Nested commits for mobile calculi: extending Join *

Roberto Bruni
Dipartimento di Informatica
Università di Pisa, Italia
bruni@di.unipi.it

Hernán Melgratti
Dipartimento di Informatica
Università di Pisa, Italia
melgratt@di.unipi.it

Ugo Montanari
Dipartimento di Informatica
Università di Pisa, Italia
ugo@di.unipi.it

ABSTRACT

In global computing applications, where the communication is asynchronous, the availability of a mechanism for some form of committed choice can be useful, and sometimes necessary. It can handle, e.g., contract stipulation, distributed agreements, negotiations with nested choice points to be carried out concurrently, causally dependent decisions. We propose a linguistic extension of the Join calculus for programming nested commits, with the possibility of abort and compensation and then we discuss its expressiveness on the basis of a few examples and of the comparison with other paradigms with similar aims but designed in different contexts.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent programming—*Distributed programming*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; F.1.1 [**Computation by abstract devices**]: Models of Computation—*Relations between models*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms

Languages, Theory

Keywords

Distributed negotiations, nested commits, join-calculus, AKL, Petri nets, Serializability

*Research supported by the MSR Cambridge Project Napi, by the FET-GC Project IST-2001-32747 Agile, by the MIUR Project COFIN 2001013518 CoMeta, and by the MURST-CNR 1999 Project, *Software Architectures on Cooperative WAN.*

1. INTRODUCTION

In recent years, wide area network computing, web programming, and, more generally, *global computing* (GC) are attracting the interest of many researchers in an attempt of laying the foundations for largely distributed applications, where several challenging issues like dynamic interconnectivity, heterogeneity, interoperability, mobility, security, and efficiency must be dealt with. Since synchronous communication in such a distributed world is unrealistic, all GC models, systems and languages proposed in the literature agree that the communication must be asynchronous. Nevertheless, the "orchestration" of data and of mobile processes is a key issue, as many commercial applications actually running on different platforms often require a coordination layer between components that are designed and implemented separately (e.g. in e-commerce or on-line auction systems).

Though orchestration can be interpreted as the atomic execution of certain control activities (e.g. in terms of transactions, like in [16, 4]), more generally, the problem is that of committing the results of long distributed decision processes as soon as the participants reach partial agreements (also called *contracts* or *negotiations*). Due to the highly distributed network topology, in this context the terminology "commit" is maybe more appropriate than "transaction" because the ordinary ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions cannot be guaranteed as such, and for certain applications they might not be desirable at all. For example, regarding atomicity, the fact that local commits are executed in all sites at the same time can hardly be ensured in a fully distributed system. Microsoft's BizTalk and Sun Microsystem's Javaspaces exploit a centralized *transaction manager* (TM) to guarantee, e.g., that if a negotiation aborts then a consistent configuration is restored in all participating sites. Though TMs hide the coordination layer to programmers and users, the design of TMs is often customized to each different system, possibly statically fixing the number and locations of participants. Thus TMs are maybe not the optimal option for such an heterogeneous and dynamic system like WANs. Apart from technical problems like the heavy task overload on TM servers, their presence can pose also delicate managing questions, like deciding at the very beginning of a transaction which TM should take control when two or more participant organizations are using different proprietary software systems.

Process description languages (PDLs), e.g. the π-calculus, are mathematical models of computation designed for isolating and studying phenomena that can occur in a concurrent language, playing a similar role as the λ-calculus has played for sequential computations. In the spirit of PDL, it would be desirable to have a linguistic extension of some well-known calculus with primitives for distributed commits. Moreover, the extension should have a low overhead and be apt to programming open systems, where applications must be designed and written separately, being able to interact with different (possibly partially specified) environments that can also change dynamically. Since, in general, negotiations cannot deterministically succeed and contracts may involve choices between available options, two key operations are the "abort with compensation" (to stop a negotiation as soon as a failure occurs or some participants withdraw their interest in carrying out the contract) and the "commit" (to store the result of a partial agreement before moving to the next phase of a long negotiation). Upon a failure, new attempts for finding an agreement between all participants can be initiated. When several agreements are necessary, commits can be used to fix them separately and in an efficient way. For example, in the "trip booking" example presented in Section 4.1, it is obvious that the destination must be chosen before trying to book an hotel room, and the selection of a particular hotel category usually depends on traveler's budget.

In this paper we propose such a linguistic extension for the Join calculus, a well-known PDL which has a running distributed implementation. In particular, we give the syntax and operational semantics of our extension called *Committed Join*, then illustrate its usage by means of a few examples and compare its *expressiveness* against other paradigms with similar aims (but in different contexts) proposed in the literature. Note that the term expressiveness is here used informally to denote the naturality in modeling certain features, and not the computational power.

To show that the extension we have in mind is not ad hoc for Join, let us conclude the Introduction by stating some intuitive requirements that any PDL with commit should try to meet. We state them borrowing some terminology from the *chemical abstract machine* (CHAM) [2], a well-known abstract model of concurrent (and distributed) computations. We will discuss in §3 how faithfully committed Join can fit these requirements.

According to the CHAM, basic data and processes are seen as molecules m living in *chemical solutions* s, possibly grouped by membranes $s = \{\![m_1, \ldots, m_n, s_1, \ldots, s_k]\!\}$ to form a generic chemical soup. Note that membranes can be nested, i.e. solutions can be hierarchical $\{\![\{\![\{\![\ldots]\!\} \ldots \{\![\ldots \{\![\ldots]\!\} \ldots]\!\}]\!\}$. In this sense, membranes can roughly be seen as anonymous *ambients* [7]. Suitable heating / cooling rules can be specified to rearrange the structure of membranes (like structural axioms in PDL). A set of *chemical reaction* rules defines the basic actions that can be performed, specifying the operational semantics of the system. The generic state of a CHAM can be seen as a tree of solutions that reflects the nesting of membranes. Each node can have local resources and multiprocessing, and what is local for a node is global to its children.

Contracts are thus decision processes distributed on several nodes, each with the possibility of consulting both local and global resources and generating local sub-contracts (e.g. modeling decisions internal to an organization). However: (i) each internal sub-decision should not be made public before a common agreement is achieved, therefore only local resources can be produced to store such information; (ii) global resources might be made available to partners upon commits, marking the conclusion of some contract; (iii) decision processes can be aborted, in which case all participants should be informed and suitable compensation procedures should be activated, possibly unlocking all the resources seized during the negotiation process and returning them to the original state; (iv) though divergence is possible in principle, well designed GC applications should guarantee that each contract eventually leads to an abort or a commit (e.g., timeouts could be used for this purpose); (v) when two processes involved in separate negotiations exchange some information, then their contracts should be merged into a unique one. Similarly, there should be mechanisms for allowing new participants to dynamically join an ongoing contract, taking part in the final decision. If a participant s already involved in some negotiation joins a new negotiation, then the two distinct negotiations must be merged, as he/she could freely establish causal dependencies among his/her activities in the two negotiations. A participant interested in carrying out two separate contracts should instead be able to emit two independent agents, carrying out the two negotiations autonomously in a concurrent way.

Note that, from what said above, (vi) it should be possible to have nested contracts. In general, an internal abort can be compensated in such a way that the main contract can be successfully completed. On the other hand, a failure of the main contract should cause the abort of all ongoing internal activities.

On transactions. As already mentioned, we prefer to talk about negotiations than transactions, because although the two notions are similar, the second terminology usually refers to different contexts.

The term transaction was coined in database community to designate sequences of operations or actions which preserve database consistency [8]. Traditional transactions must satisfy the well-known ACID properties and any concurrent execution of several transactions is considered correct if it is *serializable*: the same output can be produced by executing transactions one at a time, i.e. without interleaving steps from different transactions [1]. A variety of models were proposed in order to achieve ACID properties while maximizing system throughput (see [11]). Nevertheless, it has been widely recognized that ACID transactions are suitable to deal with small units of work that access only a few data items, but they are not appropriate to handle long-lived activities or cooperating atomic activities. Consequently, specific models (such as *Sagas* [10] and *Split and Join* [14]) has been studied. They usually adhere to some relaxed form of ACID properties, allowing for instance the sharing of temporary results between different transactions or relying on compensations to undo changes committed in advance.

Recent transaction models arose in the context of *workflow management systems* (WMS), which support the design and execution of business processes. Unlike database transactions, WMS transactions have to coordinate the execution of several independent applications and not just different database management systems. In particular, transactions are usually specified in term of dependency between tasks, such as the joint commitment of a set of activities, cascading aborts, and so on. For a detailed discussion about transactional workflows see [17].

There are also commercial WMS, such as Microsoft® *Biztalk* [16], that supports transactional aspects of workflows. This system provides a graphical environment for modeling processes as flowcharts, where several *shapes* can be composed and *boxed* into transactions. Each shape represents a basic action for sending/receiving data or redirecting the execution flow. BizTalk boxes correspond to a two-level classification of states, *observable* (i.e., outside the box) and *hidden* (i.e., inside the box), a partition that also pervades another formalism, which has been recently defined to model transactional aspects in concurrent systems, the *zero-safe Petri nets* (ZS nets) [6, 5]. BizTalk flowcharts may be easily encoded into ZS nets, which additionally provides a "dynamic" specification of transactions boundaries (as opposed to the "static" one of BizTalk) supporting multiway transactions, which retain several entry and exit points, and admit a number of participants which is statically unknown. Noticeably, it has been shown that the transactional mechanism of ZS nets can be implemented in Join, without TMS, by exploiting a novel distributed commit protocol [4].

