangle \triangleright [P : Q]\ \mathbf{in}\ a\langle\rangle]$ has type $\Box_2$ when $\vdash P : \Box_0$ and $\vdash Q : \Box_1$. The fact that the compensation includes a transaction as part of its definition does not compromise flatness because the compensation will run as an ordinary process after the transaction aborts. In fact, $[x\langle y\rangle \mid \mathit{abort} : \mathbf{def}\ a\langle\rangle \triangleright [P : Q]\ \mathbf{in}\ a\langle\rangle] \to \mathbf{def}\ a\langle\rangle \triangleright [P : Q]\ \mathbf{in}\ a\langle\rangle$, which does not introduce nesting. Rule (Def) combines the types of definitions and processes. Note that $\mathbf{def}\ D\ \mathbf{in}\ P$ is typed $\Box_0$ when neither D nor P contain transactions, i.e., if they both have type $\Box_0$. A process $\mathbf{def}\ D\ \mathbf{in}\ P$ has type $\Box_1$ when transactions appear only in definitions (i.e., either in D or in other local definitions occurring in P). Finally, $\mathbf{def}\ D\ \mathbf{in}\ P$ has type $\Box_2$ when P contains an active transaction.

A composed definition (i.e., a conjunction) is typed $\Box_i$ only when both sub-terms have type $\Box_i$ (By rule (Conj)). An ordinary definition $J \triangleright P$ is well-typed when its guarded processes P is well-typed. Moreover, it has type $\Box_0$ when P has no transactions, i.e., $\vdash P : \Box_0$. Differently, a merge rule is well-typed only when P has type $\Box_0$ (rule (Merge)). This is required in order to avoid nesting, because the instances of P will execute inside transactions.

Example 6 (Well-typed terms). Consider the mailing list process introduced in Example 5. Several sub-terms and their types are shown below:

$$P_1 = \mathbf{def}\ z\langle\rangle \triangleright 0\ \mathbf{in}\ y\langle v, z\rangle \mid l\langle y\rangle \qquad P_2 = [P_1 : l\langle y\rangle]$$
$$D_1 = l\langle y\rangle \mid \mathit{tell}\langle v\rangle \triangleright P_2 \qquad D_2 = l\langle y\rangle \mid \mathit{close}\langle\rangle \triangleright 0$$
$$\vdash P_1 : \Box_0 \qquad \vdash P_2 : \Box_2 \qquad \vdash D_1 : \Box_1 \qquad \vdash D_2 : \Box_0 \qquad \vdash D_1 \wedge D_2 : \Box_1$$

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

$$\frac{\vdash P : \tau}{P : \tau}\ (\textsc{Mol-Proc}) \qquad \frac{\vdash D : \tau}{D : \tau}\ (\textsc{Mol-Def}) \qquad \frac{\vdash P : \square_1}{\llcorner P \lrcorner : \square_0}\ (\textsc{Mol-Fzn}) \qquad \frac{S : \square_0}{\{\!|S|\!\} : \square_2}\ (\textsc{Membrane})$$

$$\frac{S_1 : \square_i \quad S_2 : \square_j}{S_1, S_2 : \square_{max(i,j)}}\ (\textsc{Soup}) \qquad \emptyset : \square_0\ (\textsc{Empty-Soup})$$

Figure 9. Flat Solution Typing.

Example 7 (Counterexample). Process **def** $x\langle\rangle \blacktriangleright [P : 0]$ **in** [**def** D **in** $x\langle\rangle : 0$] is not well-typed because it has a merge definition whose guarded process is a transaction (rule (MERGE) cannot be applied because $\not\vdash x\langle\rangle \blacktriangleright [P : 0] : \square_0$). In fact, this process can reduce in one step to **def** $x\langle\rangle \blacktriangleright [P : 0]$ **in** [**def** D **in** $[P : 0] : 0]$ when $x \notin dn(D)$, which has nested transactions.

3.3. *Properties of Flat cJoin typing*

This section summarises the main properties of our type system, namely, Join processes have type $\square_0$ (Proposition 3.3) and subject reduction holds for $\square_0$ (Lemma 3.8) and $\square_2$ (Lemma 3.9).

Proposition 3.3 (Join processes have type $\square_0$). Let P be a Join process, then $\vdash P : \square_0$.

Proof. The proof follows by induction on the structure of P.

— $P \equiv 0$ and $P \equiv x\langle\vec{y}\rangle$: the proof follows by using either rule (ZERO) or (MESS).
— $P \equiv$ **def** D **in** P' with $D = \wedge_i J_i \triangleright P_i$. By inductive hypothesis, $\vdash P' : \square_0$ and $\vdash P_i : \square_0$ for all i. By using rule (ORD-0), we conclude that $\vdash J_i \triangleright P_i : \square_0$ for all i. By repeatedly using rule (Conj) we conclude $\vdash D : \square_0$. Proof is completed by applying rule (DEF).
— $P \equiv P_1 | P_2$: the proof follows by inductive hypothesis and rule (PAR).

□

In order to prove subject reduction we need some technical preliminaries. In particular, we extend the typing from processes to solutions.

Definition 3.4 (Type of a solution). The type τ of a solution S, noted as $S : \tau$, is defined by rules in Figure 9. Moreover, S is flat iff $S : \square_2$.

We start by proving that Definition 3.4 is consistent w.r.t. structural congruence of solutions, i.e. all types are preserved by α-conversion and heating/cooling; and that the type of a solution reflects on the type of its molecules.

Proposition 3.5. Let σ be a renaming substitution. If $\vdash P : \tau$ then $\vdash P\sigma : \tau$.

Proof. Immediate by the fact that typing does not take into account names, but just the structure of terms, which cannot be changed by renaming substitutions. □

Lemma 3.6. Let $S : \square_j$. If $\{\!|S|\!\} \rightleftharpoons \{\!|S'|\!\}$ then $S' : \square_j$.

Proof. By straightforward case analysis on the applied cooling/heating rule. When the applied rule is (STR-DEF), then Proposition 3.5 is used. □

Corollary 3.7. Let $\{\!|S|\!\} \rightleftharpoons^* \{\!|\otimes_i m_i|\!\}$. Then $S : \Box_j$ iff $\forall i. m_i : \Box_i$ and $i \leq j$.

We are now ready to prove subject reduction for $\Box_0$.

Lemma 3.8 (Subject Reduction for $\Box_0$). Let $P : \Box_0$. If $P \rightarrow^* P'$ then $P' : \Box_0$.

Proof. The proof follows by induction on the length of the derivation.

— **Base case**: $P' \equiv P$ (i.e., $\{\!|P|\!\} \leftrightharpoons^* \{\!|P'|\!\}$). The proof follows by Corollary 3.7.
— **Inductive Step**: Suppose $P \rightarrow P'' \rightarrow^n P'$ with $n \geq 0$. The proof follows by case analysis on the first applied rule and inductive hypothesis. Note that $P \rightarrow P''$ implies $\{\!|P|\!\} \rightleftharpoons^* \{\!|S|\!\} \rightarrow \{\!|S'|\!\} \rightleftharpoons^* \{\!|P''|\!\}$. By CHAM semantics we know that $S \equiv \otimes_i m_i$. Since $P : \Box_0$, then $m_i : \Box_0$ for all i by Corollary 3.7. Hence the only possible rule that can be applied is (RED), because any other rule requires at least a molecule composed by a membrane, which cannot be typed $\Box_0$. Consequently, $S \equiv J \triangleright Q, J\sigma, S''$ and $S' \equiv J \triangleright Q, Q\sigma, S''$, where $S'' : \Box_0$. As $Q : \Box_0$, by Proposition 3.5, $Q\sigma : \Box_0$. Hence $S' : \Box_0$ and therefore $P'' : \Box_0$ (by Corollary 3.7). The proof follows by applying inductive hypothesis on $P'' \rightarrow^n P'$.

□

The following result ensures that $\Box_2$ is preserved by reductions.

Theorem 3.9 (Subject Reduction for $\Box_2$). Let $P : \Box_2$. If $P \rightarrow^* P'$ then $P' : \Box_2$.

Proof. The proof follows by induction on the length of the derivation.

— **Base case**: $P' \equiv P$ (i.e., $\{\!|P|\!\} \leftrightharpoons^* \{\!|P'|\!\}$). The proof follows by Corollary 3.7.
— **Inductive Step**: Suppose $P \rightarrow P'' \rightarrow^n P'$ with $n \geq 0$. The proof follows by case analysis on the first applied rule and inductive hypothesis. Note that $P \rightarrow P''$ implies $\{\!|P|\!\} \rightleftharpoons^* \{\!|S|\!\} \rightarrow \{\!|S'|\!\} \rightleftharpoons^* \{\!|P''|\!\}$. By CHAM semantics we know that $S \equiv \otimes_i m_i$. Since $P : \Box_0$, $m_i : \Box_i$ where $i \leq 2$ for all i by Corollary 3.7. Then, there are four cases:

- Rule (RED): When the reduction occurs at top-level, i.e. $S \equiv J \triangleright Q, J\sigma, S''$, $S' \equiv J \triangleright Q, Q\sigma, S''$, and $S'' : \Box_2$, the proof is similar to Lemma 3.8. The other possibility is when the reduction occurs inside a transaction, e.g. $S \equiv \{\!|S_1|\!\}, S''$ and $S' \equiv \{\!|S_1'|\!\}, S''$, where $\{\!|S_1|\!\} \rightarrow \{\!|S_1'|\!\}$ by rule (RED) and $S'' : \Box_2$. Note that $\{\!|S_1|\!\} : \Box_2$, and therefore $S_1 : \Box_0$. By Lemma 3.8, $S_1' : \Box_0$ and hence $S' : \Box_2$.

(The cases below occur at top-level, because negotiations cannot be nested in P.)

- Rule (COMMIT): $S \equiv \{\!|M|\mathbf{def}\ D\ \mathbf{in}\ 0, \llcorner Q \lrcorner|\!\}, S''$, and $S' \equiv M, S''$, with $S'' : \Box_2$ (by Corollary 3.7). As M is the parallel composition of messages, it can be typed $\Box_0$ and therefore $S' : \Box_2$.
- Rule (ABORT): $S \equiv \{\!|abort|P', \llcorner Q \lrcorner|\!\}, S''$ and $S' \equiv Q, S''$, with $S'' : \Box_2$ (by Corollary 3.7). As $\llcorner Q \lrcorner : \Box_0$, it must be $Q : \Box_1$ and therefore $S' : \Box_2$.
- Rule (MERGE): $S \equiv J_1 | \ldots | J_n \blacktriangleright R, \otimes_i \{\!|J_i\sigma, S_i, \llcorner Q_i \lrcorner|\!\}, S''$ and $S' \equiv J_1 | \ldots | J_n \blacktriangleright R, \{\!|\otimes_i S_i, R\sigma, \llcorner Q_1 | \ldots | Q_n \lrcorner|\!\}, S''$, with $S'' : \Box_2$ (by Corollary 3.7). Since $R : \Box_0$ and for all i $S_i : \Box_0$ and $Q_i : \Box_1$, we have $S' : \Box_2$.

□

Remark 3.10. Subject reduction does not hold for $\Box_1$. Consider $P = \mathbf{def}\ x\langle\rangle \triangleright [Q : Q']\ \mathbf{in}\ x\langle\rangle$, where $\vdash Q : \Box_0$ and $\vdash Q' : \Box_1$. Although $\vdash P : \Box_1$, P reduces to $P' = \mathbf{def}\ x\langle\rangle \triangleright [Q : Q']\ \mathbf{in}\ [Q : Q']$, which can be typed $\Box_2$ but not $\Box_1$.

Theorem 3.9 ensures that flat processes form a sub-calculus since reductions do not generate nesting.

Definition 3.11 (Flat cJoin). Flat cJoin is the sub-calculus of all flat processes.

4. Programming common transactional patterns in cJoin

In order to illustrate the transactional aspects of cJoin, we show how to code some common interaction patterns. We keep and enrich the hotel booking scenario as a running example.

4.1. *Multi-step Transactions: Trip Booking*

Let us assume now that the user is making plans for a trip and wants to make reservations for both a flight and hotel accommodation. Such scenarios are usually modelled by splitting the whole activity as a sequence of two independent transactions: the client executes a transaction for booking a flight first, and when it commits, a new transaction for making hotel reservation is started. If the last transaction aborts, then the first one is compensated for by cancelling the already committed flight reservation. This kind of composition is usually referred to as compensable flow composition, and it is tailored to model long running transactions in an orchestration context. Several proposals have appeared in the literature for describing transactional flow compositions (Butler and Ferreira, 2004; Butler et al., 2005b; Bruni et al., 2005). Essentially, they describe the trip booking problem as follows

$$\mathsf{TripBooking} \equiv \{\texttt{FlightBooking} \div \texttt{FlightCancelation};\ \texttt{BookHotel} \div \texttt{0}\}$$

The entire activity TripBooking is delimited by the long running transaction scope $\{\_\}$. A long running transaction is divided by ';' into sequential steps. Process `FlightBooking` in the first step allows the client to book a hotel accommodation. If it commits, then the compensation `FlightCancelation` is installed and the next step `BookHotel` $\div$ `0` is executed. The compensation is installed when a step commits and it is used only when one of the following steps fails. For instance, if `BookHotel` fails during its execution, then the previously installed compensation `FlightCancelation` is executed.

Although cJoin does not offer a built-in mechanism for these kinds of transactions, they can be coded into cJoin. Let us consider a simple language defined as follow

$$\begin{array}{lcl} L & ::= & \{S\} \\ S & ::= & P \mid P \div Q; S \end{array}$$

where P and Q are cJoin processes. A possible encoding $[\![\_]\!]$ is below

$$
\begin{array}{lcll}
[\![\{S\}]\!] & \equiv & [\![S]\!]_0 & \\
[\![P]\!]_c & \equiv & [P : c] & \\
[\![P \div Q; S]\!]_c & \equiv & \textbf{def} & comp\langle\rangle \triangleright [c \mid Q : 0] \\
 & & \wedge & cont\langle\rangle \triangleright [\![S]\!]_{comp\langle\rangle} \\
 & & \textbf{in} & [cont\langle\rangle \mid P : c] \\
 & & & \text{provided } \{cont, comp\} \cap fn(P|Q|S|c) = \emptyset
\end{array}
$$

The most interesting rule is the last one. A sequence $P \div Q; S$ is encoded w.r.t. a context c that indicates the compensation installed by the previous execution. Then, the sequence $P \div Q; S$ corresponds to a process that activates the transaction $[cont\langle\rangle \mid P : c]$, i.e., it attempts to executes P until completion (i.e., consuming all messages to its local ports). If P finishes (i.e., there is no pending message in local ports) and the transaction commits, then the message $cont\langle\rangle$ is released. This message will activate the execution of the remaining part of the sequence. If P aborts, then the previously installed compensation c is activated. The context for encoding the remaining part S of the sequence is the message $comp\langle\rangle$, corresponding to the updated compensation $[c \mid Q : 0]$. Hence, if activated, such compensation will first attempt to complete the execution of Q. If this is the case, then the transaction will commit by releasing c, which will then activate the previously installed compensations.

Then, the cJoin code for planning a trip as a multi-step transaction is in Figure 10. We model the hotel booking service H as in Figure 6, while the airline service A is analogous to H (the only difference is that A starts by publishing on the merge channel *AirlineSrv* instead of *HotelSrv*). It is worth remarking that the homonymous (private) ports *request* and *accept* defined by both A and H are different ports. The renaming mechanism inherited from the Join calculus ensures that such names are fresh, and hence there are no clashes when transactions are merged. In addition, we add one merging rule for allowing the interaction among the client and the airline. For the sake of the simplicity we do not include dynamic creation of sessions, but the presentation can be straightforwardly extended to consider them.

