NESTED COMMITS FOR MOBILE CALCULI: EXTENDING JOIN *

Roberto Bruni, Hernán Melgratti, Ugo Montanari
Dipartimento di Informatica, Università di Pisa, Italia.
{ bruni, melgratt, ugo } @di.unipi.it

Abstract In global computing applications the availability of a mechanism for some form of committed choice can be useful, and sometimes necessary. It can conveniently handle, e.g., distributed agreements and negotiations with nested choice points. We propose a linguistic extension of the Join calculus for programming nested commits, called *Committed* Join (cJoin). It provides primitives for explicit abort, programmable compensations and interactions between negotiations. We give the operational semantics of cJoin in the *reflexive* CHAM style. Then we discuss its expressiveness on the basis of a few examples and encodings. Finally, we provide a big-step semantics for cJoin processes that can be typed as *shallow* and we show that shallow processes are serializable.

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. Such applications often require a coordination layer to orchestrate their components, which are designed and implemented separately, run on different platforms and communicate asynchronously. Often, the components must agree on the activities they are carrying on (e.g. in terms of transactions, like in [16, 7]) by committing the results of long distributed decision processes as soon as the participants reach partial agreements. Applications can handle these situations in an ad hoc manner or they can rely on a fixed set of coordination primitives. In this work we are interested on studying suitable primitives for describing distributed commits in GC applications. Note that we use the term "*commit*" (also *contract* or *negotiation*) instead of "*transaction*" to emphasize the coordination aspects, which are orthogonal to ACID database transactions. For instance, in

* Research supported by the MSR Cambridge Project NAPI, by the FET-GC Project IST-2001-32747 AGILE, by the MIUR Projects COFIN COMETA and IS-MANET, and by the MURST-CNR 1999 Project.

the case of web services, the orchestration layer should provide the primitives for specifying transactional services and the valid interactions between them. Nevertheless, any service should be responsible for maintaining the consistency on their local data (i.e., assuring ACID properties on them).

Process description languages (PDLS) are mathematical models of computation designed for isolating and studying phenomena that occur in concurrent languages. In the spirit of PDL, it would be desirable to extend well-known calculi with primitives for distributed nested commits. Two key operations are the "abort with compensation" (e.g., to stop a negotiation when some participants withdraw their interest in carrying out the contract) and the "commit" (to store a partial agreement before moving to the next phase of a long negotiation). In this paper we introduce *Committed* Join (cJoin) as an extension of the Join calculus [12]. The features of cJoin are compared against two other paradigms with commit, namely AKL [13] and *Zero-Safe* nets [8].

The design of cJoin has been inspired by the requirements (i)–(vi) below. Contracts are 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 be stored locally and not made public before a common agreement is achieved; (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 activated (e.g., upon abort, new attempts for finding an agreement can be initiated) ; (iv) divergence is possible, but well designed GC applications should guarantee that each contract eventually leads to an abort or a commit; (v) when two processes involved in separate negotiations exchange some information, then their contracts should be merged into a unique one; (vi) it should be possible to have nested contracts. Though an internal abort can be compensated in such a way that the main contract can still be successfully completed, a failure of the main contract should cause the abort of all ongoing internal activities.

We define the small-step operational semantics of cJoin in the *reflexive* CHAM style [12]. We also give a big-step semantics for the sub-calculus of *shallow* processes and we show that shallow processes are serializable by proving a correspondence between their CHAM and big-step semantics. *Serializability* ensures the correctness of reasoning at different levels of abstractions when transactions become atomic transitions at the abstract level.

Synopsis. In § 2 we recall the principles of the CHAM and the syntax and semantics of Join. Committed Join is introduced in § 3. In § 4 we illustrate the main features of cJoin by showing a simple application for booking trips and the encoding of AKL. The implementation of ZS nets is presented in § 5. Finally, we study *serializability* in § 6.

2. Background: CHAM and Join

The Chemical Abstract Machine. In CHAM [3] 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 $\otimes_i\ m_i$. Solutions can be structured in a hierarchical way by using the operator *membrane* $\{\![.]\!\}$ to group a solution s into a molecule $\{\![s]\!\}$. In [3] 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* (or *structural*) rules $\leftrightharpoons$ representing syntactical rearrangements of molecules in a solution, and *reaction* rules $\rightarrow$. Structural rules are reversible: a solution obtained by applying a cooling rule can be heated back to the original state, and vice versa. Instead, reaction rules cannot be undone.