Synopsis. In § 2 we recall the principles of the CHAM and the syntax and semantics of Join. In § 3 we introduce the linguistic extension of Join with primitives for distributed negotiations, called Committed Join. We show that it is a conservative extension of Join and compare it against the informal requirements stated in this Introduction. In § 4 we illustrate the main features of Committed Join by means of two simple applications. In § 5 we show a natural encoding of ZS nets into Committed Join. Noticeably, whereas in [4] suitable controllers were needed to implement the commit protocol, zero-safe nets can now be straightforwardly encoded, taking advantage of the commit primitives. In § 6 we characterize a set of committed Join processes that are serializable. Such process are defined by a syntactical restriction and are called "*shallow*". Moreover, serializability is expressed as a big step relation between processes. Finally, we show that shallow processes are serializable by proving a correspondence between the two semantics.

2. BACKGROUND

2.1 The Chemical Abstract Machine

The CHAM was proposed in [2] as a suitable framework to model concurrent computations. Basically, states (called *solutions*) are finite multisets of *molecules* (i.e. terms of some algebra), and computations are multiset transformations. Multisets are written as $m_1, \dots, m_n$. Consequently, $\_ , \_$ is associative and commutative. We usually abbreviate $m_i, \dots, m_n$ with $\otimes_i\, m_i$. Solutions can be structured in a hierarchical way by using the operator *membrane* $\{\![.]\!\}$, grouping a solution into a molecule. A special form of molecules can be built by using *airlock* constructor, which is helpful for modeling calculi that include action prefix and synchronization primitives, but we do not use it in our presentation.

Transformations are described by specific *chemical rules*, which specify how portions of a solution may react. Usually in a CHAM there are two different kinds of chemical rules: *Heating / cooling* (or *structural*) rules $\rightleftharpoons$ representing syntactical rearrangements of molecules in a solution, and *reaction* rules $\rightarrow$ stating basic computation steps. Note that structural rules are reversible: a solution obtained by applying a cooling rule can be heated back to the original state, and vice versa. Generally, those rules closely correspond to the usual notion of structural congruence between terms. On the other hand, reaction rules cannot be undone.

The laws that rule CHAM computations state that whenever an instance of a left-hand-side of a rule is found into a solution, it can be replaced by the corresponding instance of the right-hand-side. Chemical rules $S_1 \rightarrow S_2$ have no premises and are purely local, specifying only the part of the solution that actually changes. Consequently, they can be applied freely in every solution $S, S_1 \rightarrow S, S_2$ (*chemical law*) and also in subsolutions, e.g. $\{\![S, S_1]\!\} \rightarrow \{\![S, S_2]\!\}$. In particular, they can be nested at any level of the hierarchical structure of the chemical solution. Note that, since solutions are multisets, rules can be applied concurrently (associative and commutative parallel composition of molecules can be lifted to the level of reactions, like in Petri nets).

2.2 The Join calculus

The Join calculus [9] is a process description language with asynchronous name-passing communication and it has the same expressive power as the asynchronous π-calculus. Join relies on an infinite set of names $x, y, ..., u, v, ...$ Name tuples are written $\vec{u}$. Join processes, definitions and patterns are given by the following grammar:

PROCESSES	P, Q	$::=$	$0 \mid x\langle\vec{y}\rangle \mid \mathbf{def}\ D\ \mathbf{in}\ P \mid P\vert Q$
DEFINITIONS	D, E	$::=$	$J \triangleright P \mid D \wedge E$
PATTERNS	J, K	$::=$	$x\langle\vec{y}\rangle \mid J\vert K$

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 reaction rules $J \triangleright P$ that associate *join-patterns* J with *guarded processes* P. Names introduced by the definition D of **def** D **in** P are bound in the whole process P as well as in the guarded processes contained in D. The sets of defined names *dn*, received names *rn* and free names *fn* are defined in Figure 1.

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 below:

STR-NULL	0	$\rightleftharpoons$	
STR-JOIN	$P \mid Q$	$\rightleftharpoons$	P, Q
STR-AND	$D \wedge E$	$\rightleftharpoons$	D, E
STR-DEF	$\mathbf{def}\ D\ \mathbf{in}\ P$	$\rightleftharpoons$	$D\sigma_{dn(D)}, P\sigma_{dn(D)}$ (with $range(\sigma_{dn(D)})$ globally fresh)
RED	$J \triangleright P, J\sigma$	$\rightarrow$	$J \triangleright P, P\sigma$

We remind that all reactions can be applied inside any larger solution surrounded by membranes. A process P can start its computation when is put into a solution, i.e. the initial state has the form $\{\![P]\!\}$. Rule STR-NULL states that 0 can be added or removed from any solution. Rules STR-JOIN and STR-AND stand for the associativity and commutativity of $|$ and $\wedge$, because $\_ , \_$ is such. STR-DEF denotes the activation of a local definition. Note that activation implements a static scoping discipline by properly renaming defined ports by *globally fresh* names. A name x is fresh w.r.t.

$dn(x\langle\vec{y}\rangle) = \{x\}$	$dn(J\|K) = dn(J) \cup dn(K)$
$dn(J \triangleright P) = dn(J)$	$dn(D \wedge E) = dn(D) \cup dn(E)$
$rn(x\langle\vec{y}\rangle) = \{\vec{y}\}$	$rn(J\|K) = rn(J) \cup rn(K)$
$fn(D \wedge E) = fn(D) \cup fn(E)$	$fn(J \triangleright P) = dn(J) \cup (fn(P) \backslash rn(J))$
$fn(0) = \emptyset$	$fn(x\langle\vec{y}\rangle) = \{x\} \cup \{\vec{y}\}$
$fn(P\|Q) = fn(P) \cup fn(Q)$	$fn(\mathbf{def}\ D\ \mathbf{in}\ P) = (fn(P) \cup fn(D)) \backslash dn(D)$

Figure 1: Defined, received, and free names.

a process P (respectively a definition D) if $x \notin fn(P)$ (resp. $x \notin fn(D)$). Moreover, x (respectively a set of names) is fresh w.r.t. a solution if it (resp. every name) is fresh w.r.t. every term in the solution. We write the substitution of names $x_1 \ldots x_n$ by names $y_1 \ldots y_n$ as $\sigma = \{^{y_1 \cdots y_n}/_{x_1 \ldots x_n}\}$, with $dom(\sigma) = \{x_1, \ldots, x_n\}$ and $range(\sigma) = \{y_1, \ldots, y_n\}$. We indicate with σ_N an injective substitution such that $dom(\sigma) = N$. We require names to be globally fresh, which means fresh w.r.t the implicit context in which rule is applied. Note that the version considering context would be

$$\otimes_i\ m_i, \mathbf{def}\ D\ \mathbf{in}\ P \ \rightleftharpoons\ \otimes_i\ m_i, D\sigma_{dn(D)}, P\sigma_{dn(D)}$$

with side condition $range(\sigma_{dn(D)}) \cap fn(\otimes_i\ m_i) = \emptyset$.

Consider, for instance, the solution $\{\![z\langle x, z\rangle, \mathbf{def}\ x\langle y\rangle \triangleright z\langle y, x\rangle\ \mathbf{in}\ x\langle a\rangle]\!\}$, whose second molecule contains a local definition of a port named x, which is different from the homonym port in the first molecule. When rule STR-DEF is applied, the local definition of x is renamed by using a fresh name, obtaining, for instance, the solution $\{\![z\langle x, z\rangle, x_1\langle y\rangle \triangleright z\langle y, x_1\rangle, x_1\langle a\rangle]\!\}$.

Finally, RED describes the use of an active reaction rule $(J \triangleright P)$ to consume messages corresponding to an instance of J (for a suitable substitution σ, with $dom(\sigma) = rn(J)$), and produce a new instance of its guarded process $P\sigma$. By applying RED for $\sigma = \{^a/_y\}$, the previous solution reduces to $\{\![z\langle x, z\rangle, x_1\langle y\rangle \triangleright z\langle y, x_1\rangle, z\langle a, x_1\rangle]\!\}$. Note that the local port x_1 has been extruded because it is in the content of a message on the free channel z.

3. COMMITTED JOIN

3.1 Syntax

In order to represent committed choices, we extend the syntax of the join calculus by adding terms to denote negotiation processes. They are represented by $[P : Q]$, where P is the normal execution of the activity and Q is its compensation, i.e., the process that should be executed in the case of an abort. A negotiation process is executed in isolation until reaching either a commit state or an abort condition. In the case of commit, the result is delivered to the outside, whereas its compensation is given control in the case of abort. The abort decision is caused by the presence of the special basic process *abort*. A process can define the creation of new negotiations by using reaction rules as $J \triangleright [P : Q]$, i.e., a rule that produces a new instance of the negotiation process P with compensation Q when it is fired. To allow interactions among ongoing negotiations we introduce a new kind of reaction rules, called *merge definitions*. These definitions have the form $J \blacktriangleright P$ and allow the consumption of messages produced in the scope of different contracts by joining all participants to such negotiations in a unique larger one.