The system starts its execution with the client interacting with the airline component analogously to hotel conversation described in Example 4. If the interaction aborts, then the client finishes its execution (note that transaction compensation is set to 0). If it commits, the message $cont\langle\rangle$ is released, which enables the second reaction rule of C′. Firing this rule will dynamically create a new transaction for interacting with the hotel booking service (the interaction is as in Example 4). If this newly created transaction finishes successfully, then the whole long running transaction ends by committing. If the last transaction aborts, then the compensation $comp\langle\rangle|P_2$ is released. Note that $comp\langle\rangle$ can fire the first reaction of client component, which will execute `FlightCancelation`, i.e., the compensation of the previously committed step.

$$
\begin{array}{rcl}
\mathsf{TB} & \equiv & \mathbf{def}\ \ \mathit{HotelSrv}\langle r\rangle \mid \mathit{HotelReq}\langle d,\kappa\rangle \ \blacktriangleright\ r\langle d,\kappa\rangle \\
& & \quad \wedge\ \mathit{AirlineSrv}\langle r\rangle \mid \mathit{AirlineReq}\langle d,\kappa\rangle \ \blacktriangleright\ r\langle d,\kappa\rangle \\
& & \mathbf{in}\ \ \mathsf{A} \mid \mathsf{H} \mid \mathsf{C}' \\
\\
\mathsf{A} & \equiv & [\mathbf{def}\ \ \mathit{request}\langle \mathit{details},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{priceFlight}, \mathit{accept}\rangle + \mathit{abort} \\
& & \quad \wedge\ \mathit{accept}\langle \mathit{cc}\rangle \ \triangleright\ 0 \\
& & \mathbf{in}\ \ \mathit{AirlineSrv}\langle \mathit{request}\rangle : P_1] \\
\\
\mathsf{H} & \equiv & [\mathbf{def}\ \ \mathit{request}\langle \mathit{details},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{priceRoom}, \mathit{accept}\rangle + \mathit{abort} \\
& & \quad \wedge\ \mathit{accept}\langle \mathit{cc}\rangle \ \triangleright\ 0 \\
& & \mathbf{in}\ \ \mathit{HotelSrv}\langle \mathit{request}\rangle : P_2] \\
\\
\mathsf{C}' & \equiv & \mathbf{def}\ \ \mathit{comp}\langle\rangle \ \triangleright\ [\texttt{FlightCancellation} : 0] \\
& & \quad \wedge\ \mathit{cont}\langle\rangle \ \ \triangleright\ [\mathbf{def}\ \ \mathit{offer}\langle \mathit{rate},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{card}\rangle + \mathit{abort} \\
& & \qquad\qquad\qquad\ \ \mathbf{in}\ \ \mathit{HotelReq}\langle \mathit{data}, \mathit{offer}\rangle : \mathit{comp}\langle\rangle] \\
& & \mathbf{in}\ \ [\mathit{cont}\langle\rangle \\
& & \qquad \mid \mathbf{def}\ \ \mathit{offer}\langle \mathit{rate},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{card}\rangle + \mathit{abort} \\
& & \qquad\ \ \mathbf{in}\ \ \mathit{AirlineReq}\langle \mathit{data}, \mathit{offer}\rangle : 0]
\end{array}
$$

Figure 10. Trip Booking as a multi-step transaction

4.2. *Multi-way transactions: Trip Booking Revisited*

The main drawback of planning a trip as in the previous section is that compensations are usually not for free. Clearly, cancelling a flight reservation usually requires the client to pay cancelation fees. By taking advantage of cJoin transactional primitives, we can model the trip booking example as a multi-way transaction, i.e., a transaction that retains several entry and exit points, in which parties are not necessarily aware of the remaining parties in the transaction. Clearly the hotel and airline booking services should not be necessarily aware of the combined activity to be carried on by the client, and hence they remain unchanged. The cJoin process for the whole system is shown in Figure 11.

The client is modelled by a transaction that initially sends two messages to different merge channels, one allows the merge with the airline service while the other joins it to the hotel service. Once merged, the parties interact by following the conversation pattern described in Example 4. The main difference is that the abortion of one of the parties after both merge actions take place implies the abortion of the interaction, and hence all three parties abort and run their own compensations. Otherwise, the whole interaction commits when all three parties successfully finish their transactional processes.

As far as the definition of the choreography is concerned, we note that (i) booking services are independent of the behaviour of the client, even though this may induce the transactional scope to be extended to third, unknown parties; (ii) the local description of transactional interfaces ensures transactional properties to several different global interactions; (iii) no coordinator is needed for describing transactional, multiparty choreographies.

There is still one main drawback in the code given above: client description mixes two independent flows of interaction corresponding to two different roles played by the

$$
\begin{array}{rcl}
\mathsf{TB} & \equiv & \mathbf{def}\ \ \mathit{HotelSrv}\langle r\rangle \mid \mathit{HotelReq}\langle d,\kappa\rangle \ \blacktriangleright\ r\langle d,\kappa\rangle \\
& & \quad \wedge\ \mathit{AirlineSrv}\langle r\rangle \mid \mathit{AirlineReq}\langle d,\kappa\rangle \ \blacktriangleright\ r\langle d,\kappa\rangle \\
& & \quad \mathbf{in}\ \ \mathsf{A} \mid \mathsf{H} \mid \mathsf{C}' \\
\\
\mathsf{A} & \equiv & [\mathbf{def}\ \ \mathit{request}\langle \mathit{details},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{priceFlight}, \mathit{accept}\rangle + \mathit{abort} \\
& & \quad \wedge\ \mathit{accept}\langle \mathit{cc}\rangle \ \triangleright\ 0 \\
& & \quad \mathbf{in}\ \ \mathit{AirlineSrv}\langle \mathit{request}\rangle : P_1] \\
\\
\mathsf{H} & \equiv & [\mathbf{def}\ \ \mathit{request}\langle \mathit{details},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{priceRoom}, \mathit{accept}\rangle + \mathit{abort} \\
& & \quad \wedge\ \mathit{accept}\langle \mathit{cc}\rangle \ \triangleright\ 0 \\
& & \quad \mathbf{in}\ \ \mathit{HotelSrv}\langle \mathit{request}\rangle : P_2] \\
\\
\mathsf{C}' & \equiv & [\mathbf{def}\ \ \mathit{hotelOffer}\langle \mathit{rate},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{card}\rangle + \mathit{abort} \\
& & \quad \wedge\ \mathit{airOffer}\langle \mathit{rate},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{card}\rangle + \mathit{abort} \\
& & \quad \mathbf{in}\ \ \mathit{HotelReq}\langle \mathit{dataRoom}, \mathit{hotelOffer}\rangle \\
& & \qquad \mid \mathit{AirlineReq}\langle \mathit{dataFlight}, \mathit{airOffer}\rangle : Q]
\end{array}
$$

Figure 11. Trip Booking

component either as a hotel client or as an airline client. The only binding among these two flows of execution is the fact that both interactions should finish either successfully or with abortion. A more appealing modular definition for client behaviour is below.

$$
\begin{array}{rcl}
\mathsf{C_m} & \equiv & \mathbf{def}\ \ \mathit{RoomFound}\langle\omega\rangle \mid \mathit{FlightFound}\langle\omega\rangle \ \blacktriangleright\ 0 \\
& & \mathbf{in}\ \ [\mathbf{def}\ \ \mathit{hotelOffer}\langle \mathit{rate},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{card}\rangle \mid \mathit{RoomFound}\langle \mathit{hotelOffer}\rangle + \mathit{abort} \\
& & \qquad \mathbf{in}\ \ \mathit{HotelReq}\langle \mathit{dataRoom}, \mathit{hotelOffer}\rangle : Q_1] \\
& & \quad \mid [\mathbf{def}\ \ \mathit{airOffer}\langle \mathit{rate},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{card}\rangle \mid \mathit{FlightFound}\langle \mathit{airOffer}\rangle + \mathit{abort} \\
& & \qquad \mathbf{in}\ \ \mathit{AirlineReq}\langle \mathit{dataFlight}, \mathit{airOffer}\rangle : Q_2]
\end{array}
$$

Now, a client initiates two different transactions to interact independently with the hotel and the airline. The new merge rule allows such transactions to be joined when both flows of interaction terminate. In fact, the corresponding messages to the merge names (i.e., $\mathit{RoomFound}\langle \mathit{hotelOffer}\rangle$ and $\mathit{FlightFound}\langle \mathit{airOffer}\rangle$) are generated when the client transactions accept the offers proposed by the corresponding booking services. Let us consider the monolithic code again (Figure 11). For instance, if the hotel offers a convenient rate but the airline company does not, then the whole transaction is aborted. Hence the client should start from the scratch by booking again a room and trying with a different airline company. In the modular version, it would be enough to start a new transaction for finding a flight, while the booked room will be still a valid reservation. This feature may be interesting when client also may choose one of several available booking services (as discussed in the next sections). For example, the client can try a new airline booking (after an abort) while keeping the room booking. Note that this example also shows how a typical nested transaction pattern is smoothly modelled as a flat cJoin process.

A final remark is that the above decomposition of flows works only when both interactions eventually finish by proposing the merge, otherwise one transaction may remain

blocked for ever. When this is not ensured, the interaction that fails to find a suitable reservation should notify the other transaction to abort (this situation can be handled analogously to abortions of parallel activities explained in Section 4.3).

4.3. *Speculative Computation*

This section illustrates how to program a cJoin process that commits at most one of several concurrent transactions. This programming pattern is called speculative execution or a-posteriori choice, and it is studied by several composition languages (Butler et al., 2005b; Bruni et al., 2005; Laneve and Zavattaro, 2005). Speculative computation is very much related to goal-oriented formalisms, like don't know non-determinism in concurrent logic programming, where several alternatives must be explored before one can be selected. For example, when guarded Horn clauses are considered, the selection of the rule to be applied next for reducing a goal is subordinated to successful evaluation of the guard. Once a guarded clause is applied, all the other alternatives are pruned out and that intermediate goal reduction is committed (never to be undone). As guard evaluation can possibly trigger complex computations on its own and require further clause selections giving rise to speculative computations, whose abort corresponds to the non-satisfiability of the clause. One important difference though is that the linguistic abstraction we are after is not a basic search mechanism, deemed to fail in most cases, but rather a strategy language to increase the possibility of success and to handle search failures when they happen.

Although this pattern is not a built-in operator of cJoin, it can be coded by using merge definitions. For simplicity, we will consider just two concurrent transactions, but the mechanism can be easily extended to larger sets.

We assume a client trying to book a room from one of two alternative hotels H_1 and H_2 (in no particular order), but wishing to make a reservation in only one of them. The system is described below.

$$
\begin{array}{rcll}
\mathsf{HB} & \equiv & \textbf{def} & \mathit{HotelSrv}_1\langle r\rangle \mid \mathit{HotelReq}_1\langle d,\kappa\rangle \ \blacktriangleright\ r\langle d,\kappa\rangle & (1)\\
& & & \mathit{HotelSrv}_2\langle r\rangle \mid \mathit{HotelReq}_2\langle d,\kappa\rangle \ \blacktriangleright\ r\langle d,\kappa\rangle & (2)\\
& & \textbf{in} & \mathsf{H}_1 \mid \mathsf{H}_2 \mid \mathsf{C} & (3)
\end{array}
$$

$$
\begin{array}{rcll}
\mathsf{C} & \equiv & \textbf{def} & \mathit{alt}\langle \mathit{id}\rangle \mid \mathit{booked}_1\langle w\rangle \ \blacktriangleright\ \mathit{cancelling}_2\langle\rangle & (4)\\
& & \wedge & \mathit{alt}\langle \mathit{id}\rangle \mid \mathit{booked}_2\langle w\rangle \ \blacktriangleright\ \mathit{cancelling}_1\langle\rangle & (5)\\
& & \wedge & \mathit{cancelling}_1\langle\rangle \triangleright [\textbf{def}\ w\langle\rangle \triangleright 0\ \textbf{in}\ \mathit{aborting}_1\langle w\rangle : 0] & (6)\\
& & \wedge & \mathit{cancelling}_2\langle\rangle \triangleright [\textbf{def}\ w\langle\rangle \triangleright 0\ \textbf{in}\ \mathit{aborting}_2\langle w\rangle : 0] & (7)\\
& & \wedge & \mathit{booked}_1\langle i\rangle \mid \mathit{aborting}_1\langle j\rangle \blacktriangleright \mathit{abort} & (8)\\
& & \wedge & \mathit{booked}_2\langle i\rangle \mid \mathit{aborting}_2\langle j\rangle \blacktriangleright \mathit{abort} & (9)\\
& & \wedge & a\langle\rangle \triangleright [\textbf{def}\ w\langle\rangle \triangleright 0\ \textbf{in}\ \mathit{alt}\langle w\rangle : a\langle\rangle] & (10)\\
& & \textbf{in} & [\textbf{def}\ \mathit{offer}\langle \mathit{rate},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{card}\rangle \mid \mathit{booked}_1\langle \mathit{offer}\rangle & (11)\\
& & & \qquad\qquad + \mathit{abort} & (12)\\
& & & \quad \textbf{in}\ \mathit{HotelReq}_1\langle \mathit{data}, \mathit{offer}\rangle \quad : Q_1] & (13)\\
& & & \mid [\textbf{def}\ \mathit{offer}\langle \mathit{rate},\kappa\rangle \ \triangleright\ \kappa\langle \mathit{card}\rangle \mid \mathit{booked}_2\langle \mathit{offer}\rangle & (14)\\
& & & \qquad\qquad + \mathit{abort} & (15)\\
& & & \quad \textbf{in}\ \mathit{HotelReq}_2\langle \mathit{data}, \mathit{offer}\rangle \quad : Q_2] & (16)\\
& & & \mid a\langle\rangle & (17)
\end{array}
$$

Lines (1)−(3) define the system, which is composed by hotel services H_1 and H_2 defined as in previous examples, and a client component C. The client C consists in the concurrent execution of two transactions (lines (11) − (13) and (14) − (16)) for dealing with each booking service. They behave as in previous cases, but they finish the conversation by generating the merge messages $\mathit{booked}_i\langle \mathit{offer}\rangle$ (lines (11) and (12)). Differently from multi-way transactions (Section 11), those messages are used not for joining both transactions but for encoding a kind of transactional internal choice. Let us suppose that both client transactions complete successfully; then the state of the client can be seen as follow

$$
\begin{array}{lr}
\textbf{def}\ \ldots & (4-10)\\
\textbf{in}\quad [\textbf{def}\ \ldots & (11-12)\\
\qquad \textbf{in}\ \mathit{booked}_1\langle \mathit{offer}\rangle \quad : Q_1] & (13')\\
\quad \mid [\textbf{def}\ \ldots & (14-15)\\
\qquad \textbf{in}\ \mathit{booked}_2\langle \mathit{offer}\rangle \quad : Q_2] & (16')\\
\quad \mid [\textbf{def}\ w\langle\rangle \triangleright 0\ \textbf{in}\ \mathit{alt}\langle w\rangle : a\langle\rangle] & (17')
\end{array}
$$

where definitions rules have been omitted since they remain unchanged. Line (17′) has been obtained by firing the reaction rule in line (10) with message $a\langle\rangle$ (line (17)). No transaction can commit in this state because messages sent to merge ports carry on local names (*offer* and w) – this is the standard way to force a transaction to join before committing. At this point the merge rules in lines (1) and (2) are enabled. Assuming the second one fires, the client state reduces to

$$
\begin{array}{lll}
\textbf{def}\ \ldots & & (4-10)\\
\textbf{in}\quad [\textbf{def}\ \ldots & & (11-12)\\
\quad\ \textbf{in}\ \mathit{booked}_1\langle \mathit{offer}\rangle & : Q_1] & (13')\\
\quad |\ [\textbf{def}\ \ldots & & (14-15)\\
\quad\ \wedge\, w\langle\rangle \triangleright 0 & & \\
\quad\ \textbf{in}\ \mathit{cancelling}_1\langle\rangle : a\langle\rangle | Q_2] & & (16')
\end{array}
$$

Assuming the hotel component also commits, then the second transaction commits by releasing the message $\mathit{cancelling}_1\langle\rangle$, which enables the reaction rule in line (6). After firing this rule, client state can be described as follow.

$$
\begin{array}{lll}
\textbf{def}\ \ldots & & (4-10)\\
\textbf{in}\quad [\textbf{def}\ \ldots & & (11-12)\\
\quad\ \textbf{in}\ \mathit{booked}_1\langle \mathit{offer}\rangle & : Q_1] & (13')\\
\quad |\ [\textbf{def}\ w\langle\rangle \triangleright 0\ \textbf{in}\ \mathit{aborting}_1\langle w\rangle : 0] & &
\end{array}
$$

Above state enables the merge rule in line (8). Client component moves to the following state when such rule is fired.

$$
\begin{array}{lll}
\textbf{def}\ \ldots & & (4-10)\\
\textbf{in}\quad [\textbf{def}\ \ldots & & (11-12)\\
\quad\ \wedge\, w\langle\rangle \triangleright 0 & & \\
\quad\ \textbf{in}\ \mathit{abort} & : Q_1] &
\end{array}
$$

Finally, the remaining transaction aborts and the compensation Q_1 is released. Hence, the transaction with H_2 has been committed while the conversation with H_1 has been aborted. The cases in which one transaction commits and the other aborts follow immediately since rules defined in lines (8) and (9) are never used, and therefore the auxiliary transaction generated by message $\mathit{cancelling}_{\mathsf{i}}$ remains blocked.