The laws governing 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 in every larger solution $S, S_1 \rightarrow S, S_2$ (*chemical law*) and also in grouped sub-solutions, $\{\![S, S_1]\!\} \rightarrow \{\![S, S_2]\!\}$. In particular, they can be nested at any level of hierarchical solutions. Note that, since solutions are multisets, rules can be applied concurrently.

The Join calculus. The Join calculus [12] is a well-known PDL with asynchronous name-passing communication. It has the same expressive power as the asynchronous π-calculus and it has distributed running implementations, e.g. Jocaml [10] and Polyphonic C$^\sharp$ [1]. Join 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 1.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 defined as usual.

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 correspond to terms of the Join calculus denoting processes or definitions. The chemical rules are shown in Figure 1.b. 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

(Proc)
$P, Q ::= 0 \mid x\langle\vec{y}\rangle \mid \textbf{def}\ D\ \textbf{in}\ P \mid P|Q$

(Def)
$D, E ::= J \triangleright P \mid D \wedge E$

(Pat)
$J, K ::= x\langle\vec{y}\rangle \mid J|K$

a. Syntax.

(STR-NULL) $0 \leftrightharpoons$

(STR-JOIN) $P \mid Q \leftrightharpoons P, Q$

(STR-AND) $D \wedge E \leftrightharpoons D, E$

(STR-DEF) $\textbf{def}\ D\ \textbf{in}\ P \leftrightharpoons D\sigma_{dn(D)}, P\sigma_{dn(D)}$
$(range(\sigma_{dn(D)})$ globally fresh)

(RED) $J \triangleright P, J\sigma \rightarrow J \triangleright P, P\sigma$

b. Semantics.

Figure 1. Join Calculus.

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

Figure 2 Syntax of cJoin.

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 $x_1 \dots x_n$ by names $y_1 \dots y_n$ as $\sigma = \{^{y_1 \cdots y_n}/_{x_1 \dots x_n}\}$, with $dom(\sigma) = \{x_1, \dots, x_n\}$ and $range(\sigma) = \{y_1, \dots, y_n\}$. We indicate with σ_N an injective substitution σ such that $dom(\sigma) = N$. We require names to be globally fresh, i.e. fresh w.r.t the implicit context in which the rule is applied.

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

Finally, RED describes the use of an active reaction rule ($J \triangleright P$) to consume messages 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. By applying RED to s' for $\sigma = \{^a/_y\}$, we get $s' \rightarrow \{\!\![z\langle x, z\rangle, x_1\langle y\rangle \triangleright z\langle y, x_1\rangle, z\langle a, x_1\rangle]\!\!\}$. Note that the local port x_1 has been extruded on the free channel z.

3. Committed Join

Syntax. We extend the syntax of Join as in Figure 2. A negotiation is represented by $[P : Q]$, where P is the normal activity and Q is its compensation. The normal activity P is intended to execute in isolation until reaching either a commit or an abort decision. If P commits, the obtained result is delivered to the outside of the negotiation. Instead, Q is activated when P aborts. The abort decision is signaled with the special process $abort$.

A new kind of definitions $J \blacktriangleright P$, called *merge definitions*, is introduced to describe the interactions among negotiations. Merge definitions allow the

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

Figure 3. Ordinary and merge names.

$$
\begin{array}{ll}
(\textsc{str-null}) & 0 \leftrightharpoons \\
(\textsc{str-join}) & P \mid Q \leftrightharpoons P, Q \\
(\textsc{str-and}) & D \ \wedge \ E \leftrightharpoons D, E \\
(\textsc{str-def}) & \textbf{def } D \textbf{ in } P \leftrightharpoons D\sigma_{dn(D)}, P\sigma_{dn(D)} \ \ (\mathit{range}(\sigma_{dn(D)}) \text{ globally fresh}) \\
(\textsc{red}) & J \triangleright P, J\sigma \rightarrow J \triangleright P, P\sigma \\
(\textsc{str-cont}) & [P : Q] \leftrightharpoons \{[P, \llcorner Q \lrcorner]\} \\
(\textsc{commit}) & \{[M | \textbf{def } D \textbf{ in } 0, \llcorner Q \lrcorner]\} \rightarrow M \\
(\textsc{abort}) & \{[abort | P, \llcorner Q \lrcorner]\} \rightarrow Q \\
(\textsc{merge}) & \Pi_j J_j \blacktriangleright P, \bigotimes_i \{[J_i\sigma, S_i, \llcorner Q_i \lrcorner]\} \rightarrow \Pi_j J_j \blacktriangleright P, \{[\bigotimes_i S_i, P\sigma, \llcorner \Pi_i Q_i \lrcorner]\}
\end{array}
$$

Figure 4. Operational semantics of cJoin.

consumption of messages produced in the scope of different contracts by joining all participants in a unique larger negotiation. Moreover, usual definitions can be used to create negotiations dynamically. For instance, by firing $J \triangleright [P : Q]$ a new instance of the negotiation P with compensation Q is activated.