The following grammar defines processes and definitions. In order to further simplify the description, we syntactically distinguish processes without definitions, i.e., a parallel composition of messages (M), from processes that can contain definitions (P).

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

The definition of *rn* is the same as before. For *fn* we add: $fn(abort) = \emptyset$ and $fn([P : Q]) = fn(P) \cup fn(Q)$. Additionally, we assume that names introduced by ordinary patterns in a definition D are different from names defined by the merge patterns of D. Consequently, we redefine *dn* (for definitions) from the scratch using two functions: the *defined ordinary* names dn_{o} and the *defined merge* names dn_{m} (see Figure 2). We impose that $dn_{\mathrm{o}}(D) \cap dn_{\mathrm{m}}(D) = \emptyset$ holds for every definition D in a process.

3.2 Operational Semantics

The operational semantics of Committed Join is defined in the reflexive CHAM style. Molecules m and solutions S are given by the following grammar:

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

As in ordinary Join, processes and definitions are molecules. Additionally, we consider molecules of the form $\llcorner P \lrcorner$ denoting compensations that are frozen inside a solution. The chemical rules are in Figure 3.

The first five chemical rules are the ordinary ones for Join. Rule STR-CONT describes how a term denoting a contract corresponds to a sub-solution consisting of two molecules: the process P and its compensation Q, which is frozen (because the operator $\llcorner . \lrcorner$ forbids the enclosed process to compute).

COMMIT describes the effect of a committed choice. A commit can be executed only when all internal computations have finished, i.e., all participants have done their tasks reaching a (local) state that does not contain locally defined names, which represent internal states of a negotiation. In this way, a commit means clean termination, where all names denoting internal states of contracts have been consumed. Note that, at commit time, all definitions belonging to a contract can be discarded, because the messages that are being released do not contain those names, and those names cannot be extruded. Additionally, the compensation procedure is discarded at commit time. Note that a negotiation cannot commit when *abort* is within the solution. In fact, a process P cannot reduce to $M \mid \mathbf{def}\ D\ \mathbf{in}\ 0$ when $P \equiv abort \mid P'$ as abort is not a message. On the other hand, abort condition is handled with the rule ABORT. It states that when *abort* is part of a solution the compensation process can be released. Compensations are not predefined to be executed atomically. Nevertheless, as they are

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

Figure 2: Ordinary and merge names.

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

Figure 3: Operational semantics of Committed Join.

ordinary processes, they can be defined by the programmer as negotiations themselves.

Interactions among negotiations are specified by rule MERGE, which consumes messages from different contracts and create a new larger negotiation by combining the definitions and messages of the original ones and a new instance of the guarded process $P\sigma$, where $dom(\sigma) = rn(J)$. Special attention should be put at merging negotiations in order not to capture names among local definitions in different negotiations. In particular, name clashes are avoided when the following side condition is considered $dn(S_i) \cap (dn(S_j) \cup fn(S_j)) = \emptyset$ for $i \neq j$. However, if we are guaranteed that the structural rule STR-DEF generates globally fresh names (and not just locally fresh names) then the side condition of MERGE can be safely omitted, as it is always trivially satisfied. Additionally, the compensation corresponding to the merged negotiation is defined as the parallel composition of all the original compensations.

In the case of nested contracts only negotiations at the same level can be joined. In practice, it might be useful to have the possibility of applying a top merge definition at an arbitrary depth in the hierarchy of the chemical solution. To this purpose, the rule STR-MOVE below might be added, so that merge patterns could float across contract boundaries, and sub-contracts could be merged using a global merging rule.

$$\text{STR-MOVE} \quad J \blacktriangleright P, \{[S]\} \quad \rightleftharpoons \quad \{[J \blacktriangleright P, S]\}$$

It should be noted that previous reduction rules do not consider stall situations. Consider for instance the following process $[\mathbf{def}\ x\langle\rangle | y\langle\rangle \triangleright 0\ \mathbf{in}\ x\langle\rangle : Q]$, clearly the negotiation cannot compute, because there are no active reaction rules to apply. Neither it can commit, because there is a message ($x\langle\rangle$) on a local port. In this situation the contract is blocked and should be aborted. Some of these situations can be recognized and handled locally to promote the abort (i.e., when no local rules can be applied). These situations can be represented by ad hoc rules or by a general rule to generate nondeterministically the abort (situations that real implementations typically handle with timeouts). Nevertheless, we cannot expect to axiomatize stall situations because it would mean to write axioms recognizing a non-termination condition, which is an undecidable problem. On the other hand, we do not want to limit the expressiveness of the language.

PROPOSITION 1. *Committed Join is a conservative extension of Join.*

PROOF. That any Join process is also a Committed Join process is obvious from the syntax. It remains to show that: (1) for any Join processes P and Q, if $P \rightarrow Q$ in Join, then $P \rightarrow Q$ in Committed Join; and (2) for any Join processes P, if $P \rightarrow Q$ in Committed Join, then Q is a Join process and $P \rightarrow Q$ in Join. Both implications follow straightforwardly from the fact that the chemical rules of Committed Join 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. □

With respect to the informal requirements discussed in the Introduction, we have that, as expected, membranes are exploited to define the boundaries of negotiations. Process like $P|[P_1|[P_2 : Q_2]|[P_3 : Q_3] : Q_1]$ straightforwardly model sub-negotiations. The decisions taken internally by P_2 can influence P_3 only if some merging definition is available at the level of P_1, in which case the sub-negotiations would be joined together. In absence of merge definitions, global and local resources are kept separate in each sub-negotiation. The commit of P_2 can only happen when the internal state contains only global resources (and no local ones). At commit time, the result of the negotiation is made available to P_1. An abort generated in P_2 activates the compensation at Q_2 at the level of P_1, neither forcing the abort of any other sub-negotiations (P_3) nor the abort of the main contract (P_1). Note that if $[P_2 : Q_2]$ was the result of the merging of several negotiations, then the compensation Q_2 is the union of all the compensations of the participants. Since Join is Turing complete, nontermination is undecidable, and it is left to the programmer's skill to design applications that eventually lead to an abort or a commit.

An important restriction is that local resources cannot cross negotiations boundaries and made known to siblings sub-negotiations. The only way to exchange information between siblings sub-negotiations is via merging definition, that however merge all participants into a unique negotiation that must then commit or abort (or diverge) as such.

4. EXAMPLES

4.1 Trip booking

The goal of the first example is to show the encoding of a common application in Committed Join. It shows how three components can interact in order to allow a customer to book flights and accommodations. We model the hotel component as follows:

$$
\begin{array}{rl}
\mathrm{H} \equiv & \mathbf{def}\; WaitBooking\langle\rangle \triangleright \\
& \quad [\mathbf{def} \quad request\langle o\rangle \triangleright o\langle\$\rangle \mid price\langle\$\rangle \\
& \quad\quad \wedge\; price\langle\$\rangle \mid confirm\langle v\rangle \triangleright BookedRoom\langle v\rangle \\
& \quad\quad \wedge \quad price\langle\$\rangle \triangleright abort \\
& \quad\;\; \mathbf{in}\; offerRoom\langle request, confirm\rangle : Q] \\
& \wedge\; BookedRoom\langle v\rangle \triangleright \ldots \\
& \mathbf{in}\; WaitBooking\langle\rangle \mid \ldots
\end{array}
$$

The definition of H includes a rule that creates a negotiation to attend a customer request. This negotiation begins by publishing in the merging port *offerRoom* the names of the services a client should use to make a reservation (i.e., *request* to ask for a quote and *confirm* to accept an offer). For simplicity, we use *BookedRoom* to denote the change in the state of the hotel component when a successful reservation ends. The Airline component (A) can be written similarly but using the merging name *offerFlight* to publish services instead of *offerRoom*. The third component (Customer C) can be written as below.

$$
\begin{array}{rl}
\mathrm{C} \equiv & \mathbf{def}\; BookHotel\langle\rangle \triangleright \\
& \quad [\mathbf{def}\; HotelMsg\langle r, c\rangle \triangleright \\
& \quad\quad \mathbf{def}\; offer\langle\$\rangle \triangleright c\langle visa\rangle \mid hotelOK\langle hConf\rangle \\
& \quad\quad\quad \wedge\; offer\langle\$\rangle \triangleright abort \\
& \quad\quad\quad \wedge\; hConf\langle\rangle \triangleright HotelFound\langle\rangle \\
& \quad\quad\;\; \mathbf{in}\; r\langle offer\rangle \\
& \quad\;\; \mathbf{in}\; searchRoom\langle HotelMsg\rangle : Q'] \\
& \wedge\; BookFlight\langle\rangle \triangleright \\
& \quad [\mathbf{def}\; FlightMsg\langle r, c\rangle \triangleright \\
& \quad\quad \mathbf{def}\; offer\langle\$\rangle \triangleright c\langle visa\rangle \mid flightOK\langle fConf\rangle \\
& \quad\quad\quad \wedge\; offer\langle\$\rangle \triangleright abort \\
& \quad\quad\quad \wedge\; fConf\langle\rangle \triangleright FlightFound\langle\rangle \\
& \quad\quad\;\; \mathbf{in}\; r\langle offer\rangle \\
& \quad\;\; \mathbf{in}\; searchFlight\langle FlightMsg\rangle : Q''] \\
& \wedge\; flightOK\langle fc\rangle \mid hotelOK\langle rc\rangle \blacktriangleright fc\langle\rangle \mid rc\langle\rangle \\
& \mathbf{in}\; BookHotel\langle\rangle \mid BookFlight\langle\rangle \mid \ldots
\end{array}
$$