We comment on reaction rule in line (10), which generates an auxiliary transaction for selecting the committing interaction. Note that its compensation is set to $a\langle\rangle$. Although the scenario described so far assumes that the hotel component does not generate *abort* after receiving client confirmation, this may not be the case in the most general setting. Hence, if the selected transaction aborts after being chosen for committing, then the compensation $a\langle\rangle$ is released in order to allow the remaining alternative to be eligible again.

5. Language implementation

This section addresses the problem of implementing the transactional primitives provided by cJoin. For simplicity, we only consider flat cJoin, i.e., the sub-calculus of transactions without nesting introduced in Definition 3.11. In particular, we show how flat cJoin can be encoded into Join. For the sake of the simplicity we omit many technical details in this presentation and provide just a sketch of the translation and the main results. The

formal definition of the encoding and its completeness and correctness results can be found in (Melgratti, 2005).

Intuitively, transactional processes are implemented in Join by making explicit a commit protocol used by several parties to reach an agreement. In particular, we rely on the *Distributed Two Phase Commit* (D2PC) of (Bruni et al., 2002), because it is appropriate for handling situations in which parties are not necessary aware of the whole set of participants involved in the transaction. Nevertheless, the encoding is parametric w.r.t. to the selected commit protocol, hence, the translation can be adapted to make participants conform to other proposals such as the standards WS-ATOMIC TRANSACTION or WS-BUSINESS ACTIVITY. In what follows we call a *coordinator* any party performing the selected commit protocol. We will denote coordinators by Coor. The formal definition for the Join processes Coor used in our translation can be found in (Melgratti, 2005). The code is also reported in the appendix for the interested reader, but we find unnecessary here to illustrate its code in detail, because the D2PC is not the focus of this paper. Here, let us just assume that a coordinator offers (a fresh instance of) the following ports to communicate with:

Name	Parameters	Stands for
cmp	m	to set the compensation m to be delivered on abort
cmt	κ	to start the protocol by voting commit. κ is the continuation to be released on commit
abt		to start the protocol by voting abort
join	*coord*	to join the coordinator *coord* to the same transaction

The key strategy for encoding a transaction into a Join process is to assign a coordinator to any execution thread of the transaction, i.e., any message sent to a transactional or merge port is monitored by one coordinator. This is achieved by making any such message carry on the ports that allows ones to interact with its coordinator. For instance, consider the following cJoin transaction

$$T \;\equiv\; [\,\mathbf{def}\; x\langle\rangle \triangleright P\;\mathbf{in}\; x\langle\rangle \;\; : Q]$$

The corresponding Join term, denoted $[\![T]\!]$, is defined as follows:

$$\begin{array}{rcll} [\![T]\!] & \equiv & \mathbf{def} & \mathsf{Coor} \\ & & \wedge & x\langle c, a, j\rangle \triangleright [\![P]\!]_{c,a,j} \\ & & \wedge & \mathit{undo}\langle\rangle \triangleright [\![Q]\!] \\ & & \mathbf{in} & \quad x\langle \mathit{cmt}, \mathit{abt}, \mathit{join}\rangle \\ & & & \mid \mathit{cmp}\langle \mathit{undo}\rangle \end{array}$$

The transaction T is encoded as the process $[\![T]\!]$, which introduces one fresh coordinator Coor to monitor the unique execution thread of T (i.e., message $x\langle\rangle$ in this particular case). As mentioned before, Coor defines the fresh ports *cmt*, *abt*, *join* and *cmp*. The message $x\langle\rangle$ in T, which is monitored by Coor in the translation, is encoded as the mes-

sage $x\langle cmt, abt, join\rangle$, which carries on the names needed for interacting with Coor (the usage of these names will be illustrated below). Note that the original definition of x (rule $x\langle\rangle \triangleright P$) needs to be amended to take into account the three new parameters, i.e., the original definition is mapped to $x\langle c, a, j\rangle \triangleright [\![P]\!]_{c,a,j}$. We highlight that the original guarded process P is translated as $[\![P]\!]_{c,a,j}$, where $[\![P]\!]_{c,a,j}$ denotes the encoding of the process P that is being monitored by a coordinator identified by the ports c, a, j. Finally, we introduce the new local port *undo*, which is used as the guard of the (encoded form of the) compensation Q. Note that this name is used for setting up the compensation of Coor (message $cmp\langle undo\rangle$). We assume that Coor will deliver a message to its settled compensation when the execution of the commit protocol finishes with abort. Consequently, rule $undo\langle\rangle \triangleright [\![Q]\!]$ will be enabled only when the commit protocol aborts. When fired, the encoded form of the original compensation Q is activated.

We now analyse how a monitored processes P is translated into $[\![P]\!]_{c,a,j}$. We start by considering the encoding of the following five different forms of monitored processes.

— $\mathbf{P \equiv 0}$. In this case the monitored process P has finished its execution successfully. Consequently, this thread can request its coordinator to initiate the commit protocol by voting commit. Hence, the encoding of 0 is defined as follows

$$[\![0]\!]_{c,a,j} = c\langle\rangle$$

The encoded form of 0 is a message sent to the commit port of the monitor of the thread. This message causes the coordinator to start the commit protocol by proposing commit. If all involved parties in the transaction commit, then the transaction finishes successfully. The coordinator monitoring P will finish silently, since no continuation is being set when message $c\langle\rangle$ is sent (i.e., $c\langle\rangle$ does not carry on any value).

— $\mathbf{P} = abort$. In this case the thread being monitored reaches the abort condition. Consequently, it informs its coordinator that the whole transaction must abort by sending a message to the corresponding abort port. Hence, *abort* is encoded as follows

$$[\![abort]\!]_{c,a,j} = a\langle\rangle$$

In this case the commit protocol will finish with abort because there is at least one coordinator voting for abort. As a consequence, the coordinators in the transaction will release their settled compensations. In fact the message $cmp\langle undo\rangle$ will be consumed to issue $undo\langle\rangle$ and therefore trigger $[\![Q]\!]$.

— **P is a message sent to a transactional or merge port**. The encoding of a monitored message sent to a transactional or a merge port is just obtained by extending the parameters of the original message with the ports corresponding to the monitor of the thread (as done for the initial thread of a transaction). Therefore, the encoding of a transactional message is defined below

$$[\![x\langle\vec{v}\rangle]\!]_{c,a,j} = x\langle c, a, j, \vec{v}\rangle$$

— **P is a message sent to a non transactional port**. As an example, consider the transaction $T \equiv [\,\mathbf{def}\ x\langle\rangle \triangleright P\ \mathbf{in}\ x\langle\rangle \mid R : Q]$ with $P = z\langle\rangle$. A possible reduction for T is $T \rightarrow^* [\,\mathbf{def}\ x\langle\rangle \triangleright P\ \mathbf{in}\ z\langle\rangle \mid R : Q]$. Note that the message $z\langle\rangle$ produced inside of the transaction will be delivered to its recipient only when the transaction commits.

Consequently, a rule like $x\langle\rangle \triangleright z\langle\rangle$ above should be interpreted as a transactional thread that is finishing its execution and expects to deliver $z\langle\rangle$ if the transaction finally commits. Hence, its encoding is defined as follows

$$[\![z\langle\rangle]\!]_{c,a,j} = c\langle z\rangle$$

The main difference with the encoding of 0 is that the commit message sent to the coordinator sets z as the continuation. The coordinator will start the commit protocol by voting commit after receiving this message. If the commit protocol finally terminates with commit, then this coordinator will release the message $z\langle\rangle$.

— **P has two execution threads**. For instance, $P = P_1|P_2$. In this case, the translation needs to dynamically generate a new coordinator because any thread needs to be monitored by one coordinator. The encoding is defined as follows

$$[\![P_1|P_2]\!]_{c,a,j} = \textbf{def } \mathsf{Coor} \textbf{ in } [\![P_1]\!]_{cmt,abt,join} \mid [\![P_2]\!]_{c,a,j} \mid j\langle cmt, abt\rangle \mid join\langle c, a\rangle$$

Note that we generate a new coordinator Coor that provides the definition for the fresh ports cmt, abt and $join$. Then, P_1 will be monitored by the newly defined coordinator (i.e., $[\![P_1]\!]_{cmt,abt,join}$) while P_2 will be monitored by the coordinator already assigned to the whole process $P_1|P_2$ (i.e., $[\![P_2]\!]_{c,a,j}$). Messages $j\langle cmt, abt\rangle$ and $join\langle c, a\rangle$ make coordinators aware of each other: $j\langle cmt, abt\rangle$ joins the new coordinator as a participant of the transaction being monitored by the coordinator identified by c, a, j, while $join\langle c, a\rangle$ works in the other way round.

We now focus on the encoding of firing rules. The encoding of reduction rules outside of a transaction is simply the application of the encoding to the guarded process, i.e.,

$$[\![J \triangleright P]\!] = J \triangleright [\![P]\!]$$

The translation treats similarly transactional and merge rules and we just consider two different shapes for such kind of rules:

— **Single-message join pattern.** Such rules are either like $x\langle\vec{v}\rangle \triangleright P$ or $x\langle\vec{v}\rangle \blacktriangleright P$. As mentioned before, rule $x\langle\vec{v}\rangle \triangleright P$ is encoded by adding to the port x three new parameters for identifying the coordinator associated with the thread. Moreover, that coordinator will monitor also the execution of the guarded process P. The translation is defined as follows

$$[\![x\langle\vec{v}\rangle \triangleright P]\!] = x\langle c, a, j, \vec{v}\rangle \triangleright [\![P]\!]_{c,a,j}$$

Analogously, the encoding of a merge rule is as follows

$$[\![x\langle\vec{v}\rangle \blacktriangleright P]\!] = x\langle c, a, j, \vec{v}\rangle \triangleright [\![P]\!]_{c,a,j}$$

— **Two-message join pattern.** Reactions contain synchronisations, e.g., $x\langle\rangle|y\langle\rangle \blacktriangleright P$. This rule is encoded as follows

$$x\langle c_1, a_1, j_1\rangle \mid y\langle c_2, a_2, j_2\rangle \triangleright [\![P]\!]_{c_1,a_1,j_1} \mid j_1\langle c_2, a_2\rangle \mid j_2\langle c_1, a_1\rangle \mid c_2\langle\rangle$$

The translation selects one thread (in this case the first one) to continue. In fact, the original guarded process P is assigned to the first coordinator because it is encoded as $[\![P]\!]_{c_1,a_1,j_1}$. The messages j_1 and j_2 makes coordinators part of the same transaction,

as described previously. The last message $c_2\langle\rangle$ indicates that the second thread is finished by notifying its coordinator to start the commit protocol with vote commit.

In the previous description of the encoding we just focused on a few forms of processes (for instance, we have discarded all processes containing join patterns with more than two messages). Nevertheless, it can be shown that this syntactical restriction, called the class of *canonical processes*, does not change the expressiveness of cJoin (formal details can be found in (Melgratti, 2005)). For the sake of clarity, we report below the formal statement of the results ensuring the correctness and completeness of our encoding. We will use $\rightarrow_J$ and $\rightarrow_{cJ}$ to distinguish reductions in Join from those in cJoin. Moreover, the notion of process equivalence we use relies on barbs defined as follows.

Definition 5.1 (Barb). The observation predicate $\downarrow_x$, also known as the *strong barb*, detects whether a process emits on some free name x:

$$P \downarrow_x \text{ iff } \exists P', \vec{u} : P \equiv \textbf{def } D \textbf{ in } P' | x\langle\vec{u}\rangle \text{ and } x \notin dn(D)$$

Note the processes *abort* and $[\_ : \_]$ have no strong barbs. Moreover, merge names are part of the defined names of a process, and hence not observable.

Lemma 5.2. For any canonical flat process P s.t. $\vdash P : \Box_1$ we have $\forall x. P \downarrow_x \Leftrightarrow [\![P]\!] \downarrow_x$.

Correspondence results assumes the following property about the commit protocol: compensations corresponding to every coordinator in a transaction are released when at least one coordinator is required to abort while all continuations are released when every coordinator in the transaction is required to commit. Otherwise, none coordinator finishes, and none compensations nor continuations are released. It has been shown in (Melgratti, 2005) that the D2PC used in our encoding satisfies above condition.

Theorem 5.3 (Correctness, part 1). Let P be a canonical flat process and $\vdash P : \Box_1$. If $P \rightarrow_{cJ}^* P'$ either $P' \equiv P$ or the following two conditions hold:

1 $P' \equiv \textbf{def } D' \textbf{ in } M' \mid \Pi_{i \in 1..n'} \mathcal{N}_i$, where $\mathcal{N}_i$ are cJoin transactions,
2 $\exists Q$ s.t. $[\![P]\!] \rightarrow_J^* Q$ and $Q \equiv (\textbf{def } [\![D']\!] \textbf{ in } [\![M']\!] | \Pi_{i \in 1..n'} R_i) | \textbf{def } D_g \textbf{ in } 0$, where each R_i is the *standard Join negotiation*† associated to $\mathcal{N}_i$ and D_g collects garbage definitions corresponding to instances of the commit protocol that have terminated.

Proof Sketch. The proof follows by case analysis on P. Note that P cannot be of the form $[P' : Q']$ because $\not\vdash [P' : Q'] : \Box_1$. If P has no local definitions (i.e., $P \not\equiv \textbf{def } D \textbf{ in } M$), then P is either *abort*, the inert process 0, or the parallel composition of messages (containing only free names because there are no local definitions). In all three cases, $P' = P$. Last case is when P contains local definitions, i.e., $P \equiv \textbf{def } D \textbf{ in } M$. For this case we show by induction on the length of the derivation of $P \rightarrow_{cJ}^* P'$ that the conditions 1 and 2 of the thesis hold (see Appendix). □

† A standard Join negotiation is the Join counterpart of a cJoin transaction. It basically consists of the set of transaction coordinators belonging to the same transaction and of all the execution threads associated to those coordinators. Its formal definition can be found in (Melgratti, 2005).

Theorem 5.4 (Correctness, part 2). Let P be a canonical flat process and $\vdash P : \Box_1$. If $P \rightarrow_{cJ}^* P'$ and $\vdash P' : \Box_1$, then $\exists Q$ s.t. $[\![P]\!] \rightarrow_J^* Q$ and $\forall x. P' \downarrow_x \Rightarrow Q \downarrow_x$.

Proof Sketch. By Theorem 5.3, we know that either $P' \equiv P$ or the following two conditions hold:

1 $P' \equiv \mathbf{def}\ D'\ \mathbf{in}\ M' \mid \Pi_{i\in 1..h'}\mathcal{N}_i$, where $\mathcal{N}_i$ are cJoin transactions,
2 $\exists Q$ s.t. $[\![P]\!] \rightarrow_J^* Q$ and $Q \equiv (\mathbf{def}\ [\![D']\!]\ \mathbf{in}\ [\![M']\!] | \Pi_{i\in 1..h'} R_i) | \mathbf{def}\ D_g\ \mathbf{in}\ 0$,

If $P' \equiv P$ then the thesis trivially follows by taking $Q = [\![P]\!]$ and applying Lemma 5.2.

Otherwise, $P \equiv \mathbf{def}\ D\ \mathbf{in}\ M$ for some D and M, and by $\vdash P' : \Box_1$, it must be the case that $P' \equiv \mathbf{def}\ D'\ \mathbf{in}\ M'$ for some D' and M'.