For convenience we introduce the syntactical category M of processes without definitions, i.e. a parallel composition of messages.

The definition of fn is extended with $fn(abort) = \emptyset, fn([P : Q]) = fn(P) \cup fn(Q)$ and $fn(J \blacktriangleright P) = dn(J) \cup (fn(P) \backslash rn(J))$. For a definition D, we redefine $dn(D) = dn_{\mathrm{o}}(D) \cup dn_{\mathrm{m}}(D)$, where dn_{o} denotes the *defined ordinary* names and dn_{m} the *defined merge* names (Figure 3). We assume $dn_{\mathrm{o}}(D) \cap dn_{\mathrm{m}}(D) = \emptyset$ for every definition D.

Operational Semantics. The operational semantics of cJoin is defined in the reflexive CHAM style. Molecules m and solutions S are defined below.

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

As in ordinary Join, processes and definitions are molecules. Additionally, a molecule $\llcorner P \lrcorner$ denotes a compensation that is frozen inside a solution. The chemical rules are in Figure 4. The first five rules are the ordinary ones for Join. Rule STR-CONT describes how a negotiation corresponds to a sub-solution of two molecules: the process P and its compensation Q, which is frozen (because the operator $\llcorner . \lrcorner$ forbids the enclosed process to react).

COMMIT can be executed only when all participants have done their tasks reaching a (local) state that does not contain locally defined names. This way, a commit means clean termination where all names denoting internal states of contracts have been consumed. Note that all definitions belonging to a contract are discarded at commit time because the messages that are being released do not contain those names (we recall that local names cannot be extruded). Similarly, the compensation is discarded at commit. Moreover, a negotiation cannot commit when $abort$ is within the solution because $abort$ is not a message. The abort is handled by rule ABORT, which activates the compensation procedure while discarding all terms in the solution. Compensations are not predefined to execute atomically, but they can be explicitly programmed as negotiations.

Interactions among contracts are specified by rule MERGE, which consumes messages $J_i\sigma$ from different contracts and creates a new larger negotiation by combining the existing contracts together with the new instance $P\sigma$, where $dom(\sigma) = rn(\Pi_j J_j)$. The compensation for the joint negotiation is the parallel composition of all the original compensations. When merging negotiations, clashes of locally defined names should be avoided by imposing the side condition $dn(S_i) \cap (dn(S_j) \cup fn(S_j)) = \emptyset$ for $i \neq j$. However, if we are guaranteed that STR-DEF generates globally fresh names (and not just locally fresh names) then this side condition can be safely omitted, as it is trivially satisfied.

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

Discussion. Sibling contracts can be merged only by using merge definitions introduced by their parent. In practice, it might be useful to apply a merge definition provided by any ancestor. To this aim, the rule STR-MOVE below might be added, so that merge definitions could float across contract boundaries.

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

Regarding deadlocks, note that stall negotiations are not discarded. For instance, the process $[\textbf{def}\ x\langle\rangle|y\langle\rangle \triangleright 0\ \textbf{in}\ x\langle\rangle : Q]$ cannot compute. Neither it can commit, because there is a message on the local port x. 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 non-termination, which is an undecidable problem. On the other hand, we do not want to limit the expressiveness of the language.

With respect to the requirements discussed in the Introduction, we have that, membranes are exploited to define the boundaries of negotiations. Process like $[P_1 \mid [P_2 : Q_2] \mid [P_3 : Q_3] : Q_1]$ straightforwardly model sub-negotiations. The decisions taken internally by P_2 can influence P_3 only if some merge definition is available at the level of P_1. 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. At commit time, the result of the negotiation is made available to P_1. An abort generated in P_2 activates the compensation Q_2 at the level of P_1, neither forcing the abort of

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

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

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

Figure 5. Trip booking example.

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 Q_2 is the union of all the compensations of the participants.

An important restriction is that local resources can neither cross negotiations boundaries nor be extruded to siblings negotiations. The only way to exchange information between siblings negotiations is by merging all participants into a unique negotiation that must then commit, or abort, or diverge as such.

