Orchestrating Transactions in Join Calculus*

Roberto Bruni[1], Cosimo Laneve[2], and Ugo Montanari[1]

[1] Dipartimento di Informatica, Università di Pisa, Italia,
WWW home pages: http://www.di.unipi.it/~{bruni,ugo}
{bruni,ugo}@di.unipi.it
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna, Italia,
WWW home page: http://www.cs.unibo.it/~laneve
laneve@cs.unibo.it

Abstract We discuss the principles of distributed transactions, then we define an operational model which meets the basic requirements and we give a prototyping implementation for it in join-calculus. Our model: (1) extends BizTalk with multiway transactions; (2) exploits an original algorithm, for distributed commit; (3) can deal with dynamically changing communication topology; (4) is almost language-independent. In fact, the model is based on a two-level classification of resources, which should be easily conveyed to distributed calculi and languages, providing them with a uniform transactional mechanism.

1 Introduction

Global computing requires data integration and process cooperation within and across sites and organizations. The design and execution of distributed applications for global computing thus heavily rely on *process orchestration services*, which are usually demanded to the coordination layer of the system. This is the case, for instance, for transaction managers in BizTalk and Javaspaces platforms, which handle distributed decisions in asynchronous environments (e.g. the web or WAN's).

In the orchestration of services, transactional activities must be modeled also when distributed components are designed and implemented separately (e.g. in e-commerce or on-line auction systems). To this purpose, the ordinary formulation of transaction in databases theory (see Table 1) is inadequate because:

1. distributed transactions may have several entry/exit points in parallel, rather than just one (*multiparty* or *multiway transactions*);
2. data integrity should be specialized for distributed, heterogeneous systems, where there is no explicit notion of data;
3. the communication topology of processes participating to a transaction may change dynamically. This requires a commit protocol among a number of participants which cannot be statically determined.

* Research supported by the MSR Cambridge Project *Network-Aware Programming and Interoperability*.

A *transaction* consists of a collection of actions that transforms a system from one consistent state to another and adheres to the so-called ACID properties:

Atomicity, A transaction's changes to the state are atomic: either all happen or none happen.

Consistency, A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation, Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.

Durability, Once a transaction completes successfully (commits), its changes to the state survive failures.

Table 1. Data base transactions as defined in [13].

A standard example in business-to-business e-commerce is *dynamic pricing*, e.g. the reverse auction. In a reverse auction, a *buyer* issues a request for offers to the *market place* where several *sellers* may propose their bids. The number of sellers is not know *a priori*. Later on, the buyer chooses the optimal bid, emits a payment for the seller, and removes its request from the market place. The whole negotiation must look like an atomic activity. For instance, if the negotiation aborts, the offer must be neglected, the payment refused, and the initial situation must be restored.

Microsoft® BizTalk [17] is a system explicitly designed for process orchestration. 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.

In BizTalk, transactions can be *short*, *long* or *timed*. Transactions consist of series of actions, with long and timed ones possibly grouping any combination of actions and transactions (up to two nesting levels). Short transactions retain automatic recovery, which implements database transactions as defined in Table 1. On-failure compensation in the other types of transactions is programmed by the users, and may be defined in a compositional way.

We note that BizTalk boxes correspond to a two-level classification of states, *observable* (i.e., outside the box) and *hidden* (i.e., inside the box). This partition also pervades another formalism, which has been recently defined to model transactional aspects in concurrent systems, the *zero-safe Petri nets* (ZS nets) [5,7]. BizTalk flowcharts may be easily encoded into ZS nets, while the converse encoding seems not possible. The reason is that BizTalk boxes provide a "static" specification of transactions boundaries, while the state partitioning gives a "dynamic" notion of them, which depends on the runtime behaviour (e.g. which processes communicated, which data were exchanged). This remark qualifies ZS nets as a possible formal model for orchestration services.

Building on the transactional mechanism of ZS nets we propose a fairly general notion of distributed transaction that can be easily adapted to BizTalk and other orchestration systems. This mechanism also supports multiway transactions, which retain several entry and exit points, and admit a number of participants which is statically unknown.

The feasibility of our model is demonstrated by implementing ZS nets in the join-calculus [10,11]. This formalism has been chosen for three reasons:

1. join calculus is a well-known calculus, with an assessed formal theory;
2. a distributed prototype implementation has been developed for the join calculus;
3. several comparisons between join calculus and Petri nets have been studied (see e.g. the type systems for join calculus in [8] that characterize a hierarchy of net models), making it easier to establish a formal correspondence between ZS nets and their encoding in join.

In the following, we shall work under the usual assumption that communications are reliable, i.e. that messages cannot get lost.