By property above, $\exists Q$ s.t. $[\![P]\!] \rightarrow_J^* Q = \mathbf{def}\ [\![D']\!]\ \mathbf{in}\ [\![M']\!] \mid \mathbf{def}\ D_g\ \mathbf{in}\ 0$. It is easy to notice that $\forall x : P' \downarrow_x \Rightarrow Q \downarrow_x$ because the encoding ensures that $fn(M') = fn([\![M']\!])$ and $dn([\![D']\!]) \cap fn(M) = \emptyset$. □

Theorem 5.5 (Completeness). Let P be a canonical flat process and $\vdash P : \Box_1$. If $[\![P]\!] \rightarrow_J^* Q$, then $P \rightarrow_{cJ}^* P'$ and $\forall x : norm(Q) \downarrow_x \Rightarrow P' \downarrow_x$, where $norm(Q)$ denotes the process obtained from Q by finishing the execution of all instances of the commit protocol that can reach an agreement.

Proof. We proceed by case analysis on the structure of P. Since $\vdash P : \Box_1$, then $P \not\equiv [P' : Q']$. When P has no local definitions, then it is the parallel composition of messages on free ports, the inert process 0 and *abort*. For any of these cases it holds that $[\![P]\!]$ does not have any definition, and therefore $[\![P]\!]$ cannot reduce. The only possibility is $Q = norm(Q) = [\![P]\!]$, which trivially satisfies $\forall x : Q \downarrow_x \Rightarrow [\![P]\!] \downarrow_x$. If $P \equiv \mathbf{def}\ D\ \mathbf{in}\ M$, then we show that the following three conditions hold:

1 $Q \equiv \mathbf{def}\ [\![D']\!]\ \mathbf{in}\ [\![M'_1]\!] \mid \Pi_{i\in 1..u} R'_i \mid \Pi_{k\in 1..f} T'_k \mid \mathbf{def}\ D_g\ \mathbf{in}\ 0$, where R'_i are unfinished Join negotiations (i.e., some transactional thread has not finished), while T'_k are finished negotiations, with $norm(\Pi_{k\in 1..f} T'_k) \equiv [\![M_2]\!] | \mathbf{def}\ D_c\ \mathbf{in}\ 0$.
2 $P \rightarrow_{cJ}^* P' \equiv \mathbf{def}\ D'\ \mathbf{in}\ M'_1 \mid M_2 \mid \Pi_{i\in 1..u}\mathcal{N}_i$ where $\mathcal{N}_i$ is a standard cJoin transaction corresponding to R'_i.
3 $norm(Q) \equiv \mathbf{def}\ [\![D']\!]\ \mathbf{in}\ [\![M'_1 | M_2]\!] \mid norm(\Pi_{i\in 1..u} R'_i) \mid \mathbf{def}\ D'_g\ \mathbf{in}\ 0$

Above conditions are proved by induction on the length of the derivation $[\![P]\!] \rightarrow_J^* Q$ (see Appendix).

Finally, condition $\forall x : norm(Q) \downarrow_x \Rightarrow P' \downarrow_x$ immediately follows from conditions (2) and (3).

□

We chose not to report here the full details about the encoding since the formal definition gets quite complex because of the several alternatives to be considered in the most general case. These alternatives came from the fact that the encoding of a guarded process (in addition to what was explained before) may depend on the type of the received names, in particular whether they are transactional or not. However, the increased complexity of the notation makes it heavy with no evident conceptual benefit. The interested reader can found the full details spelled out in (Bruni et al., 2003; Melgratti, 2005).

6. t-JoCaml

We take advantage of the transactional mechanism of cJoin to extend a programming language with transaction primitives. We have chosen JoCaml (Conchon and Le Fessant, 1999), one of the available implementations of Join. We start this section by giving an overview of JoCaml. Then, we describe the transactional extension we propose, called *transactional* JoCaml (t-JoCaml), and finally, we sketch the main aspects of t-JoCaml implementation.

6.1. *JoCaml*

JoCaml adds Join primitives to *Objective Caml* (Ocaml), which is a functional language with support for object oriented and imperative paradigms. JoCaml provides three main abstractions: *process*, *channels* and *join-patterns*. Processes represent communication and synchronisation tasks. Basic processes are asynchronous messages, while complex processes are obtained by composing expressions and concurrent processes. Channels are JoCaml abstractions corresponding to Join names. There are two different kind of channels: *synchronous* and *asynchronous*. Channels are defined as follows.

$$\textbf{let def}\ \ name[!](args) = P(args);;$$

The above definition creates a channel (named *name*) and a receiver for it, which will execute the *guarded process* P every time it receives a message. Any channel may be defined either as asynchronous, when its name is suffixed with the symbol !, or as synchronous, otherwise. Synchronous names must return a value, i.e., P must explicitly define the return value v by using the sentence **reply** v. Finally, join-patterns define several channels at the same time and state a synchronisation among them: the guarded process may be activated only when all channels have pending messages. The following is a possible join pattern definition

$$\begin{array}{rl}\textbf{let def} & a!(x) \mid b!(y) = P(x,y)\\ \textbf{or} & a!(x) \mid c!(z) = Q(x,z)\\ ;;\end{array}$$

The process above introduces three new asynchronous ports, namely a, b and c. Like join processes, the guarded process P (depending on variables x and y) can be activated only when both a and b have pending messages. Similarly, $Q(x,z)$ can be activated when both a and c have pending messages. Moreover, when both rules are enabled, the selection is unspecified.

Processes can also create fresh ports dynamically. Consider the following program

$$\begin{array}{l}\textbf{let def}\ new\_process() =\\ \qquad\qquad \begin{array}{rl}\textbf{let def} & a!() \mid b!() = P\\ \textbf{or} & a!() \mid c!() = Q\\ \textbf{in reply} & a,b,c\end{array}\\ ;;\end{array}$$

It declares a synchronous port *new_process* (i.e., an ordinary function) that, when called, creates a new process defining three fresh ports a, b and c. The caller is given back the names of the created ports (by clause **in reply** a, b, c).

6.2. *Transactions for JoCaml*

In order to add transactions to JoCaml, we extend its syntax by allowing the definition of a compensable transaction, abort decision, and merge definitions.

6.2.1. *t-JoCaml syntax* As already mentioned, we added two new forms of processes to JoCaml: transactional processes and abort. A transactional process is written as **let trans** P **cmp** Q, where P is an ordinary JoCaml program and Q may be either an ordinary JoCaml process or a transactional one. For instance, the client component in Figure 6 can be written

$$\begin{array}{l}\textbf{let trans def}\ \ \mathit{offer}!(\mathit{rate}, k) = k(\mathit{card})\\ \qquad\qquad\textbf{or}\ \ \mathit{offer}!(\mathit{rate}, k) = \textbf{abort}\\ \qquad\textbf{in}\ \mathit{HotelReq}(\mathit{data}, \mathit{offer})\\ \textbf{cmp}\ Q\\ ;;\end{array}$$

Note the straightforward correspondence with the cJoin definition in Figure 6. Similarly to cJoin, transactional processes in t-JoCaml may decide to abort the execution of a transaction by using the new primitive **abort**.

Merge patterns are defined by writing the keyword **board** in front of the corresponding join patterns. Then, merge definition for the booking trip system in Figure 6 can be written as below

$$\textbf{let board}\ \mathit{HotelSrv}!(r) \mid \mathit{HotelReq}!(d, k) = r(d, k)\ \textbf{in}\ldots$$

As the corresponding Join definition, the above sentence introduces two merge ports *HotelSrv* and *HotelReq*, with the guarded process $r(d, k)$, which is required to be an ordinary JoCaml process (i.e., without transaction primitives).

6.2.2. *Extending JoCaml compiler* As far as the compiler implementation is concerned, we translate syntactically JoCaml programs with transactions into ordinary JoCaml code. We do this by reusing the parsing phase of the JoCaml distribution by slightly modifying the lexer and the parser in order to recognise processes built with the primitives **trans**, **comp**, **abort** and **board**. After the construction of the parse tree, we generate a new file containing the corresponding JoCaml source code that uses the encoding presented in Section 5. The implementation is available at `http://www.di.unipi.it/~melgratt/cjoin`.

The main limitation of the current prototype version is that it cannot handle the compilation of separate units. This restriction relies on the fact that the translation is parametric on the types of free names (i.e., whether they are ordinary or merge channels). Current translation of a program assumes that the typing environment is initially empty

and it is updated when new port definitions are introduced by the program. Consequently, our prototype is unable to handle merge definitions introduced by different files. This constraint could be overcame by adding a new primitive for importing declarations of merge ports explicitly.

7. Big-Step Semantics and Serializability

In this section we introduce an alternative definition for the semantics of cJoin that allows us to reason about transactional computations at different levels of abstraction. The big-step semantics is intended to single out those computations of a system that are not transient or, in other words, to describe the evolution of a system through states that do not contain active transactions.

Example 8. Consider the cJoin process $P \equiv \mathbf{def}\ \mathsf{D}\ \mathbf{in}\ c\langle a\rangle | c\langle b\rangle$ with D defined as follows

$$\begin{aligned}\mathsf{D} \equiv\ & a\langle x\rangle | b\langle y\rangle \blacktriangleright ok\langle\rangle \\ \wedge\ & a\langle x\rangle | b\langle y\rangle \blacktriangleright abort \\ \wedge\ & c\langle x\rangle \triangleright [\mathbf{def}\ z \triangleright 0\ \mathbf{in}\ x\langle z\rangle : Q]\end{aligned}$$

The process P may evolve as follows:

$$\begin{aligned} P \ &\to\ P_1 \equiv \mathbf{def}\ \mathsf{D}\ \mathbf{in}\ [\mathbf{def}\ z \triangleright 0\ \mathbf{in}\ a\langle z\rangle : Q\{^a/_x\}] \mid c\langle b\rangle \\ &\to\ P_2 \equiv \mathbf{def}\ \mathsf{D}\ \mathbf{in}\ [\mathbf{def}\ z \triangleright 0\ \mathbf{in}\ a\langle z\rangle : Q\{^a/_x\}] \mid [\mathbf{def}\ z \triangleright 0\ \mathbf{in}\ b\langle z\rangle : Q\{^b/_x\}] \\ &\to\ P_3 \equiv \mathbf{def}\ \mathsf{D}\ \mathbf{in}\ [\mathbf{def}\ z \triangleright 0 \wedge z' \triangleright 0\ \mathbf{in}\ ok\langle\rangle : Q\{^a/_x\} | Q\{^b/_x\}] \\ &\to\ P_4 \equiv \mathbf{def}\ \mathsf{D}\ \mathbf{in}\ ok\langle\rangle \end{aligned} \tag{1}$$

Analogously,

$$\begin{aligned} P \to^* P_2 \ &\to\ P_5 \equiv \mathbf{def}\ \mathsf{D}\ \mathbf{in}\ [\mathbf{def}\ z \triangleright 0 \wedge z' \triangleright 0\ \mathbf{in}\ abort : Q\{^a/_x\} | Q\{^b/_x\}] \\ &\to\ P_6 \equiv \mathbf{def}\ \mathsf{D}\ \mathbf{in}\ Q\{^a/_x\} | Q\{^b/_x\} \end{aligned}$$

Assuming Q does not have any active transaction, the computations above are the only two possible evolutions of P to non-transient states, i.e., states that do not contain running transactions. We are aimed at defining a reduction relation $\twoheadrightarrow$ in which any multi-party transaction is described as a single computation step that fetches the messages needed to initiate all cooperating transactions and produces the processes released at commit or abort. In this example, we expect two big-step reductions for P, each of them describing one of the possible executions of the multi-party transaction, namely: $P \twoheadrightarrow P_4$ and $P \twoheadrightarrow P_6$.

We define the big-step reduction relation for a particular class of processes, called *shallow*. Shallow processes are given in terms of a syntactic restriction that imposes a particular discipline for activating transactions. We start by introducing the class of shallow processes.

Definition 7.1 (Nesting level). The *nesting level* (or just *nesting*) of P, written $nest(P)$, is defined by:

$$nest(0) = nest(abort) = nest(x\langle y\rangle) = 0 \qquad nest([P : Q]) = nest(P) + 1$$
$$nest(\mathbf{def}\ D\ \mathbf{in}\ P) = nest(P) \qquad nest(P \mid Q) = \mathsf{max}\{nest(P), nest(Q)\}$$

We remark that $nest(P)$ counts only the nesting level of the active processes independently from the nested transactions that may appear in definitions or compensations. Consider $P \equiv [x\langle y\rangle : 0]$ and $Q \equiv [P : 0]$, then $nest(P) = 1$ and $nest(Q) = nest(P) + 1 = 2$. Contrastingly, $nest([0 : Q]) = 1$ because $[0 : Q]$ is a transaction that has no active sub-transactions (note that the compensation Q is frozen, i.e., inactive).

Definition 7.2 (Shallow and stable processes). A basic definition D is a *shallow definition* if it has one of the following forms

1. $D = J \triangleright P$, where $nest(P) = 0$ or $P = [R : Q] \wedge nest(R|Q) = 0$
2. $D = J \blacktriangleright P$ and $nest(P) = 0$

A process P is *shallow* if any basic definition in P is shallow. Moreover, we call a shallow process P *stable* iff $nest(P) = 0$.

With P and Q as defined above, the process $R \equiv \mathbf{def}\ x\langle y\rangle \triangleright Q\ \mathbf{in}\ x\langle a\rangle$ is stable (i.e., $nest(R) = 0$) because it does not have any active transaction (independently from the fact that it may start a transaction in the future).

Shallowness is a constraint for the syntax of basic definitions contained by a process. Condition 1 in Definition 7.2 ensures that the firing of a basic definition increases the height of the nesting structure by at most one level, i.e., a basic definition produces either a stable process or an activate transaction without any active sub-transaction. Condition 2 forbids the creation of sub-transactions while merging. We remark that shallowness does not impose any constraint on the nesting level of active transactions. For instance, let us consider $Q \equiv [[x\langle y\rangle : 0] : 0]$ and $R \equiv \mathbf{def}\ x\langle y\rangle \triangleright Q\ \mathbf{in}\ x\langle a\rangle$. The process Q trivially satisfy shallowness condition because it does not have any definition. On the contrary, the process R is not shallow because its unique definition $x\langle y\rangle \triangleright Q$ does not satisfy condition 1. In fact, $nest([x\langle y\rangle : 0]|0) = 1 \neq 0$.

We highlight that any flat process is also shallow. It can be easily seen that shallowness is preserved by reductions, while in general this is neither the case for the nesting of a process nor for stability (i.e. any shallow process always reduces to shallow processes, while some stable processes may reduce to non stable processes).

Moreover, it can be shown that any non-shallow definition can be encoded into a shallow definition. Shallowness forbids definitions like $D_0 \equiv x\langle y\rangle \triangleright [[y\langle x\rangle : 0] : 0]$ and, more generally, $D_1 \equiv J_1 \triangleright P \mid [P_1 : Q]$ and $D_2 \equiv J_2 \triangleright [[P_1 : Q_1] : Q]$, they however can be encoded as shallow definitions by using new local ports. The cases above can be rewritten as $D_0' \equiv x\langle y\rangle \triangleright [\mathbf{def}\ z\langle\rangle \triangleright [y\langle x\rangle : 0]\ \mathbf{in}\ z\langle\rangle : 0]$, $D_1' \equiv J_1 \triangleright z\langle\rangle | P \wedge z\langle\rangle \triangleright [P_1 : Q]$ and $D_2' \equiv J_2 \triangleright [\mathbf{def}\ z\langle\rangle \triangleright [P_1 : Q_1]\ \mathbf{in}\ z\langle\rangle : Q]$ with z fresh. Note that the three new definitions are shallow when P, P_1, Q, Q_1 are stable. Then, $\mathbf{def}\ D_1'\ \mathbf{in}\ x\langle a\rangle$ is a shallow process and reduces in two steps to $\mathbf{def}\ D_1'\ \mathbf{in}\ [\mathbf{def}\ z\langle\rangle \triangleright [a\langle x\rangle : 0]\ \mathbf{in}\ [a\langle x\rangle : 0] : 0]$, which is shallow and contains nested transactions.