4. Examples

Trip booking. Figure 5 shows the encoding of the application Trip that allows a user to book flights and accommodations. Trip is defined in term of three components: the hotel H, the airline A and the customer C. The component H is a process that activates (by firing the definition for $WaitBooking$) a negotiation to serve customer requests. Such negotiation starts by publishing on the merge port $\mathit{offerRoom}$ (defined in Trip) the names of the services a client should use to reserve a room: $request$ to ask for a quote; and $\mathit{confirm}$ to accept an offer. The component A (omitted in Figure 5) is defined analogously, but it publishes services on port $\mathit{offerFlight}$ instead of $\mathit{offerRoom}$.

The component C defines two rules for creating negotiations: one for booking rooms and the other for buying flight tickets. Both contracts are quite similar. In particular, the negotiation for booking a room starts by sending a message to the merge port $searchRoom$ (defined in Trip) to obtain the names for interacting with a hotel. The first merge rule in Trip will associate an offer from a hotel with a request from a client by sending the names r and c to the corresponding port *hm*. Once received r and c on *HotelMsg*, C uses r to send a message to H for asking for a quote. Then, the hotel will answer with an offer on port offer. Whether the customer accepts or not a particular quote is modeled by the multiple definitions for the pattern $\mathit{offer}\langle\$\rangle$ in C. If the offer is not adequate then C can abort the negotiation, which will activate the compensation $Q \mid Q'$ (analogously for H and A). C can accept the offer by sending a confirmation message on port c. In this case, C also generates a message to the local port $\mathit{hotelOK}\langle \mathit{hConf}\rangle$. This message will be managed by the local merge rule defined by C. The contract will be blocked until a running negotiation for buying flight tickets generates a message on $\mathit{flightOK}$. At this time, the local merge definition can be fired and both contracts merged. Eventually the negotiation will commit by releasing the messages on *HotelFound* and *FlightFound*. Moreover, messages $\mathit{BookedRoom}\langle v\rangle$ and $\mathit{SoldFlight}\langle v\rangle$ generated by H and A to change their local states are released only at this time, when all participants have committed.

Andorra Kernel Language. As a second example, we sketch how merge definitions and nesting can be used to model some features of AKL [13], a concurrent logic programming language. We consider guarded rules $A \leftarrow G|B$, where the *head* A is an atom, the *guard* G and the *body* B are (possibly empty) conjunction of atoms, and | is the commit operator. An AKL program $\mathcal{P}$ is a list of guarded rules. An execution of $\mathcal{P}$ is initiated by providing a *goal* $\mathcal{G}$, which is a conjunction of atoms. A cJoin process that simulates $\mathcal{P}$ queried with $\mathcal{G}$ is:

$$\mathbf{def}\ \mathsf{D} \wedge [\![\mathsf{defs}(\mathcal{P})]\!] \wedge [\![\mathsf{undef}(\mathcal{P})]\!]\ \mathbf{in}\ [\![\mathbf{and}(\mathcal{G})]\!]_{trueG,falseG} \mid \mathit{unif}\langle \mathit{final}, \mathbf{tt}\rangle$$

The definitions in D are needed to promote constraints computed locally. The rules in $\mathcal{P}$ are translated separately by grouping all rules defining the same atom A. Such partition is denoted by $\mathsf{defs}(\mathcal{P})$, while $\mathsf{undef}(\mathcal{P})$ denotes all atoms in $\mathcal{P}$ without defining rules, i.e., atoms whose proofs will always fail. Constraints θ are encoded conveniently as cJoin processes ($\mathtt{tt}$ is the empty constraint). In general, the term $\mathit{unif}\langle \mathit{final}, \theta\rangle$ stores the computed constraints of a running proof. At the end, a message on either port trueG or falseG will inform the environment about the outcome of the computation. The encoding of clauses and goals is in Figure 6.

An atom A is encoded as a merge rule that substitutes a message $A\langle t, f\rangle$ by the proof of its defining rules (rule DEF). An undefined atom is encoded as a rule that always fails (UNDEF). A conjunction (AND) corresponds to a process that activates a new negotiation containing the atoms $A_i\langle t_i, f_i\rangle$ to be proved and the initial local constraints $\mathit{unif}\langle w, \mathbf{tt}\rangle$. Every sub-proof initiated by $A_i\langle t_i, f_i\rangle$ will notify its termination by using ports t_i (success) and f_i (failure). If all sub-proofs end successfully, the second definition in the negotiation can be fired producing $t\langle\rangle$ (i.e., the signal of the successful proof of the conjunction) and