This component defines two different rules for creating negotiations, one for booking rooms and the other for buying flight tickets. Both processes are quite similar. In particular, the negotiation for booking a room starts by sending a message to the merging port *searchRoom* in order to get the appropriate names for interacting with a hotel. These names will be received as a message on the local port *HotelMsg*. When such message arrives, a request for a quote is sent to the port r, which has been defined and published by the component H to attend client requests. After that, a particular offer from the hotel will be received as a message on *offer*. Whether the customer accepts or not a particular quote is modeled by the multiple reaction rules for the pattern $offer\langle\$\rangle$. A customer can confirm a received offer using the name c received earlier, i.e. the local port where the hotel negotiation expects a confirmation. Additionally, to be sure that also flight tickets were found before committing, a message $hotelOK\langle hConf\rangle$ is produced to wait until a flight can be successfully booked. This message will be managed by the merging rule defined in C. The contract will be blocked until a running negotiation for buying flight tickets sends a message to *flightOK*. At this time, the merging definition can be fired and both contracts merged. Eventually the negotiation will commit by releasing the messages on *HotelFound* and *FlightFound*, which denote changes in the state of the customer. The process representing the whole system (including the rules for merging requests and offers of services) can be written as below

$$
\begin{array}{l}
\mathbf{def}\; searchRoom\langle hm\rangle \mid offerRoom\langle r, c\rangle \blacktriangleright hm\langle r, c\rangle \\
\quad \wedge\; searchFlight\langle fm\rangle \mid offerFlight\langle r, c\rangle \blacktriangleright fm\langle r, c\rangle \\
\;\; \mathbf{in}\; \mathrm{H} \mid \mathrm{A} \mid \mathrm{C}
\end{array}
$$

4.2 Andorra Kernel Language

AKL [12, 13] is a concurrent logic programming language that includes the Prolog and committed choice languages. In this section we show how nested agreements and merging definitions can be used to provide some computational features of AKL. We consider guarded rules of the form $A \leftarrow G \mid B$, where A is an atom (*head*), G (*guard*) and B (*body*) are (possibly empty) sequences of atoms, and $\mid$ is the commit operator. An AKL program $\mathcal{P}$ is a list of guarded rules. An execution of a program is initiated by providing a conjunction of atoms $\mathcal{G}$ (*goal*).

A committed Join process that simulates the execution of $\mathcal{P}$ for a goal $\mathcal{G}$ is given by the following process.

$$
\begin{array}{l}
\mathbf{def}\; \mathsf{D} \wedge [\![\mathsf{definitions}(\mathcal{P})]\!] \wedge [\![\mathsf{undefinedAtoms}(\mathcal{P})]\!] \\
\;\; \mathbf{in}\; [\![\mathbf{and}(\mathcal{G})]\!]_{trueGoal, falseGoal} \mid unif\langle final, \mathbf{true}\rangle
\end{array}
$$

where D is a set of general definitions (described later in this section), $\mathsf{definitions}(\mathcal{P})$ is a partition of $\mathcal{P}$, where every class contains all rules of $\mathcal{P}$ that define a particular atom. For instance, $\mathsf{definitions}(A \leftarrow |, A \leftarrow |B, B \leftarrow |) = \{A \leftarrow |, A \leftarrow |B\}, \{B \leftarrow |\}$. Additionally, $\mathsf{undefinedAtoms}(\mathcal{P})$ denotes the set of all atoms appearing in $\mathcal{P}$ for which there are not defining rules, i.e., atoms whose proof will fail. Constraints or unifiers are encoded conveniently as committed Join processes, and denoted by θ, and we use **true** for the empty set. We will denote the constraints associated to a particular running proof by $unif\langle final, \theta\rangle$.

Roughly, program rules are encoded as definitions that consume a message denoting an atom to be proved and producing new processes denoting the parallel proofs of the guarded goals of the defining rules. When the whole proof completes successfully, both messages $trueGoal\langle\rangle$ and $final\langle\theta\rangle$ signal the completion of the proof of $\mathcal{G}$ for the computed constraints θ. Otherwise, message $falseGoal\langle\rangle$ is sent to notify that $\mathcal{G}$ cannot be proved. Encoding rules for program clauses and goals are given in Figure 4.

The definition of an atom A given by program rules $A \leftarrow B_1 \mid C_1, \ldots, A \leftarrow B_n \mid C_n$ is encoded as a merging rule that substitutes a token $[\![A]\!]_{true,fail}$ by the proof of its defining rules in a choice-configuration. That is, any time an atom $[\![A]\!]_{true,fail}$ is present in a proof, it can be substituted by the proof of the defining rules of A. Instead, an undefined atom is encoded as a rule that always fails.

A conjunction is encoded as a process that activates a negotiation containing the parallel composition of atoms that should be proved, i.e. the $[\![A_i]\!]_{t_i,f_i}$ that can activate the proof of individual sub-atoms in the conjunction. Additionally, the negotiation contains the constraints that have been

$$
\begin{array}{llcl}
\text{(DEFINED ATOM)} & [\![A \leftarrow B_1 \mid C_1, \ldots, A \leftarrow B_n \mid C_n]\!] & = & [\![A]\!]_{true,fail} \blacktriangleright [\![\mathbf{choice}(B_1 \mid C_1, \ldots, B_n \mid C_n)]\!]_{true,fail} \\
\text{(UNDEFINED ATOM)} & [\![A]\!] & = & [\![A]\!]_{true,fail} \blacktriangleright fail\langle\rangle \\
\text{(ATOMIC GOAL)} & [\![A]\!]_{true,fail} & = & A\langle true, fail\rangle \\
\text{(AND-CONFIG.)} & [\![\mathbf{and}(A_1, \ldots, A_n)]\!]_{true,fail} & = & \mathbf{def}\ and\langle\rangle \triangleright [\mathbf{def}\ \wedge w\langle\rangle \triangleright 0 \\
 & & & \qquad \wedge \parallel_{i\in 1..n} t_i\langle\rangle \triangleright proved\langle w\rangle \mid true\langle\rangle \\
 & & & \qquad \bigwedge_{i\in 1..n} f_i\langle\rangle \triangleright and\langle\rangle \mid failed\langle\rangle \\
 & & & \qquad \text{...Rules for handling abortion...} \\
 & & & \qquad \mathbf{in} \parallel_{i\in 1..n} [\![A_i]\!]_{t_i,f_i} \mid unif\langle w,\theta\rangle : fail\langle\rangle] \\
 & & & \ \mathbf{in}\ and\langle\rangle \\
\text{(CHOICE-CONFIG.)} & [\![\mathbf{choice}(A_1|B_1, \ldots, A_n|B_n)]\!]_{true,fail} & = & \mathbf{def} \bigwedge_{i\in 1..n} i\langle\rangle \triangleright [\![\mathbf{and}(B_i)]\!]_{true,fail} \\
 & & & \ \mathbf{in}\ [\mathbf{def}\ \parallel_{i\in 1..n} f_i\langle\rangle \mid proving\langle\rangle \triangleright abort \\
 & & & \qquad \bigwedge_{i\in 1..n} t_i\langle\rangle \mid proving\langle\rangle \triangleright i\langle\rangle \mid chosen\langle\rangle \\
 & & & \qquad \text{...Rules for handling abortion...} \\
 & & & \qquad \mathbf{in} \parallel_{i\in 1..n} [\![\mathbf{and}(A_i)]\!]_{t_i,f_i} \mid proving\langle\rangle : fail\langle\rangle]
\end{array}
$$

Figure 4: Encoding of AKL committed rules.

computed locally $unif\langle w,\theta\rangle$ corresponding to the proof of the conjunction. Note that every sub-proof initiated by $[\![A_i]\!]_{t_i,f_i}$ eventually will end by notifying its successful or failed termination by using ports t_i and f_i, respectively. Rule $w\langle\rangle \triangleright 0$ is a convenient way to define a local identifier for the negotiation and has no computational meaning. The second rule handles the case when all sub-proofs end successfully. In this case, message $true\langle\rangle$ is produced to signal the successful proof of the conjunction, although it is not released until the commit. Moreover $proved\langle w\rangle$ is generated to activate the promotion of the computed constraints (rules for promotion are defined in D and are explained below). The last rules define the behavior of the contract when one of the subatoms reports a failure, i.e., a message on a port f_i is received. All running sub-contracts should be aborted and the contract should commit by releasing the activation of another instance of the whole negotiation. Note that instead of reporting a failure, a new activation of the proof is generated because committed JOIN is not aimed at providing mechanisms for exhaustive search, and consequently we cannot assure that different possibilities for evaluating subgoals are explored in a proper manner.