In the following $\mathcal{P}$ and $\mathcal{Q}$ will denote shallow processes, $\mathcal{D}$ a shallow definition, $\mathcal{S}$ a stable process, and $\mathcal{B}$ a shallow definition containing just merge rules. We abbreviate $\mathbf{def}\ \mathcal{D}\ \mathbf{in}\ \mathcal{P}$ as $\mathcal{D} \vdash \mathcal{P}$, and $\vdash \mathcal{P}$ as $\mathcal{P}$. Terms are considered up-to structural equivalence

$$\text{(PAR)}\ \frac{\mathcal{D} \vdash \mathcal{P} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}' \qquad \mathcal{D} \vdash \mathcal{Q} \twoheadrightarrow \mathcal{D} \vdash \mathcal{Q}'}{\mathcal{D} \vdash \mathcal{P} \mid \mathcal{Q} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}' \mid \mathcal{Q}'} \qquad \text{(SEQ)}\ \frac{\mathcal{D} \vdash \mathcal{P} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}'' \qquad \mathcal{D} \vdash \mathcal{P}'' \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}'}{\mathcal{D} \vdash \mathcal{P} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}'}$$

$$\text{(GLOBAL FIRING)}\ \mathcal{D} \wedge J \triangleright \mathcal{P} \vdash J\sigma \twoheadrightarrow \mathcal{D} \wedge J \triangleright \mathcal{P} \vdash \mathcal{P}\sigma \qquad \text{(LOCAL FIRING)}\ \frac{\widetilde{\mathcal{B}} \vdash \mathcal{S} \twoheadrightarrow \widetilde{\mathcal{B}} \vdash \mathcal{S}'}{\mathcal{D} \wedge \mathcal{B} \vdash [\mathcal{S} : \mathcal{Q}] \twoheadrightarrow \mathcal{D} \wedge \mathcal{B} \vdash [\mathcal{S}' : \mathcal{Q}]}$$

$$\text{(MERGE)}\ \mathcal{D} \wedge \Pi_i J_i \blacktriangleright \mathcal{S} \vdash \Pi_i[\mathcal{D}_i \vdash J_i\sigma \mid \mathcal{S}_i : \mathcal{Q}_i] \twoheadrightarrow \mathcal{D} \wedge \Pi_i J_i \blacktriangleright \mathcal{S} \vdash [\textstyle\bigwedge_i \mathcal{D}_i \vdash (\Pi_i \mathcal{S}_i) | \mathcal{S}\sigma : \Pi_i \mathcal{Q}_i]$$

$$\text{(LOCAL COMMIT)}\ \mathcal{D} \vdash [M \mid \mathcal{D}' \vdash 0 : \mathcal{S}] \twoheadrightarrow \mathcal{D} \vdash M \qquad \text{(ABORT)}\ \mathcal{D} \vdash [abort | \mathcal{P} : \mathcal{S}] \twoheadrightarrow \mathcal{D} \vdash \mathcal{S} \qquad \text{(IDLE)}\ \mathcal{D} \vdash \mathcal{P} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}$$

$$\text{(SERIALIZABLE)}\ \frac{\mathcal{D} \vdash \mathcal{S} \twoheadrightarrow \mathcal{D} \vdash \mathcal{S}'}{\mathcal{D} \vdash \mathcal{S} \twoheadrightarrow \mathcal{D} \vdash \mathcal{S}'}$$

Figure 12. Big-step semantics of cJoin.

generated by closure w.r.t. the equations for the associativity and commutativity of $\mid$ and $\wedge$, 0 the unit for $\mid$, and

$$\mathcal{D} \vdash (\mathcal{P} \mid \mathbf{def}\ \mathcal{D}'\ \mathbf{in}\ \mathcal{Q}) \;=\; \mathcal{D} \wedge \mathcal{D}'\sigma_{dn} \vdash \mathcal{P} \mid \mathcal{Q}\sigma_{dn} \qquad range(\sigma_{dn}) \cap (fn(\mathcal{D}) \cup fn(\mathcal{P}) \cup fn(\mathbf{def}\ \mathcal{D}'\ \mathbf{in}\ \ \mathcal{Q})) = \emptyset$$

The big-step reduction relation $\twoheadrightarrow$ is given by the inference rules in Figure 12 over stable processes. The relation $\twoheadrightarrow$ is defined in terms of the auxiliary relation $\rightarrow\!\!\!\!\triangleright$ over shallow processes. Rule SERIALIZABLE singles out as big steps those computation from stable states to stable states. Note that computation steps can be composed in parallel (PAR) and sequentially (SEQ), even with idle transitions (IDLE). Rule GLOBAL FIRING, abbreviated GF, corresponds to the firing of an ordinary definition in a top-level process. Instead LOCAL FIRING states possible internal transitions of a running transaction. LOCAL FIRING represents suitable sub-transactions as ordinary transitions at an abstract level. In fact, the computations occurring at a lower level in the nesting hierarchy (premise of LOCAL FIRING) that are relevant to its containing transaction are those relating stable processes, i.e., $\mathcal{S}$ and $\mathcal{S}'$. A transaction has available, in addition to its own definitions, the merge definitions introduced by its parent. In fact, a merge definition applied to a single transaction behaves as an ordinary rule but defined in a global scope. The operator $\widetilde{\ }$ transforms merge definitions in ordinary ones: $\widetilde{J \blacktriangleright \mathcal{P}} = J \triangleright \mathcal{P}$ and $\widetilde{\mathcal{B} \wedge \mathcal{B}'} = \widetilde{\mathcal{B}} \wedge \widetilde{\mathcal{B}'}$.

Rules LOCAL COMMIT (abbreviated LC) and ABORT handle the termination of a transaction, whereas MERGE describes the interaction among sibling transactions. This time, transactions can be joined only if they do not contain running transactions.

Example 9. Consider the process P introduced in Example 8. Processes P, P_4 and P_6 are stable and it can be easily checked that $P \twoheadrightarrow P_4$ and $P \twoheadrightarrow P_6$ as expected. A proof for the big-step reduction corresponding to the small-step reduction shown in Equation (1) of Example 8 can be built as follows. First, the small step $P \rightarrow P_1$ corresponds

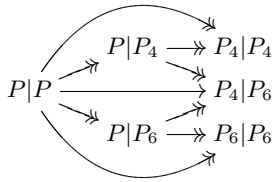

Figure 13. Big-step reductions for the Example 8.

to the following proof

$$\dfrac{\mathsf{D} \vdash c\langle a\rangle \;\twoheadrightarrow\; \mathsf{D} \vdash [\mathbf{def}\ z \triangleright 0\ \mathbf{in}\ a\langle z\rangle : Q\{^a/_x\}]\ \text{GF} \quad \mathsf{D} \vdash c\langle b\rangle \;\twoheadrightarrow\; \mathsf{D} \vdash c\langle b\rangle\ \text{IDLE}}{P = \mathsf{D} \vdash c\langle a\rangle | c\langle b\rangle \;\twoheadrightarrow\; P_1 = \mathsf{D} \vdash [\mathbf{def}\ z \triangleright 0\ \mathbf{in}\ a\langle z\rangle : Q\{^a/_x\}] | c\langle b\rangle}\ \text{PAR}$$

The proof for $P_1 \twoheadrightarrow P_2$ is built analogously. Then, the proof for $P \twoheadrightarrow P_4$ can be completed as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{P \twoheadrightarrow P_1} \quad \dfrac{\vdots}{P_1 \twoheadrightarrow P_2}}{P \twoheadrightarrow P_2}\ \text{SEQ} \quad P_2 \twoheadrightarrow P_3\ \text{MERGE}}{P \twoheadrightarrow P_3}\ \text{SEQ} \quad P_3 \twoheadrightarrow P_4\ \text{GLOBAL COMMIT}}{P \twoheadrightarrow P_4}\ \text{SEQ}}{P \twoheadrightarrow P_4}\ \text{SERIALIZABLE}$$

Consider now the process $P|P$. Figure 13 shows the possible evolutions of $P|P$. The graph illustrates how the result of a computation involving the (possible interleaved) execution of several multi-party transactions, e.g., $P|P \twoheadrightarrow P_4|P_6$, can be also obtained by executing one transaction at a time. For $P|P \twoheadrightarrow P_4|P_6$ we can sequentially compute either $P|P \twoheadrightarrow P|P_4 \twoheadrightarrow P_4|P_6$ or $P|P \twoheadrightarrow P|P_6 \twoheadrightarrow P_4|P_6$.

It is in this sense that the big-step reduction relation enforces serializability and allows us to analyse the behaviour of a set of interacting transactions independently from the rest of the system. Moreover, when considering nested transactions, the transactions completed at a particular level of nesting can be treated as ordinary transitions at the upper level.

The remaining of this section is devoted to show the correspondence between both semantics for shallow processes, which ensures that small-step reductions are serializable for shallow processes.

The next auxiliary result shows that any derivation in cJoin starting from a shallow process without nested transactions has an equivalent cJoin reduction that only merges transactions without ongoing sub-transactions.

Proposition 7.3. If $\mathcal{D}, \mathcal{S}, \otimes_k[\mathcal{S}'_k : \mathcal{S}''_k] \rightarrow^* \mathcal{D}', \otimes_i[\mathcal{P}_i : \mathcal{S}_i] \rightarrow \mathcal{D}', [\mathcal{P} : \|_i \mathcal{S}_i]$, then there exists a derivation $\mathcal{D}, \mathcal{S}, \otimes_k[\mathcal{S}'_k : \mathcal{S}''_k] \rightarrow^* \mathcal{D}', \otimes_i[\mathcal{S}'''_i : \mathcal{S}_i] \rightarrow \mathcal{D}', [\mathcal{S}' : \|_i \mathcal{S}_i] \rightarrow^* \mathcal{D}', [\mathcal{P} : \|_i \mathcal{S}_i]$.

Proof. By induction on the length of the derivation $\mathcal{D}, \mathcal{S}, \otimes_k[\mathcal{S}'_k : \mathcal{S}''_k] \rightarrow^n \mathcal{D}', \otimes_i[\mathcal{P}_i : \mathcal{S}_i]$. The base case follows immediately by taking $\mathcal{S} = \emptyset$ and $k = i$. The inductive step follows by noting first that any reduction $\mathcal{D}, \mathcal{Q} \rightarrow \mathcal{D}', x\langle\vec{v}\rangle, \mathcal{Q}'$ implies either (i) $\mathcal{Q} = x\langle\vec{v}\rangle|\mathcal{Q}''$ or (ii) $\mathcal{Q} = [\mathcal{S}_1 : \mathcal{S}_2]|\mathcal{Q}''$ and $[\mathcal{S}_1 : \mathcal{S}_2] \rightarrow \mathcal{D}'', x\langle\vec{v}\rangle|\mathcal{S}_0$ (this can be shown by case analysis on the applied rule). This property tell us that the generation of a message does not imply the generation of a new transaction. Then, note that the only possibility for the last reduction $\mathcal{D}', \otimes_i[\mathcal{P}_i : \mathcal{S}_i] \rightarrow \mathcal{D}', [\mathcal{P} : \|_i \mathcal{S}_i]$ is due to the application of a merge reaction involving all transactions. Therefore, any $[\mathcal{P}_i : \mathcal{S}_i]$ is such that $\mathcal{P}_i = M_i|\mathcal{P}'_i$. If all $\mathcal{P}_i$ are stable we are done. Otherwise, we proceed by noting that any $[\mathcal{P}_i : \mathcal{S}_i]$ has been generated from the elements of the original solution $\mathcal{D}, \mathcal{S}, \otimes_k[\mathcal{S}'_k : \mathcal{S}''_k]$. The interesting case is when the whole reduction cannot be divided into sequences like $\mathcal{D}, \mathcal{S}, \otimes_k[\mathcal{S}'_k : \mathcal{S}''_k] \rightarrow^* \mathcal{D}'', \mathcal{S}'', \otimes_h[\mathcal{P}_h : \mathcal{P}_h] \rightarrow^* \mathcal{D}', \otimes_i[\mathcal{P}_i : \mathcal{S}_i]$ (these cases can be handled by using inductive hypothesis). Therefore, we note that there exists a partition of $\mathcal{S}, \otimes_k[\mathcal{S}'_k : \mathcal{S}''_k]$ s.t. for each i there exists some $\mathcal{I}$ in the partition and $\mathcal{D}, \mathcal{I} \rightarrow^* \mathcal{D}'', [\mathcal{P}_i : \mathcal{S}_i]$. By using inductive hypothesis we can build $\mathcal{D}, \mathcal{I} \rightarrow^* \mathcal{D}'', \otimes_{k_i}[\mathcal{R}_{k_i} : \mathcal{R}'_{k_i}] \rightarrow \mathcal{D}'', [\mathcal{R} : \mathcal{R}'] \rightarrow^* \mathcal{D}'', [\mathcal{P}_i : \mathcal{S}_i]$. Since $\mathcal{P}_i = M_i|\mathcal{P}'_i$, we can show (by repeatedly using the property that ensures that transactions are not generated together with messages) that $\mathcal{R}_{k_i} = M_i|\mathcal{R}_{k_i}$. Consequently, we can build a reduction that merges first all $[\mathcal{R}_{k_i} : \mathcal{R}'_{k_i}]$, and then reduces to the final configuration. □

In particular, serializable transactions can postpone the activation of each sub-transaction until all other cooperating sub-transactions needed to commit can be activated.

Next results state the correspondences between both semantics.

Lemma 7.4. $\mathcal{D}, \otimes_i[\mathcal{S}_i : \mathcal{P}_i], \mathcal{S} \rightarrow^* \mathcal{D}, \otimes_j[\mathcal{S}'_j : \mathcal{P}'_j], \mathcal{S}'$ if and only if $\mathcal{D} \vdash \|_i[\mathcal{S}_i : \mathcal{P}_i] \mid \mathcal{S} \twoheadrightarrow \mathcal{D} \vdash \|_j[\mathcal{S}'_j : \mathcal{P}'_j] \mid \mathcal{S}'$.

Proof. ($\Rightarrow$) By induction on the length of the derivation. The base case corresponds to the IDLE axiom. The inductive step follows by case analysis of the first applied reduction. The interesting case is the application of RED, which presents two cases: (i) when it is applied at top level, i.e., producing a new transaction or a new stable process, and (ii) when it generates a sub-transaction in one of the existing transactions. First case is immediate by inductive hypothesis and SEQ. In the second case, there is a transaction k, i.e., $[\mathcal{S}_k : \mathcal{P}_k]$, that reduces to $[[\mathcal{S}'' : \mathcal{P}'], \mathcal{S}'_k : \mathcal{P}_k]$. At this point, there are two possibilities. First, consider that $[[\mathcal{S}'' : \mathcal{P}'], \mathcal{S}'_k : \mathcal{P}_k]$ reduces to $[\mathcal{S}''_k : \mathcal{P}_k]$, then it is possible to build the proof shown in Figure 14, and then the proof follows by using IDLE for the non-modified processes and inductive hypothesis and SEQ. The remaining possibility is when sub-transaction $[\mathcal{S}'' : \mathcal{P}']$ can finish only after the parent transaction is merged. Then by Proposition 7.3 an equivalent derivation $\mathcal{D}, \otimes_i[\mathcal{S}_i : \mathcal{P}_i], S \rightarrow^* \mathcal{D}, [S'' : \|_i \mathcal{P}_i], S''' \rightarrow^* \mathcal{D}, [\mathcal{P} : \|_i \mathcal{P}_i], S''' \rightarrow^* \mathcal{D}, \otimes_j[\mathcal{S}'_j : \mathcal{P}'_j], \mathcal{S}'$ that merges only transactions not containing sub-transactions can be found. Then, proof follows by applying inductive hypothesis for

$$\dfrac{\dfrac{\dfrac{\vdots}{\widetilde{\mathcal{B}} \vdash \mathcal{S}_k \twoheadrightarrow \widetilde{\mathcal{B}} \wedge \mathcal{D}'' \vdash [\mathcal{S}'' : \mathcal{P}'] \mid \mathcal{S}'_k}\ \text{GLOBAL FIRING,IDLE,PAR} \quad \widetilde{\mathcal{B}} \wedge \mathcal{D}'' \vdash [\mathcal{S}'' : \mathcal{P}'] \mid \mathcal{S}'_k \twoheadrightarrow \widetilde{\mathcal{B}} \vdash \mathcal{S}''_k\ \text{IND. HYP.}}{\widetilde{\mathcal{B}} \vdash \mathcal{S}_k \twoheadrightarrow \widetilde{\mathcal{B}} \vdash \mathcal{S}''_k}\ \text{SEQ}}{\mathcal{D}' \wedge \mathcal{B} \vdash [\mathcal{S}_k : \mathcal{P}_k] \twoheadrightarrow \mathcal{D}' \wedge \mathcal{B} \vdash [\mathcal{S}''_k : \mathcal{P}_k]}\ \text{LOCAL FIRING}$$

Figure 14. Proof sketch for Lemma 7.4.

both parts $\mathcal{D}, \otimes_i [\mathcal{S}_i : \mathcal{P}_i], S \to^* \mathcal{D}, [S'' : \|_i \mathcal{P}_i], S'''$ and $\mathcal{D}, [S'' : \|_i \mathcal{P}_i], S''' \to^* \mathcal{D}, [\mathcal{P} : \|_i \mathcal{P}_i], S''' \to^* \mathcal{D}, \otimes_j [\mathcal{S}'_j : \mathcal{P}'_j], \mathcal{S}'$.