$$
\begin{array}{lrl}
(\textsc{Def}) & [\![A \leftarrow B_1|C_1,\ldots,A \leftarrow B_n|C_n]\!] = & A\langle t,f\rangle \blacktriangleright [\![\mathbf{choice}(B_1\,|\,C_1,\ldots,B_n|\,C_n)]\!]_{t,f} \\
(\textsc{Undef}) & [\![A]\!] = & A\langle t,f\rangle \blacktriangleright f\langle\rangle \\
(\textsc{And}) & [\![\mathbf{and}(A_1,\ldots,A_n)]\!]_{t,f} = & \mathbf{def}\; and\langle\rangle \triangleright [\mathbf{def}\quad w\langle\rangle \triangleright 0 \\
 & & \qquad \wedge\; \Pi_{i=1}^{n}\; t_i\langle\rangle \triangleright done\langle w\rangle \mid t\langle\rangle \\
 & & \qquad \bigwedge_{i=1}^{n}\; f_i\langle\rangle \triangleright and\langle\rangle \mid failed\langle\rangle \\
 & & \qquad \text{...Rules for handling abortion...} \\
 & & \qquad \mathbf{in}\; \Pi_{i=1}^{n} A_i\langle t_i, f_i\rangle | unif\langle w, \mathtt{tt}\rangle : f\langle\rangle] \\
 & & \mathbf{in}\; and\langle\rangle \\
(\textsc{Choice}) & [\![\mathbf{choice}(A_1|B_1,\ldots,A_n|B_n)]\!]_{t,f} = & \mathbf{def}\; \bigwedge_{i=1}^{n}\; i\langle\rangle \triangleright [\![\mathbf{and}(B_i)]\!]_{t,f} \\
 & & \mathbf{in}\; [\mathbf{def}\quad \Pi_{i=1}^{n}\; f_i\langle\rangle \mid proving\langle\rangle \triangleright abort \\
 & & \qquad \bigwedge_{i=1}^{n}\; t_i\langle\rangle \mid proving\langle\rangle \triangleright i\langle\rangle | chosen\langle\rangle \\
 & & \qquad \text{...Rules for handling abortion...} \\
 & & \quad \mathbf{in}\; \Pi_{i=1}^{n}\; [\![\mathbf{and}(A_i)]\!]_{t_i,f_i} | proving\langle\rangle : f\langle\rangle]
\end{array}
$$

Figure 6. Encoding of AKL committed rules.

$done\langle w\rangle$ that activates the promotion of computed constraints managed by D (omitted for space limitation). Note that $t\langle\rangle$ will be released outside only at commit, after constraints have been promoted. Instead, if a sub-atom fails, all running sub-contracts are aborted and the contract commits by releasing the activation of a new proof.

Rule CHOICE opens a negotiation for proving one of the guards of a multiple choice goal. When a guard is successful the negotiation can commit by releasing the message $i\langle\rangle$, which activates the body B_i of the chosen goal.

If there is an AKL refutation for the goal $\mathcal{G}$ with computed constraints $\mathbf{and}(\mathtt{tt}, \theta)$ then the messages $final\langle\theta\rangle$ and $trueG\langle\rangle$ can be released. On the other hand, $falseG\langle\rangle$ is generated only if $\mathcal{G}$ cannot be proved.

5. Encoding Zero-Safe nets

Zero-safe nets (ZS *nets*) [8] extends Place/Transition Petri nets (PT *nets*) with a mechanism for expressing concurrent transactions. Recently, they have been used in [7] to encode short-running transactions of Biztalk, a commercial workflow management system [16]. ZS nets additionally provides a "dynamic" specification of transactions boundaries supporting multiway transactions, which retain several entry and exit points, and admit a number of participants which is statically unknown. However, ZS nets are not suitable to express some interesting aspects, such as mobility, programmable compensations and nesting.

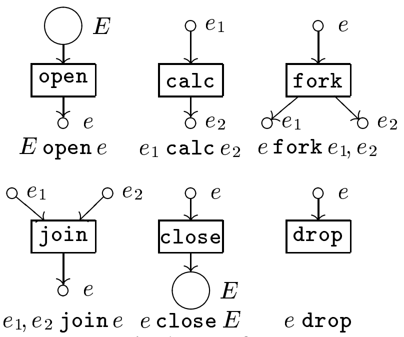
In this section we show that ZS nets can be straightforwardly encoded in cJoin. A distributed implementation of ZS nets in Join has been presented in [7], but there the encoding is complicated by the need of attaching a local transaction manager to each transition.