A choice-configuration $\mathbf{choice}(A_1|B_1, \ldots, A_n|B_n)$ is encoded as a process that starts a contract containing the parallel proof of all guards A_i. Differently from and-configurations, the whole negotiation for a choice-configuration can end as soon as one of the parallel proofs completes successfully ($t_i\langle\rangle \mid proving\langle\rangle \triangleright i\langle\rangle \mid chosen\langle\rangle$). In this case, the produced message $i\langle\rangle$, which will be released at commit, identifies the body corresponding to the rule whose guard has been proved. Additionally, message $chosen\langle\rangle$ activates the abortion of all running sub-contract. The remaining definitions handle the abortion and the consumption of messages notifying the failed termination of aborted sub-proofs. The whole contract produces at commit time only one message $i\langle\rangle$, which will activate the corresponding body $[\![\mathbf{and}(B_i)]\!]_{true,fail}$. At this point, one rule has been chosen (i.e. committed) and the final result for the whole choice-configuration depends on the result of the execution of $[\![\mathbf{and}(B_i)]\!]_{true,fail}$, i.e. the body of the committed choice.

The rules for merging constraints are presented in Figure 5. Rule (4) describes the activation of and, a process that computes atomically $\theta_1 \wedge \theta_2$. If θ_1 and θ_2 are consistent, it will produce $unif\langle z, \theta_1 \wedge \theta_2\rangle$; if not, the computation will finish with $fail\langle\rangle$. Consider the following negotiation that is ready to promote its local constraints

$[\mathbf{def} \ldots \mathbf{in}\ unif\langle w,\theta'\rangle \mid proved\langle w\rangle \mid true\langle\rangle : fail\langle\rangle] \mid unif\langle z,\theta\rangle$ by (2) reduces to
$[\mathbf{def}\ \ldots\ \mathbf{in}\ tell\langle w,\theta'\rangle \mid true\langle\rangle : fail\langle\rangle] \mid unif\langle z,\theta\rangle$
and then, by rules (1) and (4), to
$[\mathbf{def}\ \ldots\ \mathbf{in}\ and\langle\theta',\theta,z,fail\rangle \mid true\langle\rangle : unif\langle z,\theta\rangle \mid fail\langle\rangle]$.

When θ and θ' are consistent, the last process reduces to $[\mathbf{def}\ \ldots\ \mathbf{in}\ unif\langle z,\theta' \wedge \theta\rangle \mid true\langle\rangle : unif\langle\theta\rangle \mid fail\langle\rangle]$ and finally it can commit and release $unif\langle z,\theta' \wedge \theta\rangle$, i.e., the updated constraints for the enclosing solution, and $true\langle\rangle$, i.e., the notification that the sub-goal has been proved. On the other hand, when θ and θ' are not consistent the message $fail\langle\rangle$ will be produced and then, by rule (3), the $abort$ can be generated. In this case the negotiation may abort by releasing its compensation $unif\langle z,\theta\rangle | fail\langle\rangle$, which will restore the original constraints and notify failed termination of the proof.

5. ENCODING ZERO-SAFE NETS

In Petri nets, *places* are repositories of resources, called *tokens*, and *transitions* fetch and produce tokens. Net configurations, called *markings*, are multisets of places, recording the number of tokens available in each place. Zero-safe nets (ZS nets) [5] extend (place/transition) Petri nets with the simplest mechanism for coordinating the activities of transitions into transactions: token passing on suitable places. The places of zero-safe nets are partitioned into ordinary and transactional ones (called *stable* and *zero*, respectively). Correspondingly, markings U can be seen as pairs (S, Z) with $U = S + Z$, where S and Z are the multisets of stable and zero resources, respectively. A transaction goes from a multiset of stable places (*stable marking*) to another stable marking. The key point is that stable tokens produced during a transaction are made available only at commit time, when no zero tokens are left. Thus transaction orchestration is entirely demanded to zero places. We write (T, S) for a ZS net with set of transitions T and initial stable marking S.

The operational semantics of ZS nets is defined by the

(1) $\mathit{unif}\langle w,\theta\rangle \blacktriangleright [\mathit{unif}\langle w,\theta\rangle : \mathit{unif}\langle w,\theta\rangle]$	(2) $\mathit{unif}\langle w,\theta\rangle \mid \mathit{proved}\langle w\rangle \blacktriangleright \mathit{tell}\langle w,\theta\rangle$
(3) $\mathit{fail}\langle\rangle \blacktriangleright \mathit{abort}$	(4) $\mathit{tell}\langle w,\theta_1\rangle \mid \mathit{unif}\langle z,\theta_2\rangle \blacktriangleright \mathit{and}\langle\theta_1,\theta_2,z,\mathit{fail}\rangle$

Figure 5: Constraint handling rules.

two relations $\Rightarrow$ and $\rightarrow$ in Figure 6. Transitions are written $U[\rangle U'$, with U and U' multisets of stable and zero places. Rules FIRING and STEP are the ordinary ones for Petri nets, for the execution of one/many transition(s). However, sequences of steps differ from the ordinary transitive closure of $\rightarrow$: The rule CONCATENATION composes zero tokens in series but stable tokens in parallel, hence stable tokens produced by the first step cannot be consumed by the second step. Transactions are step sequences from stable markings to stable markings, when CLOSE can be applied. The moves $(S,\emptyset) \Rightarrow (S',\emptyset)$ define all the atomic activities of the net, and hence they can be performed in parallel and sequentially as the transitions of an ordinary net. It is worth noting that a step $(S,\emptyset) \Rightarrow (S',\emptyset)$ can be itself the parallel composition of several transactions (by rule STEP).

A distributed Join implementation of ZS nets has been already presented in [4] and later used to write an efficient running prototype in Jocaml. However, the encoding in Join is complicated by the need of creating a local manager for each transition and implementing the commit and abort protocols. In this section we show that ZS nets can be straightforward encoded in committed Join. As done in [4], and without loss of generality, we restrict to nets made with the basic shapes in Figure 7(a) (which are shown to be as expressive as the general nets in [6]), for E any stable place and e, e_1, e_2 any zero places (uppercase letters denote stable places, while lowercase letters denote zero places).

The translation is illustrated in Figure 7(b). It associates a Committed Join definition with each basic shape of transitions. Places are seen as channel names and tokens as messages. Tokens in stable places carry no value, while tokens in zero places carry the identifier of the transaction to which they belong. For T the set of transitions and S the initial state of the ZS nets, we let $[\![T]\!] = \bigwedge_{t\in T}[\![t]\!]$ and then consider the Committed Join process $\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S]\!]$, which consists of the translation of the initial marking S in the environment containing all the definitions associated with transitions in T. Note that transitions whose pre-sets contain zero places are translated as merge definitions, otherwise as ordinary Join definitions. This is because zero places serve for orchestration purposes, while stable places are the main resources.

A transaction can be opened by firing a transition of the form E `open` e. In Committed Join, this means opening a new negotiation whose internal state contains the definition of a fresh name z (the identifier of the transaction), the message $e\langle z\rangle$, and whose compensation is set by default to give back the stable resources consumed so far ($E\langle\rangle$). Note that the dummy definition $z\langle\rangle \triangleright 0$ is just a convenient way of declaring the local identifier z in Join syntax and has no computational meaning. Since e is a name defined externally via merge definitions, then the reaction MERGE can be used to compute inside negotiations and merge them. When two transactions are merged with reaction rule $e_1\langle z_1\rangle|e_2\langle z_2\rangle \blacktriangleright e\langle z_1\rangle$, then the identifiers z_1 and z_2 become part of the same transaction and have the same scope, becoming equivalent identifiers for the same larger negotiation. When computing inside a negotiation, each zero token carries one of the equivalent identifiers for that negotiation. In the given definition we arbitrarily use z_1 to identify the larger negotiation (we could have used z_2 as well). If stable messages $E\langle\rangle$ are released inside the negotiation, e.g., by firing e `close` E, then they cannot be used until commit, because all the rules that can consume them are outside the negotiation boundaries, in the top chemical soup. The commit can happen if and only if the negotiation reaches a local state containing only stable messages $E\langle\rangle$ and definitions of transactions identifiers. Then, the reaction COMMIT can take place, closing the negotiations and releasing the stable tokens to the top level. Since we do not have nested negotiations, the version of Committed Join we consider does not have the chemical rule STR-MOVE, which is useless.

The main behavioral difference between the Committed Join encoding in Figure 7(b) and the Join encoding in [4] relies on the treatment of failures, as here no abort can be generated (and consequently compensations cannot be activated). A possible solution would be to add a *timeout* component each time a new negotiation is open, which is able to produce the abort via the rule $A = \mathit{timeout}\langle\rangle \blacktriangleright \mathit{abort}$. In this case, we should let $[\![E\ \mathtt{open}\ e]\!] = E\langle\rangle \triangleright [\mathbf{def}\ z\langle\rangle \triangleright 0\ \mathbf{in}\ e\langle z\rangle|\mathit{timeout}\langle\rangle : E\langle\rangle]$ and take $\mathbf{def}\ A \wedge [\![T]\!]\ \mathbf{in}\ [\![S]\!]$ as translation of the net.