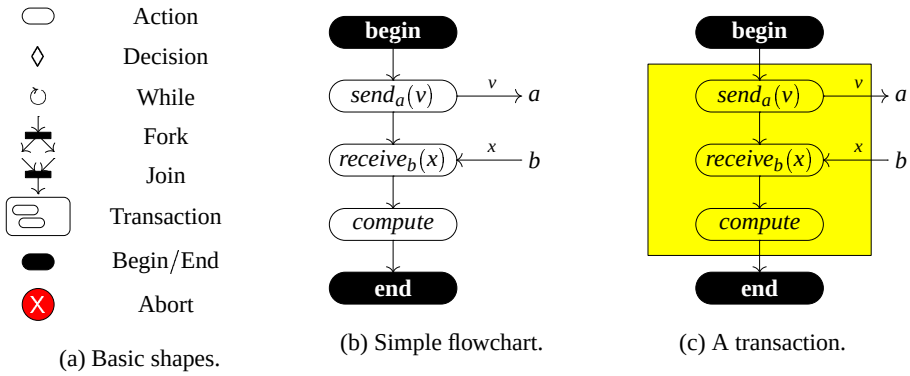
We discuss an implementation of ZS nets, which we call *distributed*, as opposed to the *centralized* approach that relies on a unique transaction coordinator (like in BizTalk's design). The distributed implementation is the core result of the paper. It exploits the whole expressive power of the join-calculus for the dynamic definition of local transaction controllers. Moreover, it exploits a *distributed* 2PC *protocol* (a variant of the *decentralized* 2PC protocol [1]) that, up-to our knowledge, is novel to this contribution.

In the decentralized 2PC, every process participating to a transaction is aware of the other processes involved in the same transaction. When a process is ready to commit, it sends a message to all the companion processes (first phase). At the same time it collects all the commit messages coming from the companion processes. When all the messages have been received, the process commits (second phase). Alternatively, if a process fails, it will send a failure message to every companion process, and roll back to its own initial state (in this paper we implement the ideal compensation mechanism of Table 1). Every process receiving a failure message also fails.

In the distributed 2PC protocol, the number of participants and their names are not statically fixed, and processes that are ready to commit know only the participants with whom have directly cooperated. For this reason, in the distributed 2PC protocol, the role of the coordinator is played by all participants. In particular, every participant builds, in linear time, its own transitive closure of the processes cooperating in the transaction. When the transitive closure is stable, the participant commits. Alternatively, the participant waits for a failure message, and restores the initial state when such message arrives.

Synopsis. In Sections 2 and 3, we give a brief account of the transactional mechanism of BizTalk, and discuss its modeling via ZS nets, where also multiway transactions can be designed. In Section 4, we present the distributed encoding of ZS nets in join calculus. In Section 5 we discuss suitable extensions of the distributed encoding. Finally, some remarks and conclusions are drawn in Section 6.

Due to space limitation, we refer the interested reader to the forthcoming technical report [3] for the proofs of main results and for the comparison of the distributed implementation of ZS nets against the centralized modeling with a unique transaction coordinator. Noticeably, while the centralized encoding can be carried out in a "flat" version of the join-calculus, where definitions are not nested (i.e. in high-level nets, according to [8]), the distributed encoding shows the expressiveness of the full language.



(a) Basic shapes. (b) Simple flowchart. (c) A transaction.

Figure 1. Flowcharts in BizTalk Orchestration Visual Designer.

2 Coordinating agents in Biztalk

Microsoft® *BizTalk Orchestration Visual Designer* provides a graphical environment for building business processes, which integrates three different views: the *flowchart* view, the *implementation* view, and the *data flow* view. We focus on flowcharts, [1] which are realized by connecting several basic shapes (see Figure 1(a)).

In Figure 1(b), we illustrate a simple BizTalk process sending the value v on channel a, then inputting a value on the variable x from channel b, and, later on, performing some calculation. A transaction is defined by boxing series of operations like in Figure 1(c), meaning that either all the operations succeed or the system is rolled-back to the initial state, e.g. the emission of the message v on channel a and the input of x from b depend on the successful termination of the "*compute*" operation.