We recall that, in Petri nets, *places* are repositories of *tokens* and *transitions* fetch and produce tokens. Net configurations, called *markings*, are multisets of tokens. The places of ZS nets are partitioned into ordinary and transactional

$$\text{(FIRING)}\quad \frac{S+Z\,[\rangle\, S'+Z' \in T}{(S+S'', Z+Z'') \rightarrow_T (S'+S'', Z'+Z'')} \qquad \text{(STEP)}\quad \frac{(S_1, Z_1) \rightarrow_T (S_1', Z_1') \quad (S_2, Z_2) \rightarrow_T (S_2', Z_2')}{(S_1+S_2, Z_1+Z_2) \rightarrow_T (S_1'+S_2', Z_1'+Z_2')}$$

$$\text{(CONCATENATION)}\quad \frac{(S_1, Z) \rightarrow_T (S_1', Z'') \quad (S_2, Z'') \rightarrow_T (S_2', Z')}{(S_1+S_2, Z) \rightarrow_T (S_1'+S_2', Z')} \qquad \text{(CLOSE)}\quad \frac{(S, \emptyset) \rightarrow_T (S', \emptyset)}{(S, \emptyset) \Rightarrow_T (S', \emptyset)}$$

Figure 7. Operational semantics of ZS nets (+ denotes multiset union).



a. Basic shapes of ZS nets.

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

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

b. Translation of basic shapes and markings.

Figure 8. Encoding of ZS nets in cJoin.

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. Transitions are written $U[\rangle U'$. 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. 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 two relations $\Rightarrow_T$ and $\rightarrow_T$ in Figure 7. 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 $\rightarrow_T$: 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. CLOSE selects the moves $(S, \emptyset) \Rightarrow_T (S', \emptyset)$, which defines the transactions of the net.

As done in [7], and without loss of generality, we restrict to ZS nets whose transitions have the basic shapes in Figure 8.a, for E any stable place and e, e_1, e_2 any zero places. The translation in Figure 8.b associates a cJoin definition to each basic shape. Places are seen as ports and tokens as messages. Tokens in stable places carry no value, while tokens in zero places carry the identifier of the transaction they belong to. The cJoin process associated to the ZS net (T, S) is **def** $[\![T]\!]$ **in** $[\![S]\!]$, where $[\![T]\!] = \bigwedge_{t \in T} [\![t]\!]$.

A transaction can be opened by firing a transition of the form E $\texttt{open}\ e$. In cJoin, this means opening a new negotiation whose internal state contains the definition of a fresh name z, the message $e\langle z\rangle$, and whose compensation, by default, gives back the stable resources $E\langle\rangle$. The dummy definition $z\langle\rangle \triangleright 0$ is the cJoin way of declaring a fresh identifier z for the transaction. When two transactions are merged by applying e.g. $e_1\langle z_1\rangle|e_2\langle z_2\rangle \blacktriangleright e\langle z_1\rangle$, then z_1 and z_2 become equivalent identifiers for the same larger negotiation. When computing inside a negotiation, each zero token carries one of the possibly many equivalent identifiers for that negotiation (e.g., z_1). If stable messages $E\langle\rangle$ are released inside the negotiation, e.g., by firing e $\texttt{close}$ E, then they are frozen until commit, because the only rules that can fetch 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 (and dummy definitions). Then, the reaction COMMIT can close the negotiations and release all stable tokens to the environment. The following result assures the correctness and completeness of $[\![\_]\!]$.

THEOREM 2 $(S,\emptyset) \Rightarrow^*_T (S',\emptyset)$ *iff* **def** $[\![T]\!]$ **in** $[\![S]\!] \rightarrow^*$ **def** $[\![T]\!]$ **in** $[\![S']\!]$.

The main behavioral difference between the cJoin encoding in Figure 8 and the Join encoding in [7] 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 $[\![E\ \texttt{open}\ e]\!] = E\langle\rangle \triangleright [$**def** $z\langle\rangle \triangleright 0$ **in** $e\langle z\rangle|\mathit{timeout}\langle\rangle : E\langle\rangle]$ and encode the net $N = (T,S)$ as **def** $A \wedge [\![T]\!]$ **in** $[\![S]\!]$.

6. Serializability and Big-Step Semantics

The semantics of cJoin given in Figure 4 allows the cooperation among several negotiations. Nevertheless, we would like to reason about a process by analyzing interacting negotiations independently from the rest of the system. A concurrent execution $T_1 \parallel \ldots \parallel T_n$ of several transactions is said *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 [2]. Serializability is important because it allows to reason about the behavior of a system by considering one transaction at a time. In this section we introduce a syntactical restriction on processes, called *shallowness*, and show that it guarantees serializability.