We devote the rest of the section to prove the correspondence of our encoding and ZS nets.

LEMMA 1. *Let (T,S) be a* ZS *net. If $(S,Z) \rightarrow (S',Z')$ then* $\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S]\!] \mid \|_i[\mathbf{def}\ \wedge_k w_k\langle\rangle \triangleright 0\ \mathbf{in}\ \|_k[\![\mathcal{Z}_i^k]\!]_{w_k} : \ldots] \rightarrow^* \mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![\mathcal{S}'']\!] \mid \|_j[\mathbf{def}\ \wedge_l\ w_l\langle\rangle \triangleright 0\ \mathbf{in}\ [\![\mathcal{Z}_j^{'l}]\!]_{w_k} \mid [\![\mathcal{S}'_j]\!] : \ldots]$, *where $Z = \bigcup_i \bigcup_k \mathcal{Z}_i^k$, $Z' = \bigcup_j \bigcup_l \mathcal{Z}_j^{'l}$, $S' = \bigcup_j \mathcal{S}'_j \cup \mathcal{S}''$ and $[\![\_]\!]_w$ is defined as follow:*

$$[\![z]\!]_w ::= z\langle w\rangle \qquad\qquad [\![Z_1 + Z_2]\!]_w ::= [\![Z_1]\!]_w \mid [\![Z_2]\!]_w$$

PROOF SKETCH. By induction on the structure of the proof. The base case corresponds to rule FIRING and follows by analyzing the shape of the fired transition (according to Figure 7(a)). The inductive step follows by rule case analysis. Rule STEP corresponds to parallel computations in CHAM. The proof for CONCATENATION follows by applying inductive hypothesis on both premises obtaining two derivations and noting that, by CHAM semantics, a derivation is also valid when processes are added into a solution, i.e., if

$$\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S_1]\!] \mid \|_i[\mathbf{def}\ \wedge_k\ w_k\langle\rangle \triangleright 0\ \mathbf{in}\ [\![\mathcal{Z}_i^k]\!]_{w_k} : \ldots] \rightarrow^*$$
$$\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![\mathcal{S}_1''']\!] \mid \|_j[\mathbf{def}\ \wedge_l\ w_l\langle\rangle \triangleright 0\ \mathbf{in}\ [\![\mathcal{Z}_j^{''l}]\!]_{w_l} \mid [\![\mathcal{S}'_j]\!] : \ldots]$$

also the following derivation is possible:

$$\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S_2]\!] \mid [\![S_1]\!] \mid \|_i[\mathbf{def}\ \wedge_k w_k\langle\rangle \triangleright 0\ \mathbf{in}\ [\![\mathcal{Z}_i^k]\!]_{w_k} : \ldots] \rightarrow^*$$
$$\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S_2]\!] \mid [\![\mathcal{S}_1''']\!] \mid \|_j[\mathbf{def}\ \wedge_l w_l\langle\rangle \triangleright 0\ \mathbf{in}\ [\![\mathcal{Z}''^{l}{}_j]\!]_w \mid [\![\mathcal{S}'_j]\!] : \ldots]$$

Analogously, $\mathcal{S}'_j$ and $\mathcal{S}_1'''$ can be added to the derivation associated to the second premise. Finally, the derivation corresponding to the whole proof can be constructed by combining sequentially the two extended derivations. $\square$

THEOREM 1. *Let (T,S) be a* ZS *net. If $(S,\emptyset) \Rightarrow (S',\emptyset)$ then* $\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S]\!] \rightarrow^* \mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S']\!]$.

$$\frac{S + Z \; [\rangle \; S' + Z'}{(S + S'', Z + Z'') \rightarrow (S' + S'', Z' + Z'')} \; (\text{FIRING}) \qquad \frac{(S_1, Z_1) \rightarrow (S'_1, Z'_1) \quad (S_2, Z_2) \rightarrow (S'_2, Z'_2)}{(S_1 + S_2, Z_1 + Z_2) \rightarrow (S'_1 + S'_2, Z'_1 + Z'_2)} \; (\text{STEP})$$

$$\frac{(S_1, Z) \rightarrow (S'_1, Z'') \quad (S_2, Z'') \rightarrow (S'_2, Z')}{(S_1 + S_2, Z) \rightarrow (S'_1 + S'_2, Z')} \; (\text{CONCATENATION}) \qquad \frac{(S, \emptyset) \rightarrow (S', \emptyset)}{(S, \emptyset) \Rightarrow (S', \emptyset)} \; (\text{CLOSE})$$

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

PROOF SKETCH. By Lemma 1, there exists a derivation for $(S, \emptyset) \rightarrow (S', \emptyset)$ containing negotiations. Nevertheless, those negotiations do not contain messages to ports encoding zero-safe places, and therefore they can commit and release all messages to ports denoting stable places. Consequently, rule COMMIT can be applied repeatedly to finish all running negotiations. □

LEMMA 2. *Let (T, S) be a ZS net. If there exists* $\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S'']\!] \mid \|_i [\mathbf{def}\ \wedge_k\, w_k\langle\rangle \triangleright 0\ \mathbf{in}\ [\![\mathcal{Z}_i^k]\!]_{w_k} \mid [\![\mathcal{S}_i]\!] : \ldots] \rightarrow^*$ $\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![\mathcal{S}''']\!] \mid \|_j [\mathbf{def}\ \wedge_l\, w_l\langle\rangle \triangleright 0\ \mathbf{in}\ [\![\mathcal{Z}_j'^l]\!]_{w_l} \mid [\![\mathcal{S}'_j]\!] : \ldots]$ *then* $(S, Z) \rightarrow (S', Z')$, *where* $Z = \bigcup_i \bigcup_k \mathcal{Z}_i^k$, $Z' = \bigcup_j \bigcup_l \mathcal{Z}_j'^l$, $S = \bigcup_i \mathcal{S}_i \cup \mathcal{S}''$ *and* $S' = \bigcup_j \mathcal{S}'_j \cup \mathcal{S}'''$

PROOF SKETCH. By induction on the length of the derivation. The base case corresponds to an idle movement of the net. The inductive step follows by analyzing the last applied rule in the reduction. For RED and MERGE, the proof is constructed by using inductive hypothesis on the initial derivation until the last step, and also for the final step. Then by analyzing the different shapes of the definition used for the last reduction (Figure 7(b)) conclude that the only possible cases correspond to rule CONCATENATION. Instead, a commit step at the end corresponds to an idle movement of the net. Note that, there are no ABORT steps, because the presented encoding does not generate process *abort*. □

THEOREM 2. *Let (T, S) be a ZS net. If* $\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S]\!] \rightarrow^*$ $\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S']\!]$ *then* $(S, \emptyset) \Rightarrow (S', \emptyset)$.

PROOF SKETCH. Immediate from Lemma 2, by noting that all negotiations can be finished by applying several commit steps. □

THEOREM 3. *Let $N = (T, S)$ be a ZS net. Then* $(S, \emptyset) \Rightarrow^* (S', \emptyset)$ *iff* $\mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S]\!] \rightarrow^* \mathbf{def}\ [\![T]\!]\ \mathbf{in}\ [\![S']\!]$.

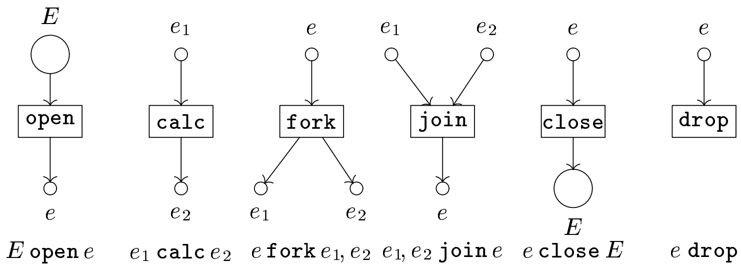
PROOF SKETCH. Immediate from Theorems 1 and 2. □

6. SERIALIZABILITY AND BIG STEP SEMANTICS

Traditionally, any concurrent execution $T_1 \parallel \ldots \parallel T_n$ of several ACID transactions is considered correct if it is *serializable*, i.e., 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 steps from different transactions) and produces the same result [1]. Serializability allows to reason about the behavior of a system by considering just the execution of one transaction at a time, provided that transactions do not interfere with each other. In committed Join this is not the case because merging definitions allow negotiations to cooperate in order to carry on a particular activity. Nevertheless, we would like to be able to reason about the behavior of a process by analyzing interacting negotiations independently from the rest of the system. On the other hand, serializable transactions can be described in an abstract representation of a system as single transitions between stable states, i.e., from the state that begins a transaction to the state obtained after the commit. By analogy, we would like to think about a set of cooperating negotiations in committed Join as a transition mapping the state previous to the independent creation of each cooperating negotiation to the state obtained after the joint commit. Moreover, we would like this property to hold at every level of the nesting hierarchy. Therefore, we will consider negotiations to be serializable when the beginning of a sub-negotiation can be postponed until all the cooperating sub-negotiations needed to commit can be generated inside its parent negotiation. After being created, all interacting sub-negotiations compute isolated from the system, although interactions among them are possible. Well-programmed negotiations eventually will decide to commit or abort, producing a stable state. At an abstract level, those computations can be viewed as transitions starting with the consumption of messages needed to initiate all interacting sub-negotiations and finishing by generating the processes released at commit or abort.