($\Leftarrow$) By induction on the structure of the proof. □

Theorem 7.5. Let $\mathcal{S}, \mathcal{S}'$ be stable processes. Then $\mathcal{S} \to^* \mathcal{S}'$ iff $\vdash \mathcal{S} \twoheadrightarrow \vdash \mathcal{S}'$.

Proof. Immediate by Lemma 7.4. □

An informal explanation of the serializability result can be given by colouring transaction scopes and reductions as explained below. Let $\mathcal{S}, \mathcal{S}'$ be stable processes such that there exists $P_0, P_1, ..., P_k$ with $\mathcal{S} \equiv P_0 \to P_1 \to ... \to P_k \equiv \mathcal{S}'$. We traverse the computation backward, one reduction $P_i \to P_{i+1}$ at the time. If the reduction is originated from a commit or abort, then we assign it a fresh colour and use the same colour to paint the brackets of the corresponding transaction scope in P_i. If the reduction is originated from the merge of several transactions, then we paint the reduction and all the involved transaction scopes in P_i with the same colour as the merged transaction scope in P_{i+1}. If the reduction is an ordinary one, then we paint it with the same colour as the one of the immediately enclosing brackets of its target term in P_{i+1} and we use the same colour to paint the immediately enclosing brackets of its source term in P_i, if any. At each step, we paint with the same colours as in P_{i+1} all the transaction scopes that are not directly involved in the step. Then each colour can be viewed as representing activities of a distinct transaction, and we say that c_1 is a sub-colour of c_2 if the transaction associated with c_1 is a sub-transaction of the one associated with c_2. The serializability result essentially guarantees that another sequence of reductions $\mathcal{S} \equiv Q_0 \to Q_1 \to ... \to Q_k \equiv \mathcal{S}'$ can be found, such that all steps of different colours are either contiguous or separated by steps of some sub-colour.

8. Related Work and Concluding Remarks

We have proposed cJoin as a formal framework for designing and programming multiparty LRTs. Our calculus features name mobility, asynchronous communication and has a prototype implementation called t-JoCaml. We have included several examples that witness the flexibility of the calculus, together with a serializability result that hold for a wide range of processes, called shallow. The encodability of full cJoin in Join is an open issue, for which we have found no solution yet, because it would require the implementation of a scoping discipline for restricting communication over distributed processes. Nevertheless,

in Sections 3 and 4 we have shown that the implemented fragment is valuable enough to model a large variety of frequently used patterns. Section 8.1 also witnesses that our proposal is quite original w.r.t. other ones found in the literature.

Transactions have been largely studied by the database community as the main mechanism for ensuring data consistency when executing concurrent sequences of operations (Eswaran et al., 1976). Many different models were proposed to meet the so-called ACID properties (i.e., all or nothing transactions) such as the *flat model* (Eswaran et al., 1976), *flat transactions with save-points* and *chained transactions* (Gray and Reuter, 1993), *nested transactions* (Moss, 1981) and the *multi-level model* (Lomet, 1992; Schek and Weikum, 1992; Weikum, 1991) among others. These models are based on locking mechanisms that prevent concurrent transactions from accessing shared objects simultaneously (Bernstein et al., 1987; Kohler, 1981; Fekete et al., 1994; Gray and Reuter, 1993). Hence, the execution of a transaction may suffer considerable delays while waiting for others transactions to commit. Therefore, ACID transactions are regarded suitable only for handling transactions with short duration. Alternative models for long running transactions leave out ACID properties by relying on weaker notions of atomicity. The seminal proposal in this direction is Sagas (Garcia-Molina and Salem, 1987), which introduces the model of multistep transactions with programmed compensations described in Section 4.1. This model has been generalised by *Open Nested transactions* (Schek and Weikum, 1992), in which compensable steps can be organised hierarchically. Several other models have appeared in the literature for allowing a flexible description of steps dependencies, such in (Kaiser and Pu, 1992; Hutchinson et al., 1988; Elmagarmid et al., 1990).

As previously mentioned, the main goal of the above research line is to ensure consistent database updates. Differently, transactions in composition languages are aimed at coordinating atomic executions of independent activities. In this sense, composition languages are directly related to workflow management systems (WMS) (see (Georgakopoulos et al., 1995; Rusinkiewicz and Sheth, 1995) for a detailed description of WMS). Transactional execution of workflows has been an active research topic for WMS community, as testified by several existing transactional WMS, like Contracts (Reuter and Wächter, 1992), METEOR (Krishnakumar and Sheth, 1995) or METEOR2 (Kochut et al., 1996)). Recently, the works in (Butler et al., 2002; Butler et al., 2005b; Bruni et al., 2005; Butler et al., 2005a), further improved in (Ripon, 2008; Bruni et al., 2011), have formally studied the semantics of orchestration languages with transactions. Differently from workflow community, this new research line is concerned about workflow systems as programming languages, and thus the focus is on giving precise, formal syntax, semantics and reasoning techniques for transaction primitives in orchestration.

Like previous approaches, cJoin is aimed at using programming language approach to study transactional composition. Nevertheless, it is targeted to the study of transactions for choreographies by adding transactions to a calculus for communicating processes. We devote the remaining part of this section to compare cJoin with other proposals from the literature that have similar aims.

8.1. *Language Comparison*

In order to systematically analyse the proposals appeared in the literature we first identify a set of aspects or choices for extending communicating processes calculi with transactions. Then, we present a comparison of the different approaches in terms of those selected features.

8.1.1. *Undoability.* A transactional mechanism provides a way for repairing the effects of the partial executions of aborted transactions. There are three main design options:

1 *Automatic Roll-back*: If a transaction $[P]$ aborts, then the scope $[\_]$ ensures that all the effects of the partial execution of P are automatically removed from the state.
2 *Static Programmable Roll-back*: A transactional process P is associated statically with another process Q such that Q is activated whenever P aborts. Programmers are responsible for writing Q in such a way that the effects of the partial executions of P are compensated for.
3 *Dynamic Programmable Roll-back*: Differently from static programmable roll-back, compensations are built during execution. Programmers are responsible for describing how compensations change when transactions execute.
4 *Pre-committed Compensation*: A LRT is divided into several steps. The successful execution of a step (i.e., it commits) may install ad hoc programs to be run only when the whole LRT aborts.

8.1.2. *Permeability.* Permeability refers to the degree of isolation provided by transactional scopes, i.e., whether messages can cross transactional scopes or not.

1 *Impermeable*: Messages cannot flow across transactional boundaries, e.g., in $P \equiv x!z.0 \mid y?v.0 \mid [\, x?w.P \mid y!z.Q \,]$ no communication is possible on x and y.
2 *Permeable in input*: Messages generated outside transactions can be received by transactional processes, e.g., process P above can reduce by communicating over x but not over y.
3 *Permeable in output:* Transactional processes may send messages to receivers outside the transaction. In the previous example, P can communicate on y but not on x.
4 *Permeable:* Messages freely flow across transactional boundaries. The previous process P may communicate on both x and y.
5 *Selective Permeability:* Messages may flow across transactional boundaries only when sent over channels that are in some particular class.

8.1.3. *Dynamicity.* Dynamicity characterizes the way in which the execution of a transactional scope may relate with the execution of other scopes.

1 *Static*: A static scope has no relation with other scopes, i.e., after being created it can neither affect nor be affected by the behaviour of other scopes.
2 *Joinable*: A scope is joinable if its abortion or commitment may condition or may be conditioned by the abortion or commitment of other scopes.
3 *Splittable*: A scope $[P]$ is splittable if it is possible to take a part of P and run it as an independent scope.

4 *Dynamic*: A dynamic scope is both joinable and splittable.

8.1.4. *Naming.* The naming policy indicates the way in which transactional scopes are identified. In particular, they can be

1 *Anonymous*: Scopes have no explicit identification. This mean that transactional processes do not refer scopes explicitly.
2 *Named*: Scopes have a name and processes refer to them explicitly. Usually a transaction is aborted by sending a message to its scope name. In addition, scope names may be *unique*, i.e., a name unequivocally identify a transactional scope; or *multiple*, i.e., a name may refer to several transactional scopes.

8.1.5. *Interaction model.* Transactional process calculi differ on the underlying interaction model. We use here a coarse-grain distinction in two main categories.

1 *Shared Dataspaces:* Processes communicate by writing to and reading from a shared blackboard.
2 *Message Passing:* Processes communicate by sending and receiving messages on specific ports or channels.

8.1.6. *Nesting.* Nesting relates to the capability of decomposing the execution of a transaction into a hierarchy of *sub-transactions*. In this scheme, any sub-transaction executes atomically and concurrently with respect to its parent and siblings, deciding freely to commit or abort. Nevertheless, if the parent aborts all its sub-transactions are also undone.

8.1.7. *Preemption.* Following the classification in (Berry, 1993), the abortion of an execution may take two different styles of preemption. Abortion may be (i) *may*-preemptive, i.e., a transaction may take an arbitrary number of internal computation steps before handling the abort condition, or (ii) *must*-preemptive, i.e., no further internal computation steps are allowed when a transaction reaches the abort (abortion is honoured immediately).

8.1.8. *Comparison.* We will use the previous seven categories to compare cJoin against the closest process calculi appeared in the literature: PLinda (Anderson and Shasha, 1992), TSpaces (Busi and Zavattaro, 2002), TraLinda (Bruni and Montanari, 2004), πt (Bocchi et al., 2003), Webπ_∞ (Lucchi and Mazzara, 2004), RCCS (Danos and Krivine, 2004), $\rho\pi$ (Lanese et al., 2010a), $dc\pi$ (Vaz et al., 2008) and TransCCS (de Vries et al., 2010). Figure 15 summarises the features of all selected approaches. We remark that all considerations made for Webπ_∞ are still valid for its timed version (Laneve and Zavattaro, 2005).

Languages PLinda and TSpaces are aimed at providing a model for traditional serializable (i.e., ACID) transactions, hence they provide *input* permeability by allowing a transaction to read or to consume data from a shared dataspace (i.e., the communication models represents a shared database). Differently, when the transaction produces

	Undoability	Permeability	Dynamicity	Naming	Interaction	Nesting	Preemption
PLinda	*Automatic*; *Static*	*Input*	*Static*	*Anonym.*	*DataSpaces*	*No*	*May*
TSpaces	–	*Input*	*Static*	*Unique*	*DataSpaces*	*No*	–
TraLinda	*Automatic*	*Impermeable*	*Joinable*	*Anonym.*	*DataSpaces*	*No*	*May*
πt	*Static*; *Precomm.*	*Permeable*	*Static*	*Anonym.*	*Msg Passing*	*Yes*	*May*
webπ_∞	*Static*	*Permeable*	*Splittable*	*Multiple*	*Msg Passing*	*No*	*May*
RCCS	*Automatic*	*Permeable*	*Joinable*	*Anonym.*	*Msg Passing*	*No*	–
$\rho\pi$	*Automatic*	*Permeable*	*Joinable*	*Anonym.*	*Msg Passing*	*No*	–
$dc\pi$	*Dynamic*	*Permeable*	*Static*	*Unique*	*Msg Passing*	*Yes*	*Must*/ *May*
TransCCS	*Automatic* *Static*	*Permeable*	*Joinable*	*Unique*	*Msg Passing*	*Yes*	–
cJoin	*Static*	*Selective*	*Joinable*	*Anonym.*	*Msg Passing*	*Yes*	*May*

Figure 15. Comparison of Transactional Process Calculi

a new datum, it is locked until the transaction commits and hence there is no *output* permeability. In both cases transaction scopes are defined statically and cannot change dynamically. Transactions have no name in PLinda, while they are named in TSpaces. Names in TSpaces are included as a handy way for defining the semantics of the language, nevertheless programmers do not need to be aware of them. Although the syntax of TSpaces allows transaction names to be duplicated, its operational semantics ensures that each transactional name is treated as unique. None of those languages allows nesting (the syntax of TSpaces allows it but its semantics does not). Transactions in PLinda have an automatic, perfect roll-back. Moreover, programmers have the possibility of specifying ad hoc programs to be run after roll-back when a transaction aborts. Abortion is not considered in TSpaces.

TraLinda adds joinable transactions to a shared dataspace coordination language. Multi-way transactions of cJoin are analogous to TraLinda. Nevertheless, cJoin consider programmable compensations instead of perfect roll-back, nested transactions instead of flat ones, and message passing communication instead of shared dataspace.

Both $\pi\mathtt{t}$ and Webπ_∞ are similar in spirit to cJoin. The main differences rely on the policies adopted for transactional scopes. While $\pi\mathtt{t}$ and Webπ_∞ allows transactional processes to freely interact with other processes, cJoin imposes a more strict policy: transactional processes may interact over selected channels only with transactional process but, in this case, they should reach the same decision (i.e., commit or abort). These differences come from the fact that scopes in $\pi\mathtt{t}$ and Webπ_∞ are permeable, while they are selective permeable and joinable in cJoin. All three languages provide a mechanism for programmable compensation (it is called fault handler in $\pi\mathtt{t}$). In addition, $\pi\mathtt{t}$ provides a mechanism for undoing precommitted subtransactions. Both $\pi\mathtt{t}$ and cJoin have nesting, while Webπ_∞ has not. Word nesting is used in Webπ_∞ to refer to splittable scopes.

RCCS and $\rho\pi$ provide transactions relying on a built-in distributed backtracking mechanism, which can achieve perfect roll-back. Transactions are joinable in the sense that processes that have communicated are required to backtrack together. RCCS is an extension of CCS (Milner, 1980), therefore the underlying communication is just process synchronisation. On the contrary, $\rho\pi$ supports higher order communication (in fact, $\rho\pi$ extends RCCS to higher-order π). Differently from cJoin, RCCS and $\rho\pi$ do not support nesting, scopes are permeable, and transactions are automatically rolled-back. Abortion can be fired spontaneously (the execution of a transaction can be aborted at any time).

Compensations in cJoin are statically defined while they are dynamically built in dcπ. The main idea behind dcπ is that any input prefix is associated with a compensation. Then, any time a process executes an input action, it also installs a compensation that will be activated if the corresponding transaction aborts. As shown in (Lanese et al., 2010b), dynamic compensations are more expressive than static compensations. Transactions in dcπ are completely permeable and static, like in $\pi\mathtt{t}$ and Webπ_∞. Hence, the commitment or abortion of one transaction does not affect the behaviour of the others. As for Webπ_∞, transactional scopes are named and their names are used for signalling abortion. Consequently, transaction in dcπ can be aborted internally or externally, while in cJoin the abort condition can be reached only internally. Moreover, dcπ adopts may-preemption for handling abortion generated externally and must-preemption for internal abort. On the contrary, aborts are only internal and may-preemptive in cJoin. The main reason for this choice is that a transaction in cJoin can be the consequence of merging several independent, possible distributed transactions. Thus, the implementation of a must-preemptive abortion would be problematic in a distributed setting without central coordination.

The only joinable mechanism for transactions that we are aware of is the one proposed in TransCCS. As in cJoin, a transaction in TransCCS can be merged with other transactions during execution. Merging is completely symmetric in cJoin, i.e., the abort of a merged transaction releases the compensations corresponding to all original transactions. Differently, the merging of transactions in TransCCS generates a nested transaction (i.e., one transaction is included as sub-transaction of the other). Then, the abortion of one transaction in this hierarchy releases the compensation of the original transaction and automatically rolls back the original state of the transactions that has been included as a subtransaction. Finally, abort is spontaneous in TransCCS.