The idea is to describe multi-party negotiations as abstract transitions that fetch the messages needed to initiate all sub-negotiations separately and produce the processes released at commit or abort. Consequently, serializable negotiations can postpone the activation of each sub-negotiation until all other cooperating sub-negotiations needed to commit can be activated.

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

(SEQ)
$$\frac{\mathcal{D}\vdash\mathcal{P} \twoheadrightarrow \mathcal{D}\vdash\mathcal{P}'' \qquad \mathcal{D}\vdash\mathcal{P}'' \twoheadrightarrow \mathcal{D}\vdash\mathcal{P}'}{\mathcal{D}\vdash\mathcal{P} \twoheadrightarrow \mathcal{D}\vdash\mathcal{P}'}$$

(GLOBAL FIRING)
$$\mathcal{D} \wedge J \triangleright \mathcal{P} \vdash J\sigma \twoheadrightarrow \mathcal{D} \wedge J \triangleright \mathcal{P} \vdash \mathcal{P}\sigma$$

(LOCAL FIRING)
$$\frac{\widetilde{\mathcal{B}}\vdash\mathcal{S} \twoheadrightarrow \widetilde{\mathcal{B}}\vdash\mathcal{S}'}{\mathcal{D} \wedge \mathcal{B}\vdash[\mathcal{S} : \mathcal{Q}] \twoheadrightarrow \mathcal{D} \wedge \mathcal{B}\vdash[\mathcal{S}' : \mathcal{Q}]}$$

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

(LOCAL COMMIT)
$$\mathcal{D}\vdash[M \mid \mathcal{D}'\vdash 0 : \mathcal{S}] \twoheadrightarrow \mathcal{D}\vdash M$$

(ABORT)
$$\mathcal{D}\vdash[abort|\mathcal{P} : \mathcal{S}] \twoheadrightarrow \mathcal{D}\vdash\mathcal{S}$$

(IDLE)
$$\mathcal{D}\vdash\mathcal{P} \twoheadrightarrow \mathcal{D}\vdash\mathcal{P}$$

Figure 9. Big-step semantics of cJoin.

DEFINITION 3 (SHALLOWNESS) *The* nesting $nest(P)$ *of* P *is defined by:*

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

P *is* shallow *if any basic definition* D *in* P *satisfies one of the two conditions:*

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

We refer definitions in shallow processes as *shallow definitions*. Moreover, we call a process P *stable* iff P is shallow and $nest(P) = 0$. The shallow property imposes a discipline for activating negotiations. In particular, condition 1 assures that the firing of an ordinary rule increases the height of the nesting by at most one level (i.e., a definition produces either a stable process or one negotiation without nested sub-contracts). Condition 2 forbids the creation of sub-negotiations while merging. The absence of condition 2 would prevent the possibility of postponing the activation of some negotiations until all cooperating negotiations can be activated. Shallowness forbids rules such as $J_1 \triangleright P \mid [P_1 : Q]$ and $J_2 \triangleright [[P_1 : Q_1] : Q]$, which however can be encoded as shallow definitions by using new local ports, e.g. as $D_1 \equiv J_1 \triangleright x\langle\rangle | P \ \wedge\ x\langle\rangle \triangleright [P_1 : Q]$ and $D_2 \equiv J_2 \triangleright [\textbf{def}\ x\langle\rangle \triangleright [P_1 : Q_1]\ \textbf{in}\ x\langle\rangle : Q]$, respectively. Moreover **def** D_2 **in** $J_2\sigma$ reduces in two steps 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 stable process, and $\mathcal{B}$ a shallow definition containing just merge rules. We abbreviate **def** $\mathcal{D}$ **in** $\mathcal{P}$ as $\mathcal{D} \vdash \mathcal{P}$, and $\vdash \mathcal{P}$ as $\mathcal{P}$. Terms are considered up-to structural equivalence generated by closure w.r.t the equations for the associativity and commutativity of $|$ and $\wedge$, 0 the unit for $|$, and

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

We characterize serializability by the big-step reduction relation between shallow processes presented in Figure 9.

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 represents suitable sub-negotiations 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 negotiation are those relating stable processes, i.e., $\mathcal{S}$ and $\mathcal{S}'$. A negotiation has available, in addition to its own definitions, the merge definitions introduced by its parent. In fact, a merge definition applied on a single contract behaves as an ordinary rule but defined in a global scope. The operator $\widetilde{\ }$ transforms merge definitions in ordinary ones: $\widetilde{J \blacktriangleright \mathcal{P}} \equiv J \triangleright \mathcal{P}$ and $\widetilde{\mathcal{B} \wedge \mathcal{B}'} \equiv \widetilde{\mathcal{B}} \wedge \widetilde{\mathcal{B}'}$. If the rule STR-MOVE is also considered for cJoin, then the premise of LOCAL FIRING must be $\widetilde{\mathcal{B}} \wedge \mathcal{B} \vdash \mathcal{S} \twoheadrightarrow \widetilde{\mathcal{B}} \wedge \mathcal{B} \vdash \mathcal{S}'$.