In this section we introduce a syntactical restriction on processes, called *shallow*, that guarantees serializability. We characterize serializability by giving the big step semantics of Committed Join for shallow processes. Shallow condition imposes a discipline for activating sub-negotiations. That is, definitions in a shallow process are such that: (i) merging definitions do not contain active negotiations; and (ii) guarded processes in ordinary definitions (i.e., the sub-terms P occurring as $J \triangleright P$) either do not contain active negotiations or consist of a unique negotiation $[P : Q]$ where P and Q have no active negotiations. Condition (i) forbids the creation of sub-negotiations and merging of negotiations at the same time. This prevents the possibility of postponing the activation of the created sub-negotiations until all interacting sub-negotiations could be activated. Condition (ii) assures that the firing of an ordinary rule increases the height of the nesting structure of at most one level. Moreover, every definition produces just stable processes, i.e., processes without active negotiations, or a unique sub-negotiation without nesting sub-contracts.

The shallow condition forbids definitions such as $J_1 \triangleright P \mid [P_1 : Q]$ and $J_2 \triangleright [[P_1 : Q_1] : Q]$. Nevertheless, they can be easily encoded as shallow definitions by using local new ports. For instance, the previous examples can be written as $D_1 \equiv J_1 \triangleright x\langle\rangle | P \ \wedge\ x\langle\rangle \triangleright [P_1 : Q]$ and $D_2 \equiv J_2 \triangleright [\mathbf{def}\ x\langle\rangle \triangleright [P_1 : Q_1]\ \mathbf{in}\ x\langle\rangle : Q]$. It should be clear that we are not limiting the nesting depth of a running negotiation. In fact, the shallow process $\mathbf{def}\ D_2\ \mathbf{in}\ J_2\sigma$ reduces to $\mathbf{def}\ D_2\ \mathbf{in}\ [\mathbf{def}\ x\langle\rangle \triangleright [P_1\sigma : Q_1\sigma]\ \mathbf{in}\ x\langle\rangle : Q\sigma]$, and then



(a) Basic shapes of ZS nets transitions.

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

$$
\begin{aligned}
[\![E]\!] &::= E\langle\rangle \\
[\![S_1 + S_2]\!] &::= [\![S_1]\!] | [\![S_2]\!]
\end{aligned}
$$

(b) Translation of basic shapes and stable markings.

Figure 7: Encoding of ZS Nets in Committed Join.

to **def** D_2 **in** [**def** $x\langle\rangle \triangleright [P_1\sigma : Q_1\sigma]$ **in** $[P_1\sigma : Q_1\sigma] : Q\sigma]$, which has nested negotiations.

In the following, $\mathcal{P}$ and $\mathcal{Q}$ will denote shallow processes, $\mathcal{D}$ a shallow definition, $\mathcal{S}$ a shallow process without active negotiations, and $\mathcal{B}$ a shallow definition containing just merging rules. We will abbreviate **def** $\mathcal{D}$ **in** $\mathcal{P}$ as $\mathcal{D} \vdash \mathcal{P}$, and we will write $\vdash \mathcal{P}$ for $\mathcal{P}$. Moreover, we will consider terms up-to structural equivalence, given by the following rule

$$
\begin{aligned}
&\mathcal{D} \vdash \mathcal{P} \mid \mathbf{def}\ \mathcal{D}'\ \mathbf{in}\ \mathcal{Q} \quad \equiv \quad \mathcal{D} \wedge \mathcal{D}'\sigma_{dn} \vdash \mathcal{P} \mid \mathcal{Q}\sigma_{dn} \\
&\quad range(\sigma_{dn}) \cap (fn(\mathcal{D}) \ \cup\ fn(\mathcal{P}) \ \cup\ fn(\mathbf{def}\ \mathcal{D}'\ \mathbf{in}\ \mathcal{Q})) \ = \ \emptyset
\end{aligned}
$$

plus equations stating that 0 is the unit for | and that the operators | and $\wedge$ are associative and commutative. This rules are analogous to the structural rules of the Join calculus.

The big step semantics is given by inference rules in Figure 8. Note that steps can be composed in parallel (PAR) and sequentially (SEQ), even with idle transitions (IDLE). Rule GLOBAL FIRING corresponds to the firing of an ordinary definition in a top-level process. Instead LOCAL FIRING states possible internal transitions of a running contract. LOCAL FIRING captures the idea that certain computations of sub-negotiations can be represented as ordinary transitions at an abstract level by relating computations at different level of abstraction. In fact, the only computations occurring at a lower level in the nesting hierarchy (premise of LOCAL FIRING) that are relevant to its upper level are those relating processes that do not contain active negotiations, i.e., $\mathcal{S}$ and $\mathcal{S}'$. Note that this rule does not relate every computation in the lower level with a computation in the upper level, as could do a rule like the following:

$$
\frac{\widetilde{\mathcal{B}} \vdash \mathcal{P} \ \twoheadrightarrow^* \ \widetilde{\mathcal{B}} \vdash \mathcal{P}'}{\mathcal{D} \wedge \mathcal{B} \vdash [\mathcal{P} : \mathcal{Q}] \ \twoheadrightarrow \ \mathcal{D} \wedge \mathcal{B} \vdash [\mathcal{P}' : \mathcal{Q}]}
$$

Additionally, note that the definitions available to a process contained in a negotiation are those explicitly defined by it and by the merging definitions introduced by the nesting process. In fact, a merging definitions can be applied considering only a single contract, behaving in this case as an ordinary reduction defined in the global scope. The operator $\widetilde{\ }$ transforms merging definitions of the containing process in ordinary ones. It is defined as follows.

$$
\widetilde{J \blacktriangleright \mathcal{P}} \quad \equiv \quad J \triangleright \mathcal{P} \qquad\qquad \widetilde{\mathcal{B} \wedge \mathcal{B}'} \quad \equiv \quad \widetilde{\mathcal{B}} \wedge \widetilde{\mathcal{B}'}
$$

As an additional consideration, rule LOCAL FIRING presented in Figure 8 corresponds to the version of committed Join without structural rule STR-MOVE. When such rule is added, the appropriate version of LOCAL FIRING should be

$$
\frac{\widetilde{\mathcal{B}} \wedge \mathcal{B} \vdash \mathcal{S} \ \twoheadrightarrow^* \ \widetilde{\mathcal{B}} \wedge \mathcal{B} \vdash \mathcal{S}'}{\mathcal{D} \wedge \mathcal{B} \vdash [\mathcal{S} : \mathcal{Q}] \ \twoheadrightarrow \ \mathcal{D} \wedge \mathcal{B} \vdash [\mathcal{S}' : \mathcal{Q}]}
$$

that allows merging definitions to be used at any level of the nesting structure. Rules LOCAL COMMIT and ABORT handle the termination of the execution of a negotiation, whereas MERGE describes the interaction of sibling negotiations. Note that negotiations can be joined only if they do not contain running contracts.

Note also that the big step relation described previously corresponds to our notion of serializability. In fact, the completed negotiations at a particular level are represented as ordinary transitions at the upper level and all the interacting transactions can be analyzed independently from the rest of the system.

The following results state the correspondence between the two semantics for shallow processes, and consequently assure serializability of shallow processes.

PROPOSITION 2. *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 an equivalent derivation* $\mathcal{D}, \mathcal{S}, \otimes_k[\mathcal{S}'_k : \mathcal{S}''_k] \rightarrow^* \mathcal{D}, \otimes_i[\mathcal{S}'''_i : \mathcal{S}_i] \rightarrow \mathcal{D}, [\mathcal{S}' : \|_i\mathcal{S}_i] \rightarrow^* \mathcal{D}, [\mathcal{P} : \ \|_i\mathcal{S}_i]$.

Proposition 2 states that for any derivation starting from a process without nested negotiations there exists an equivalent reduction that merges only negotiations without active sub-negotiations.

LEMMA 3. $\mathcal{D}, \otimes_i[\mathcal{S}_i : \mathcal{P}_i], \mathcal{S} \rightarrow^* \mathcal{D}, \otimes_j[\mathcal{S}'_j : \mathcal{P}'_j], \mathcal{S}'$ *iff* $\mathcal{D} \vdash \|_i[\mathcal{S}_i : \mathcal{P}_i] \mid \mathcal{S} \ \twoheadrightarrow \ \mathcal{D} \vdash \|_j[\mathcal{S}'_j : \mathcal{P}'_j] \mid \mathcal{S}'$.