We remark that other interesting approaches such as Pike (Chothia and Duggan, 2004)

and Transactional Linda (Jagannathan and Vitek, 2004) have been left out from our discussion, since they are parametric frameworks in which the behaviour of a transaction does not rely on language primitives. Roughly, transactional processes in those calculi are associated with particular structures that record all process activities. Then, before granting a process the possibility of executing an action, the requested action is checked against the execution history to determine whether it will preserve consistency or not. Hence, different log definitions (in particular, the inference rules that check consistency) can provide different flavours of transactions.

Similarly we leave out of the comparison the interesting work in (Bocchi and Tuosto, 2010), where the basis are set for a theory of testing equivalence for distributed transactions in the presence of transactional attribute (inspired by the Java Transaction API). Transaction attributes discipline how services are executed with respect to the transactional scope of the invoking party. However, the calculus proposed in (Bocchi and Tuosto, 2010) does define a notion of commit, but mostly focuses on compensation handling.

Acknoweledgements We want to thank Nick Benton, Luca Cardelli, Cédric Fournet and Cosimo Laneve with whom we discussed preliminary versions of cJoin. We are very much indebted to the anonymous reviewers of this special issue for their careful revisions and detailed comments that helped us to improve the presentation and eliminate some technical inaccuracies. Finally, we thank Ivan Lanese and Davide Sangiorgi, the editors of this special issue, for inviting us to submit this contribution.

References

Anderson, B. and Shasha, D. (1992). Persistent linda: Linda + transactions + query processing. In *Research Directions in High-Level Parallel Programming Languages*, pages 93–109. Springer Verlag.

Benton, N., Cardelli, L., and Fournet, C. (2002). Modern concurrency abstractions for C♯. In *Proceedings of ECOOP 2002*, volume 2374 of *Lect. Notes in Comput. Sci.*, pages 415–440. Springer Verlag.

Bernstein, P., Hadzilacos, V., and Goodman, N. (1987). *Concurrency, Control and Recovery in Database Systems.* Addison-Wesley Longman.

Berry, G. (1993). Preemption in concurrent systems. In *Proceedings of FSTTCS'93*, volume 761 of *Lect. Notes in Comput. Sci.*, pages 72–93. Springer Verlag.

Berry, G. and Boudol, G. (1992). The chemical abstract machine. *Theoret. Comput. Sci.*, 96(1):217–248.

Bocchi, L., Laneve, C., and Zavattaro, G. (2003). A calculus for long-running transactions. In *Proceedings of FMOODS'03*, volume 2884 of *Lect. Notes in Comput. Sci.*, pages 124–138. Springer Verlag.

Bocchi, L. and Tuosto, E. (2010). Testing attribute-based transactions in SOC. In *Proceedings of FMOODS/FORTE 2010*, volume 6117 of *Lect. Notes in Comput. Sci.*, pages 87–94. Springer Verlag.

Boreale, M., Bruni, R., De Nicola, R., and Loreti, M. (2008). Sessions and pipelines for structured service programming. In Barthe, G. and de Boer, F. S., editors, *Proceedings of FMOODS'08*, volume 5051 of *Lect. Notes in Comput. Sci.*, pages 19–38. Springer Verlag.

BPEL (2003). BPEL Specification. version 1.1. Available at `http://www.ibm.com/developerworks/library/ws-bpel`.

BPMN (2010). Business process modelling notation (BPMN). Available at `http://www.bpmi.org`.

Bruni, R., Kersten, A., and Lanese, I. (2011). A new strategy for distributed compensations with interruption in long-running transactions. In *Proceedings of WADT 2010*, Lect. Notes in Comput. Sci. Springer Verlag. To appear.

Bruni, R., Laneve, C., and Montanari, U. (2002). Orchestrating transactions in join calculus. In *Proceedings of CONCUR 2002*, volume 2421 of *Lect. Notes in Comput. Sci.*, pages 321–336. Springer Verlag.

Bruni, R., Melgratti, H., and Montanari, U. (2003). Flat committed join in join. In *Proceedings of CoMeta 2003*, volume 104 of *Elect. Notes in Th. Comput. Sci.*, pages 39–59. Elsevier Science.

Bruni, R., Melgratti, H., and Montanari, U. (2004). Nested commits for mobile calculi: extending Join. In *Proceedings of the 3rd IFIP-TCS 2004*, pages 569–582. Kluwer Academic Publishers.

Bruni, R., Melgratti, H., and Montanari, U. (2005). Theoretical foundations for compensations in flow composition languages. In *Proceedings of POPL 2005*, pages 209–220. ACM Press.

Bruni, R. and Montanari, U. (2004). Concurrent models for linda with transactions. *Math. Struct. in Comput. Sci.*, 14(3):421–468.

Busi, N. and Zavattaro, G. (2002). On the serializability of transactions in shared dataspaces with temporary data. In *Proceedings of SAC 2002)*, pages 359–366. ACM Press.

Butler, M., Bruni, R., Ferreira, C., Hoare, T., Melgratti, H., and Montanari, U. (2005a). Comparing two approaches to compensable flow composition. In *Proceedings of CONCUR 2005*, volume 3653 of *Lect. Notes in Comput. Sci.*, pages 383–397. Springer Verlag.

Butler, M., Chessell, M., Ferreira, C., Griffin, C., Henderson, P., and Vines, D. (2002). Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4):743–758.

Butler, M. and Ferreira, C. (2004). An operational semantics for StAC, a language for modelling long-running business transactions. In *Proceedings of Coordination 2004*, volume 2949 of *Lect. Notes in Comput. Sci.*, pages 87–104. Springer Verlag.

Butler, M., Hoare, T., and Ferreira, C. (2005b). A trace semantics for long-running transactions. In *Proceedings of 25 Years of CSP*, volume 3525 of *Lect. Notes in Comput. Sci.*, pages 133–150. Springer Verlag.

Caires, L., Ferreira, C., and Vieira, H. T. (2009). A process calculus analysis of compensations. In Kaklamanis, C. and Nielson, F., editors, *Proceedings of TGC'08*, volume 5474 of *Lect. Notes in Comput. Sci.*, pages 87–103. Springer Verlag.

Chothia, T. and Duggan, D. (2004). Abstractions for fault-tolerant global computing. *Theor. Comput. Sci.*, 322(3):567–613.

Conchon, S. and Le Fessant, F. (1999). Jocaml: Mobile agents for Objective-Caml. In *Proceedings of ASA/ MA'99*, pages 22–29. IEEE Computer Society.

Danos, V. and Krivine, J. (2004). Reversible communicating systems. In *Proceedings of CONCUR 2004*, volume 3170 of *Lect. Notes in Comput. Sci.*, pages 293–307. Springer Verlag.

de Vries, E., Koutavas, V., and Hennessy, M. (2010). Communicating transactions - (extended abstract). In Gastin, P. and Laroussinie, F., editors, *Proceedings of CONCUR'10*, volume 6269 of *Lect. Notes in Comput. Sci.*, pages 569–583. Springer Verlag.

Eisentraut, C. and Spieler, D. (2009). Fault, compensation and termination in ws-bpel 2.0 - a comparative analysis. In Bruni, R. and Wolf, K., editors, *Proceedings of WS-FM'08*, volume 5387 of *Lect. Notes in Comput. Sci.*, pages 107–126. Springer Verlag.

Elmagarmid, A., Leu, Y., Litwin, W., and Rusinkiewicz, M. (1990). A multidatabase transaction model for interbase. In *Proceedings of VLDB'90*, pages 507–518. Morgan Kaufmann.

Eswaran, K., Gray, J., Lorie, R., and Traiger, I. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633.

Fekete, A., Lynch, N., Merritt, M., and Weihl, W. (1994). *Atomic Transactions*. Morgan Kaufmann Publishers.

Fournet, C. and Gonthier, G. (1996). The reflexive chemical abstract machine and the Join calculus. In *Proceedings of POPL'96*, pages 372–385. ACM Press.

Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., and Rémy, D. (1996). A calculus of mobile agents. In *Proceedings of CONCUR'96*, volume 1119 of *Lect. Notes in Comput. Sci.*, pages 406–421. Springer Verlag.

Garcia-Molina, H. and Salem, K. (1987). Sagas. In *Proceedings of the ACM Special Interest Group on Management of Data Annual Conference*, pages 249–259. ACM Press.

Georgakopoulos, D., Hornick, M., and Sheth, A. (1995). An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153.

Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Hutchinson, N., Kaiser, G., and Pu, C. (1988). Split-transactions for open-ended activities. In *Proceedings of VLDB'88*, pages 26–37. Morgan Kaufmann.

Jagannathan, S. and Vitek, J. (2004). Optimistic concurrency semantics for transactions in coordination languages. In *Proceedings of COORDINATION 2004*, volume 2949 of *Lect. Notes in Comput. Sci.*, pages 183–198. Springer Verlag.

Kaiser, G. and Pu, C. (1992). Dynamic restructuring of transactions. In *Database Transaction Models for Advanced Applications*, pages 265–295. Morgan Kaufmann.

Kochut, K., Miller, J., Sheth, A., and Wang, X. (1996). Corba-based run-time architectures for workflow management systems. *Journal of Database Management, Special Issue on Multidatabases*, 7(1):16–27.

Kohler, W. (1981). A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys*, 13(2):149–183.

Krishnakumar, N. and Sheth, A. (1995). Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2):155–186.

Lanese, I., Mezzina, C., and Stefani, S. (2010a). Reversing higher-order pi. In *Proceedings of CONCUR'10*, volume 6269 of *Lect. Notes in Comput. Sci.*, pages 478–493. Springer Verlag.

Lanese, I., Vaz, C., and Ferreira, C. (2010b). On the expressive power of primitives for compensation handling. In *Proceedings of ESOP'10*, volume 6012 of *Lect. Notes in Comput. Sci.*, pages 366–386. Springer Verlag.

Laneve, C. and Zavattaro, G. (2005). Foundations of web transactions. In *Proceedings of FOSSACS 2005*, volume 3441 of *Lect. Notes in Comput. Sci.*, pages 282–298. Springer Verlag.

Leymann, F. (2001). WSFL Specification. version 1.0. Available at `http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

Lomet, D. (1992). MLR: A recovery method for multi-level systems. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 185–194. ACM Press.

Lucchi, R. and Mazzara, M. (2004). A framework for generic error handling in business processes. In *Proceedings of WS-FM 2004*, volume 105 of *Elect. Notes in Th. Comput. Sci.*, pages 133–145. Elsevier Science.

Melgratti, H. (2005). *Models and Languages for Global Computing Transactions*. PhD thesis, Computer Science Department, University of Pisa.

Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lect. Notes in Comput. Sci.* Springer Verlag.

Milner, R., Parrow, J., and Walker, J. (1992). A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77.

Moss, J. (1981). *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT.

Reuter, A. and Wächter, H. (1992). *Transaction Models for Advanced Applications*, chapter The Contract Model, pages 219–263. Morgan Kaufmann.

Ripon, S. (2008). *Extending and Relating Semantic Models of Compensating CSP.* PhD thesis, School of Electonics and Computer Science, University of Southampton.

Rusinkiewicz, M. and Sheth, A. (1995). Specification and execution of transactional workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pages 592–620. ACM Press and Addison-Wesley.

Schek, H.-J. and Weikum, G. (1992). Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann.

Thatte, S. (2001). XLANG: Web Services for Business Process Design. Available at `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`.

Vaz, C., Ferreira, C., and Ravara, A. (2008). Dynamic recovering of long running transactions. In *Proceedings of TGC 2008*, volume 5474 of *Lect. Notes in Comput. Sci.*, pages 201–215. Springer Verlag.

Weikum, G. (1991). Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180.

WSCDL (2004). Web Services Choreography Description Language. Version 1.0. Available at `http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/`.

WSCI (2002). WSCI Specification. version 1.0. Available at `http://www.w3.org/TR/wsci/`.

Appendix A. Correctness and completeness of the implementation

Regarding the proofs of the theorems stating the correctness and completeness of the encoding of flat cJoin in Join, we report here the details omitted from the proof sketches in Section 5.

A.1. *Correctness, part 1*

Proof of Theorem 5.3. The proof follows by case analysis on P. Note that P cannot be of the form $[P' : Q']$ because $\not\vdash [P' : Q'] : \Box_1$. If P has no local definitions (i.e., $P \not\equiv \mathbf{def}\ D\ \mathbf{in}\ M$), then P is either *abort*, the inert process 0, or the parallel composition of messages (containing only free names because there are no local definitions). In all three cases, $P' = P$. Last case is when P contains local definitions, i.e., $P \equiv \mathbf{def}\ D\ \mathbf{in}\ M$. We show that for any derivation $P \rightarrow_{cJ}^{*} P_1$ the following two conditions hold:

— $P_1 \equiv \mathbf{def}\ D'\ \mathbf{in}\ M' \mid \Pi_{i\in 1..n}\mathcal{N}_i$, where $\mathcal{N}_i$ are cJoin transactions,

— $\exists Q_1$ s.t. $[\![P]\!] \rightarrow_{J}^{*} Q_1$ and $Q_1 \equiv (\mathbf{def}\ [\![D']\!]\ \mathbf{in}\ [\![M']\!] | \Pi_{i\in 1..n} R_i) | \mathbf{def}\ D_g\ \mathbf{in}\ 0$, where each R_i is the standard Join negotiation associated to $\mathcal{N}_i$. Besides, D_g are garbage definitions corresponding to instances of the commit protocol that have terminated.

The proof follows by induction on the length of the derivation $P \rightarrow_{cJ}^{*} P_1$. Base case ($m = 0$) follows immediately (since $P_1 = P$, it is enough to take $Q_1 = [\![P]\!]$). For the inductive step ($m = k + 1$) we consider reductions $P \rightarrow_{cJ}^{k} P_1' \rightarrow_{cJ} P_1$. By inductive hypothesis on $P \rightarrow_{cJ}^{k} P_1'$ we know that

1 $P_1' \equiv \mathbf{def}\ D''\ \mathbf{in}\ M'' \mid \Pi_{i\in 1..n''}\mathcal{N}_i''$,
2 $\exists Q_1'$ s.t. $[\![P]\!] \rightarrow_J^{*} Q_1'$ and $Q_1' \equiv (\mathbf{def}\ [\![D'']\!]\ \mathbf{in}\ [\![M'']\!] | \Pi_{i\in 1..n''} R_i'')|\mathbf{def}\ D_g''\ \mathbf{in}\ 0$, where each R_i'' is the standard Join negotiation associated to $\mathcal{N}_i''$.

We proceed by case analysis on the applied rule for $P_1' \rightarrow_{cJ} P_1$. Interesting cases are those that terminate a cJoin transaction, i.e., when rule (COMMIT) or (ABORT) are used. Rule (COMMIT) can be only applied when P_1' has a transaction $\mathcal{N}_1$ that does not contain *abort*, messages to local ports nor messages with local names as parameters. In this case, there exists M_1'' such that $\mathcal{N}_1'' \rightarrow_{cJ} M_1''$ and $P_1 \equiv \mathbf{def}\ D''\ \mathbf{in}\ M'' \mid M_1'' \mid \Pi_{i\in 2..n''}\mathcal{N}_i''$. By definition of a standard Join negotiation, R_1'' is such that all coordinators have been asked to commit and the set of continuations of such coordinators is $[\![M_1'']\!]$. We rely on a commit protocol that is ensured to terminate by releasing all continuations when all coordinators are asked to commit, then $R_1'' \rightarrow^{*} [\![M_1'']\!] \mid \mathbf{def}\ D_{g1}\ \mathbf{in}\ 0$. Therefore, $Q_1' \rightarrow^{*} Q_1 \equiv (\mathbf{def}\ [\![D'']\!]\ \mathbf{in}\ [\![M'']\!]|[\![M_1'']\!]|\Pi_{i\in 2..n''} R_i'')|\mathbf{def}\ D_g''\ \mathbf{in}\ 0|\mathbf{def}\ D_{g1}\ \mathbf{in}\ 0$. Then, it is enough to take $D' = D''$, $M' = M''|M_1''$, $n = n'' - 1$ with $\mathcal{N}_i' = \mathcal{N}_{i+1}''$ and $R_i' = R_{i+1}''$, and $D_g = D_g'' \wedge D_{g1}$.