Rules LOCAL COMMIT and ABORT handle the termination of a negotiation, whereas MERGE describes the interaction among sibling negotiations. This time, negotiations can be joined only if they do not contain running contracts.

The big-step relation enforces serializability. In fact, the completed negotiations at a particular level become ordinary transitions at the upper level and all interacting transactions can be analyzed independently from the rest of the system. The following result states the correspondence between both semantics for shallow processes, proving that shallow processes are serializable.

THEOREM 4 *Let $\mathcal{S}, \mathcal{S}'$ be stable processes. Then $\mathcal{S} \rightarrow^* \mathcal{S}'$ iff $\vdash \mathcal{S}' \twoheadrightarrow \vdash \mathcal{S}'$.*

7. Concluding remarks

We have proposed cJoin as a linguistic extension of Join with natural primitives for modeling nested negotiations. The expressiveness of cJoin has been demonstrated by means of an informal discussion about the satisfaction of the general requirements enumerated in the Introduction and three sample applications (trip booking and the encodings of AKL and ZS nets). Additionally, we have defined a syntactical restriction of processes that assures serializability.

Unlike *workflow systems*, cJoin does not fix a set of constructors to describe dependencies in the style of ConTracts [15]. The actual dependencies of a negotiation are known at execution time and are consequence of its interaction with other contracts. This feature distinguishes cJoin from [4], where processes running as transactions can interact freely with the environment. On the other hand, cJoin is aimed at providing a way to model multi-party transactions by describing their interacting agents and not their global structure, such as [6, 9]. Nevertheless, our language does not provide default mechanisms for undoing pre-committed activities of aborted transactions, differently from [9]. We leave as a future work the comparison with other calculi that models ACID transactions [17], long-running negotiations [4, 11], and exception handling [14].

Negotiations in cJoin have the flavour of multiway transactions in ZS nets, where participants are not statically fixed. We plan to reuse or to extend the D2PC proposed in [7] to have a full encoding of cJoin in Join itself. This would allow us to extend implementations of Join, such as Jocaml or Polyphonic C$^\sharp$, by providing primitives for handling distributed negotiations. As a preliminary result, the encoding for the subcalculus of *flat* processes is in [5].

Acknowledgments. We thank Nick Benton, Luca Cardelli, Cédric Fournet and Cosimo Laneve with whom we discussed preliminary versions of cJoin.

References

[1] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. *Proc. of ECOOP 2002*, *LNCS* 2374, pp. 415–440. Springer Verlag, 2002.

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

[3] G. Berry and G. Boudol. The chemical abstract machine. *TCS*, 96(1):217–248, 1992.

[4] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. *Proc. of FMOODS'03*, *LNCS* 2884, pp. 194–208. Springer Verlag, 2003.

[5] R. Bruni, H. Melgratti, and U. Montanari. Flat Committed Join in Join. *Proc. of COMETA 2003*, *ENTCS*. To appear.

[6] BPEL Specification, May 2003. `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`.

[7] R. Bruni, C. Laneve, and U. Montanari. Orchestrating transactions in join calculus. *Proc. of CONCUR 2002*, *LNCS* 2421, pp. 321–336. Springer Verlag, 2002.

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

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

[10] S. Conchon and F. Le Fessant. Jocaml: Mobile agents for Objective-Caml. *Proc. of ASA'99/MA'99*, 1999.

[11] D. Duggan. Abstractions for Fault-Tolerant Global Computing. *TCS*. To appear.

[12] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the Join calculus. *Proc. of POPL'96*, pp. 372–385. ACM Press, 1996.

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

[14] M. Mazzara and R. Lucchi. A framework for generic error handling in business processes. *Proc. of WS-FM'04*, *ENTCS*. To appear.

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

[16] U. Roxburgh. Biztalk orchestration: transactions, exceptions, and debugging, 2001. `http://msdn.microsoft.com/library/en-us/dnbiz/html/btsorch.asp`.

[17] J.Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A Semantic Framework for Designer Transactions. *Proc. of ESOP'04*, *LNCS* 2986, pp. 249–263. Springer Verlag, 2004.