PROOF SKETCH. ($\Rightarrow$) By induction on the length of the derivation. The base case corresponds to the IDLE axiom. The inductive step follows by rule case analysis of the first applied derivation rule. 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 negotiation or a new stable process, and (ii) when it generates a sub-negotiation in one

$$\textsc{(idle)}\quad \mathcal{D} \vdash \mathcal{P} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P} \qquad \textsc{(par)}\quad \frac{\mathcal{D} \vdash \mathcal{P} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}' \qquad \mathcal{D} \vdash \mathcal{Q} \twoheadrightarrow \mathcal{D} \vdash \mathcal{Q}'}{\mathcal{D} \vdash \mathcal{P} \mid \mathcal{Q} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}' \mid \mathcal{Q}'} \qquad \textsc{(seq)}\quad \frac{\mathcal{D} \vdash \mathcal{P} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}'' \qquad \mathcal{D} \vdash \mathcal{P}'' \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}'}{\mathcal{D} \vdash \mathcal{P} \twoheadrightarrow \mathcal{D} \vdash \mathcal{P}'}$$

$$\textsc{(global firing)}\quad \mathcal{D} \wedge J \triangleright \mathcal{P} \vdash J\sigma \twoheadrightarrow \mathcal{D} \wedge J \triangleright \mathcal{P} \vdash \mathcal{P}\sigma \qquad \textsc{(local firing)}\quad \frac{\widetilde{\mathcal{B}} \vdash \mathcal{S} \twoheadrightarrow \widetilde{\mathcal{B}} \vdash \mathcal{S}'}{\mathcal{D} \wedge \mathcal{B} \vdash [\mathcal{S} : \mathcal{Q}] \twoheadrightarrow \mathcal{D} \wedge \mathcal{B} \vdash [\mathcal{S}' : \mathcal{Q}]} \qquad \textsc{(local commit)}\quad \mathcal{D} \vdash [M \mid \mathcal{D}' \vdash 0 : \mathcal{S}] \twoheadrightarrow \mathcal{D} \vdash M$$

$$\textsc{(abort)}\quad \mathcal{D} \vdash [abort|\mathcal{P} : \mathcal{S}] \twoheadrightarrow \mathcal{D} \vdash \mathcal{S} \qquad \textsc{(merge)}\quad \mathcal{D} \wedge \|_i\ J_i \blacktriangleright \mathcal{S} \vdash \|_i[\mathcal{D}_i \vdash J_i\sigma \mid \mathcal{S}_i : \mathcal{Q}_i] \twoheadrightarrow \mathcal{D} \wedge \|_i\ J_i \blacktriangleright \mathcal{S} \vdash [\textstyle\bigwedge_i \mathcal{D}_i \vdash (\|_i\ \mathcal{S}_i\ )|\ \mathcal{S}\sigma : \|_i\ \mathcal{Q}_i]$$

Figure 8: Big step semantics of Committed Join.

of the existing negotiations. First case is immediate by inductive hypothesis and SEQ. In the second case, there is a negotiation 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 9, and then the proof follows by using IDLE for the non-modified processes and inductive hypothesis and SEQ. The remaining possibility is when sub-negotiation $[\mathcal{S}'' : \mathcal{P}']$ can finish only after the parent negotiation is merged. Then by Proposition 2 an equivalent derivation $\mathcal{D}, \otimes_i[\mathcal{S}_i : \mathcal{P}_i], S \to^* \mathcal{D}, [S'' : \|_i\mathcal{P}_i], S''' \to^* \mathcal{D}, [\mathcal{P} : \|_i\mathcal{P}_i], S''' \to^* \mathcal{D}, \otimes_j[\mathcal{S}'_j : \mathcal{P}'_j], \mathcal{S}'$ that merges only negotiations not containing sub-negotiations can be found. Then, proof follows by applying inductive hypothesis for 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 4. *Let $\mathcal{S}, \mathcal{S}'$ be shallow processes with not running negotiations. Then $\mathcal{S} \to^* \mathcal{S}'$ iff $\vdash \mathcal{S}' \twoheadrightarrow\ \vdash \mathcal{S}'$.*

7. CONCLUDING REMARKS

We have proposed a linguistic extension of Join with natural primitives for modeling nested negotiations. The expressiveness of the resulting PDL, called *Committed Join*, has been demonstrated by means of: (1) an informal discussion about the satisfaction of the general requirements enumerated in the Introduction; (2) two examples of common applications; and (3) the encoding in Committed Join of zero-safe nets. Additionally, we defined a syntactical restriction of processes that assures serializability, which has been characterized through a big step semantics.

In Committed Join, all dependencies between negotiations (commit or abort dependencies) must be programmed explicitly by using merging definitions in a proper way. Unlike WMS approaches, Committed Join does not fix a set of constructors to describe execution dependency of transaction in the style of, e.g., CONTRACTS model [15]. Neither it states a default mechanism for undoing pre-committed activities of aborted transactions as in [10] and in MINI-XLANG [3].

Negotiations in Committed Join have the flavour of multiway transactions in ZS-Nets, where participants are not statically fixed. Moreover, our extension distinguishes between local and global resources (i.e., names), and the notion of negotiation relies mainly on this distinction. In [4], an encoding of ZS nets in the Join was proposed, providing a distributed implementation of transactions, centered on the novel Distributed Two Phase Commit (D2PC) protocol. A prototype implementation of the D2PC has been developed in Jocaml: it takes the description of a ZS net N, and then it generates a Jocaml program that efficiently models the behavior of N. We plan to reuse or to extend the D2PC in order to have a suitable implementation of Committed Join in Join itself. This would allow us to extend distributed implementations of the Join, such as Jocaml or Polyphonic C$^\sharp$, by providing primitives for handling negotiations in distributed applications.

Acknowledgements. We would like to thank Nick Benton, Luca Cardelli, Cédric Fournet and Cosimo Laneve with whom we discussed preliminary versions of the linguistic extension of Join.

8. REFERENCES

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

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

[3] L. Bocchi, C. Laneve, and G. Zavattaro. Transactions with an extended π-calculus. Unpublished manuscript, 2003.

[4] R. Bruni, C. Laneve, and U. Montanari. Orchestrating transactions in join calculus. In L. Brim, P. Jancar, M. Kretinsky, and A. Kucera, editors, *Proceedings of CONCUR 2002, 13th International Conference on Concurrency Theory*, volume 2421 of *Lect. Notes in Comput. Sci.*, pages 321–336. Springer Verlag, 2002.

[5] R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Inform. and Comput.*, 156(1-2):46–89, 2000.

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

[7] L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of FoSSaCS'98, Foundations of Software Science and Computational Structures*, volume 1378 of *Lect. Notes in Comput. Sci.*, pages 140–155. Springer Verlag, 1998.

$$
\dfrac{\dfrac{\dfrac{\vdots\ \textsc{global firing,idle,par}}{\widetilde{\mathcal{B}} \vdash \mathcal{S}_k \ \rightarrow\ \widetilde{\mathcal{B}} \wedge \mathcal{D}'' \vdash [\mathcal{S}'' : \mathcal{P}'] \mid \mathcal{S}'_k} \qquad \widetilde{\mathcal{B}} \wedge \mathcal{D}'' \vdash [\mathcal{S}'' : \mathcal{P}'] \mid \mathcal{S}'_k \ \rightarrow\ \widetilde{\mathcal{B}} \vdash \mathcal{S}''_k \ \textsc{ind. hyp.}}{\widetilde{\mathcal{B}} \vdash \mathcal{S}_k \ \rightarrow\ \widetilde{\mathcal{B}} \vdash \mathcal{S}''_k}\ \textsc{seq}}{\mathcal{D}' \wedge \mathcal{B} \vdash [\mathcal{S}_k : \mathcal{P}_k] \ \rightarrow\ \mathcal{D}' \wedge \mathcal{B} \vdash [\mathcal{S}''_k : \mathcal{P}_k]}\ \textsc{local firing}
$$

Figure 9: Proof sketch for Lemma 3

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

[9] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the Join calculus. In *Proceedings of POPL'96, 23rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[10] H. Garcia-Molina and K. Salem. Sagas. In U. Dayal and I. Traiger, editors, *Proceedings of the ACM Special Interest Group on Management of Data Annual Conference*, pages 249–259. ACM Press, 1987.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In D. Warren and P. Szeredi, editors, *Proceedings of ICLP'90, 7th International Conference on Logic Programming*, pages 31–46. MIT Press, 1990.

[13] S. Haridi, S. Janson, and C. Palamidessi. Structural operational semantics of akl. *Journal of Future Generation Computer Systems*, 8:409–421, 1992.

[14] N. Hutchinson, G. Kaiser, and C. Pu. Split-transactions for open-ended activities. In F. Bancilhon and D. DeWitt, editors, *Proceedings of VLDB'88, 14th International Conference on Very Large Data Bases*, pages 26–37. Morgan Kaufmann, 1988.

[15] A. Reuter and H. Wächter. *Transaction Models for Advanced Applications*, chapter The Contract Model. Morgan Kaufmann, 1992.

[16] U. Roxburgh. Biztalk orchestration: Transactions, exceptions, and debugging, 2001. Microsoft Corporation. Available at `http://msdn.microsoft.com/library/en-us/dnbiz/html/bizorchestr.asp`.

[17] A. Sheth and D. Worah. Transactions in transactional workflows. In *Advanced Transaction Models and Architectures*, pages 3–34. Kluwer, 1997.