The case for (ABORT) follows analogously. □

A.2. *Completeness*

Proof of Theorem 5.5. We proceed by case analysis on the structure of P. Since $\vdash P : \Box_1$, then $P \not\equiv [P' : Q']$. When P has no local definitions, then it is the parallel composition of messages on free ports, the inert process 0 and *abort*. For any of these cases it holds that $[\![P]\!]$ does not have any definition, and therefore $[\![P]\!]$ cannot reduce. The only possibility is $Q = norm(Q) = [\![P]\!]$, which trivially satisfies $\forall x : Q \downarrow_x \Rightarrow [\![P]\!] \downarrow_x$. If $P \equiv \mathbf{def}\ D\ \mathbf{in}\ M$, then we show that the following three conditions hold:

1 $Q \equiv \mathbf{def}\ [\![D']\!]\ \mathbf{in}\ [\![M_1']\!] \mid \Pi_{i\in 1..u} R_i' \mid \Pi_{k\in 1..f} T_k' \mid \mathbf{def}\ D_g\ \mathbf{in}\ 0$, where R_i' are unfinished Join negotiations (i.e., some transactional thread has not finished), while T_k' are finished negotiations, with $norm(\Pi_{k\in 1..f} T_k') \equiv [\![M_2]\!]|\mathbf{def}\ D_c\ \mathbf{in}\ 0$.
2 $P \rightarrow_{cJ}^{*} P' \equiv \mathbf{def}\ D'\ \mathbf{in}\ M_1' \mid M_2 \mid \Pi_{i\in 1..u}\mathcal{N}_i$ where $\mathcal{N}_i$ is a standard cJoin transaction corresponding to R_i'.
3 $norm(Q) \equiv \mathbf{def}\ [\![D']\!]\ \mathbf{in}\ [\![M_1'|M_2]\!] \mid norm(\Pi_{i\in 1..u} R_i') \mid \mathbf{def}\ D_g'\ \mathbf{in}\ 0$

Above conditions can be shown by induction on the length of the derivation $[\![P]\!] \rightarrow_J^{n} Q$.

— **Base case** $Q = [\![P]\!]$. It is enough to take $P' = P$. Clearly $P \rightarrow_{cJ}^{*} P' = P$. Since $\vdash P : \Box_1$, then $Q = [\![P]\!]$ has no coordinators (i.e., $n = 0$ and $f = 0$).
— **Inductive step** $[\![P]\!] \rightarrow_J^{k} Q'' \rightarrow_J Q$. By inductive hypothesis on $[\![P]\!] \rightarrow_J^{k} Q''$

1 $Q'' \equiv;\mathbf{def}\ [\![D'']\!]\ \mathbf{in}\ [\![M_1'']\!] \mid \Pi_{i\in 1..u} R_i'' \mid \Pi_{k\in 1..F} T_k'' \mid \mathbf{def}\ D_g''\ \mathbf{in}\ 0$, where R_i'' are unfinished Join negotiations, T_k'' are finished negotiations and $norm(\Pi_{k\in 1..f}\ T_k'') \equiv [\![M_2'']\!]|\mathbf{def}\ D_c''\ \mathbf{in}\ 0$.

2 $P \rightarrow^*_{cJ} P'' \equiv \mathbf{def}\ D''\ \mathbf{in}\ M_1'' \mid M_2'' \mid \Pi_{i\in 1..u}\mathcal{N}_i''$ where $\mathcal{N}_i''$ is the standard cJoin transaction corresponding to R_i''.

Then the proof proceeds by case analysis of applied rule for reducing $Q'' \rightarrow_J Q$.

– for some $h \in 1..u''$, $R_h'' \rightarrow_J R_h'$. There are two different cases:

1 The applied rule corresponds to the commit protocol. Since the protocol is confluent, then $norm(R_h') = norm(R_h'')$. Then, it is enough to take $P' = P''$, which satisfies all conditions.

2 The applied rule is not part of the commit protocol. Consequently, the applied rule is the encoding of some rule in P, and has the following shape $[\![x\langle\vec{u}\rangle \triangleright P_3]\!]$ or $[\![x\langle\vec{u}\rangle | x_1\langle\vec{u}_1\rangle \triangleright P_3]\!]$. We consider here the last case, which is the most interesting one. Hence, there exists a definition

$$x\langle c_1, a_1, j_1\rangle \mid y\langle c_2, a_2, j_2\rangle \triangleright [\![P]\!]_{c_1,a_1,j_1} \mid j_1\langle c_2, a_2\rangle \mid j_2\langle c_1, a_1\rangle \mid c_2\langle\rangle$$

Moreover, R_h'' contains the messages for activating the rule. The application of the rule removes the consumed messages and activates the guarded process. The application of the rule will cause the two coordinators j_1 and j_2 to be joined to the same transaction. The effect of normalisation will depend on the structure of P_3

(a) $P_3 = y\langle\vec{v}\rangle$, s.t. y is a message to a local port, then R_h' contains $y\langle c_1, a_1, j_1\rangle$ and the obtained transaction is unfinished. Clearly, this reduction corresponds to a reduction that merges two cJoin transactions.

(b) If P_3 consists of a message to a merge port, then the proof is analogous to the previous case.

(c) $[\![P_3]\!]c_1, a_1, j_1$ produces a commit vote, there are two cases: (i) if the vote is the last one, then, by normalising, R_i commits. It is easy to notice that this case corresponds to the case in which all local names have been consumed, then there exist P' s.t. $P'' \rightarrow_{cJ} P'$ by using commit; (ii) if some coordinators still wait the vote, then it is enough to take $P' = P''$.

(d) $P_3 = abort$, then the encoding $[\![P_3]\!]c_1, a_1, j_1$ produces a commit vote to on the port a_1. The normalisation makes all coordinators in R_h'' to abort and to release the compensations. It is easy to notice that this corresponds to $P'' \rightarrow^*_{cJ} P'$ by producing first the abort in the negotiation h and then applying rule (ABORT), which releases all compensations.

3 If the reduction is $\Pi_{k\in 1..f} T_k'' \rightarrow_J \mathcal{R}$. Since all T_k'' are finished negotiations and that normalisation procedure is confluent, then $norm(\Pi_{k\in 1..f} T_k'') = norm(\mathcal{R})$. Therefore, it is enough to take $P' = P''$.

4 The applied rule is a definition in $[\![D'']\!]$:

(a) The applied rule is part of the encoding of an ordinary definition:

- messages are in $[\![M_1'']\!]$, then immediate by reducing P'' by consuming messages in M_1''.
- if a message is in some T_k''. This is possible only if some coordinator has

finished and released the continuation or the compensation, by correctness of the commit protocol, the message is in $[\![M_2'']\!]$, hence it is possible to fire the corresponding rule in P''.

Note that messages cannot be part of some R_h because those transactions have not reached a decision, so global messages are kept by coordinators.

(b) the applied rule is part of the encoding of a merge definition. This case is similar to the reduction internal to a negotiation and follows by analysing the pattern of the applied rule.

Finally, condition $\forall x : norm(Q) \downarrow_x \Rightarrow P' \downarrow_x$ immediately follows from conditions (2) and (3).

□

Appendix B. Formal definition of Coor

Before giving the full Join code for coordinators, we describe intuitively their behaviour with the transition state diagram in Figure 16. The initial state is called *state*. While in the initial state, a coordinator may accept requests for being joined (event *join*) with another participants. Any request is confirmed either with *okjoin* or *nojoin*. In both cases the coordinator returns to the initial state. In the initial state the coordinator can also receive the message to start the execution of the protocol, either with *cmt* (i.e., commit) or *abt* (i.e., abort). After receiving *cmt* the coordinator goes to the state *commit*. While in state *commit*, a coordinator behaves like in the original protocol, i.e. by notifying all known parties and by receiving commit confirmation until all parties commit. In such case, the coordinator reaches the state *finished*. Instead, if the coordinator receives the message *abt* when being in *state* or *commit*, it goes to state *abort*. While in *abort*, coordinators notify all known parties and discover the whole set of participants (analogously to commit). When all abort confirmations are received, the coordinator reaches the final state *finished*.

The Join code defining coordinators Coor is presented in Figure 17. Rule (0) fixes the initial state of the coordinator and is the only initially enabled rule of our encoding. This rule consumes the message $cmp\langle x\rangle$ and sets x as the compensation to be activated on abort. The current state of the coordinator is represented with the message $state\langle\alpha, \beta\rangle$, where α is the compensation to be released on abort and β is the list containing the channels corresponding to the coordinators of other parties in the same transaction (note that β is initially empty). The following three rules (i.e., (1)-(3)) handle the joining of new parties in the transaction. When the coordinator is in the state $state\langle\alpha, \beta\rangle$ and receives a request $join\langle t, f\rangle$ for updating the state, it may accept the request (rule (1)) by passing to the state *waitjoin* and sends on t the private ports on which it expects the update confirmation (i.e., message *okjoin*) or the cancellation (i.e., message *nojoin*). Rule (2) handles the reception of a join confirmation, which updates the set of known parties, while rule (3) deals with the cancellation. In both cases the coordinator transits to the initial state (possibly updating it).

Remark B.1. For simplicity, we abstract away from this two-step communication in

the presentation of Section 5 and we simply described join as a one-way message communication on port *join*.

Rule (4) starts the protocol with the commit vote, while rules (5)–(7) handle committing phase, and are analogous to the D2PC of (Bruni et al., 2002). There are two subtle differences: (i) channels *state* and *commit* have the extra parameter β, which is a list of the ports abt_i of known participants to be used only if the state *abort* is reached; and (ii) coordinators goes to state *finished* after commit (rule (7)). Nevertheless, the behaviour for committing coordinators are as in the original proposal in (Bruni et al., 2002).

The behaviour for the aborting phase is given by rules (8)–(13). Rules (8) and (9) start the aborting phase when the coordinator receives a message on channel *abt* and it is either in the initial state (rule (8)) or in the commit phase (rule (9)). In both cases the coordinator triggers a message $abort\langle\beta, \beta', \beta'', \alpha\rangle$, which carries the following values:

— β records the set of *abt* ports of known participants that must still be contacted (analogous to ℓ);
— β' stores the list of ports abt_i of known participants involved in the same transaction, which is typically augmented during the D2PC with the sets sent by other participants (analogous to ℓ');
— β'' records the parties who have already sent their consensus for abort (analogous to ℓ'');
— α store the messages to be released when aborting, i.e., the activation of the compensation.

Note that the behaviour for the aborting phase (rules (10)–(13)) is analogous to the committing phase, and it can be described as follow:

1 **first phase**. The participant sends the abort vote to every known thread in β (rule (10)). The message contains the list β' of all known participants, and the sender identification *abt*.
2 **second phase**. The participant collects the messages sent by other parties and updates its own synchronisation set (rule (11) and (12)). A request will be also sent to the new items in the synchronisation set (by repeating the first phase for them).
3 When the set of aborting parties is transitively closed, the protocol terminates locally and the coordinator transits to the state *finished* and releases the compensation α (rule (13)).

Rules (14)–(16) are for collecting garbage, and state that messages received when the protocol has finished are ignored. Moreover rules (17)-(19) state that the state of a coordinator cannot be updated when the protocol has begun.

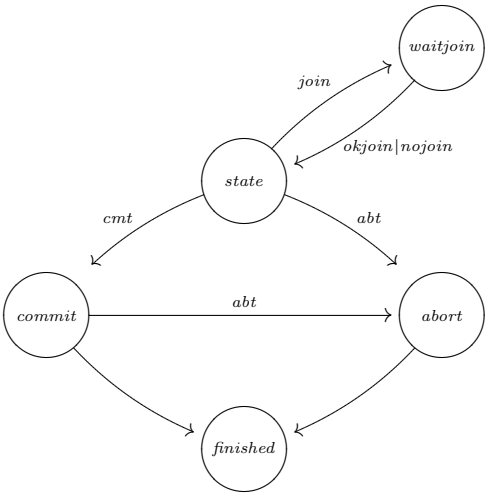

Figure 16. States of coordinators

$$
\begin{array}{lll}
(0) & \mathsf{Coor} \equiv & cmt\langle x\rangle \triangleright state\langle\{x\},\emptyset\rangle \\
(1) & \wedge & state\langle\alpha,\beta\rangle \mid join\langle t,f\rangle \triangleright t\langle okjoin, nojoin\rangle \mid waitjoin\langle\alpha,\beta\rangle \\
(2) & \wedge & waitjoin\langle\alpha,\beta\rangle \mid okjoin\langle\beta'\rangle \triangleright state\langle\alpha,\beta\cup\beta'\rangle \\
(3) & \wedge & waitjoin\langle\alpha,\beta\rangle \mid nojoin\langle\rangle \triangleright state\langle\alpha,\beta\rangle \\
 & & \\
(4) & \wedge & state\langle\alpha,\beta\rangle \mid cmt\langle\ell,\kappa\rangle \triangleright commit\langle\ell\setminus\{lock\},\ell,\{lock\},\alpha,\kappa,\beta\rangle \\
(5) & \wedge & commit\langle\{l\}\cup\ell,\ell',\ell'',\alpha,\kappa,\beta\rangle \triangleright commit\langle\ell,\ell',\ell'',\alpha,\kappa,\beta\rangle \mid l\langle\ell', lock, abt\rangle \\
(6) & \wedge & commit\langle\ell,\ell',\ell'',\alpha,\kappa,\beta\rangle \mid lock\langle\ell''',l,a\rangle \triangleright \\
 & & \qquad commit\langle\ell\cup(\ell'''\setminus\ell'),\ell'\cup\ell''',\ell''\cup\{l\},\alpha,\kappa,\beta\cup\{a\}\rangle \\
(7) & \wedge & commit\langle\emptyset,\ell,\ell,\alpha,\kappa,\beta\rangle \triangleright \mathsf{release}\langle\kappa\rangle \mid finished\langle\rangle \\
 & & \\
(8) & \wedge & state\langle\alpha,\beta\rangle \mid abt\langle\beta',a\rangle \triangleright abort\langle(\beta\cup\beta')\backslash\{abt\},\beta\cup\beta',\{abt,a\},\alpha\rangle \\
(9) & \wedge & commit\langle\emptyset,\ell',\ell'',\alpha,\kappa,\beta\rangle \mid abt\langle\beta',a\rangle \triangleright \\
 & & \qquad abort\langle(\beta\cup\beta')\backslash\{abt\},\beta\cup\beta',\{abt,a\},\alpha\rangle \mid a\langle\beta,abt\rangle \\
(10) & \wedge & abort\langle\{a\}\cup\beta,\beta',\beta'',\alpha\rangle \triangleright abort\langle\beta,\beta',\beta'',\alpha\rangle \mid a\langle\beta',abt\rangle \\
(11) & \wedge & abort\langle\beta,\beta',\beta'',\alpha\rangle \mid lock\langle\ell''',l,a\rangle \triangleright abort\langle\beta\cup(\{a\}\backslash\beta'),\beta'\cup\{a\},\beta'',\alpha\rangle \\
(12) & \wedge & abort\langle\beta,\beta',\beta'',\alpha\rangle \mid abt\langle\beta',a\rangle \triangleright \\
 & & \qquad abort\langle\beta\cup(\beta'''\backslash(\beta''\cup\{a\})),\beta\cup\beta''',\beta''\cup\{a\},\alpha\rangle \\
(13) & \wedge & abort\langle\emptyset,\beta,\beta,\alpha\rangle \triangleright \alpha\langle\rangle \mid finished\langle\rangle \\
 & & \\
(14) & \wedge & finished\langle\rangle \mid cmt\langle\ell,cnt\rangle \triangleright finished\langle\rangle \\
(15) & \wedge & finished\langle\rangle \mid lock\langle\ell,l,a\rangle \triangleright finished\langle\rangle \\
(16) & \wedge & finished\langle\rangle \mid abt\langle\beta,a\rangle \triangleright finished\langle\rangle \\
 & & \\
(17) & \wedge & finished\langle\rangle \mid join\langle t,f\rangle \triangleright f\langle\rangle \mid finished\langle\rangle \\
(18) & \wedge & commit\langle\ell,\ell',\ell'',\alpha,\kappa,\beta\rangle \mid join\langle t,f\rangle \triangleright f\langle\rangle \mid commit\langle\ell,\ell',\ell'',\alpha,\kappa,\beta\rangle \\
(19) & \wedge & abort\langle\beta,\beta',\beta'',\alpha\rangle \mid join\langle t,f\rangle \triangleright f\langle\rangle \mid abort\langle\beta,\beta',\beta'',\alpha\rangle
\end{array}
$$

Figure 17. Join code of coordinators.