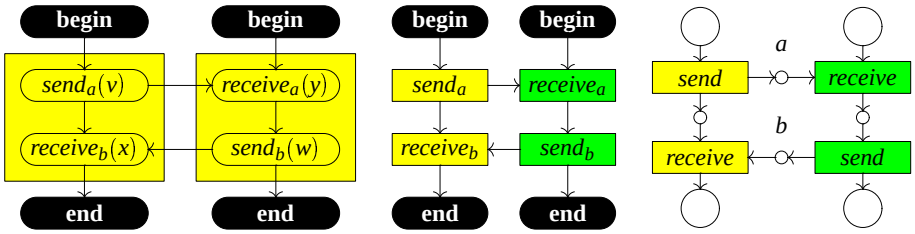
Figure 2(a) illustrates a process which implements a *rendez-vous* between two transactions: a commit is performed provided both the transactions are ready to commit. The same diagram can also be used to model a remote procedure call ($send_a(v)$ plays the role of the invocation and $send_b(w)$ plays the role of the transmission of the result).

Transaction commits are demanded to the *Distributed Transaction Coordinator* (DTC) running on the server machine and coordinating a two-phase commit protocol (2PC). Roughly, the 2PC works as follows [13]:

1. the DTC sends a vote request to all participants;
2. upon vote request, each participant either votes **no** and aborts, or votes **yes**;
3. the DTC collects all votes. If all votes are **yes**, then the DTC sends **commit** to all participants. Otherwise, the DTC sends **abort** to all participants that voted **yes**;
4. each participant that voted **yes** waits for DTC response and decides accordingly.

The correctness of the 2PC algorithm strongly relies on the knowledge by DTC of every participant to the transaction. To this aim, every transaction is paired with suitable components—the *COM+ components*—that record participants to the DTC. Actually, the process in Figure 2(a) shows up a further issue: it is required DTC become

[1] Our actual understanding of BizTalk's transactions is mostly based on documents available on the web through MSDN Library `http://msdn.microsoft.com/library` [15,17].



(a) A rendez-vous in BizTalk. (b) First step. (c) Second step.

Figure 2. Alternative drawings of BizTalk transactions.

aware that the two transactions must commit together. This can be achieved by making communication *transactional*, which amounts to employ COM+ components (that, at run-time, report to DTC that the two transactions communicated). For long and timed transactions, users' compensation code can be given for rolling back in case of failure [14].

In the following we assume that long and timed transactions possess an *ideal compensation code* which implements the properties in Table 1. For instance, the process of Figure 2(a) either rendez-vous or nothing occurs. We also assume that transactions are not nested. In BizTalk, nested transactions allow to define a common programming pattern: a long (inter-companies) transaction grouping a sequence of short (intra-company) ones. The two assumptions above simplify the theory: we discuss the general case in Section 5.

2.1 BizTalk without boxes

BizTalk flowcharts may be converted into a standard formalism for concurrent computations, the *Petri nets*; actually into a transactional flavour of them, the *zero-safe nets* [5]. We split this conversion into two steps:

First step: coloring basic shapes. In BizTalk, transactions are depicted by grouping basic shapes into (shadowed) boxes (Figures 1(c) and 2(a)). Since short transactions can be neither intersected nor nested, instead of using boxes, one may color basic shapes therein. Shapes included in different boxes have different colors; shapes outside boxes are white. Figure 2(b) illustrates this alternative drawing for Figure 2(a).

Second step: adding states. We insert circles in the pictures. Circles represent states, or *places* in Petri net terminology, and are drawn in the middle of every link. There are two kinds of circles: large ones (*stable places*) and small ones (*transactional places*), respectively for states *outside* transactions and states *within* some transaction. In case of a transition link, if the link connects two shapes of the same color (different from white) then the circle is small, otherwise it is large. In case of a communication, if the action is transactional then the circle is small, otherwise it is large. Begin and end shapes are regarded as stable places. This second step is shown in Figure 2(c) for the rendez-vous flowchart in Figure 2(a).

$$\text{(FIRING)}\quad \frac{S+Z \triangleright S'+Z' \in T}{(S+S'',Z+Z'') \to_T (S'+S'',Z'+Z'')} \qquad \text{(STEP)}\quad \frac{(S_1,Z_1) \to_T (S_1',Z_1') \;\; (S_2,Z_2) \to_T (S_2',Z_2')}{(S_1+S_2,Z_1+Z_2) \to_T (S_1'+S_2',Z_1'+Z_2')}$$

$$\text{(CONCATENATION)}\quad \frac{(S_1,Z) \to_T (S_1',Z'') \;\; (S_2,Z'') \to_T (S_2',Z')}{(S_1+S_2,Z) \to_T (S_1'+S_2',Z')} \qquad \text{(COMMIT)}\quad \frac{(S,\varnothing) \to_T (S',\varnothing)}{(S,\varnothing) \Rightarrow_T (S',\varnothing)}$$

Table 2. Operational semantics of ZS nets ($+$ denotes multiset union).

The semantics of the net in Figure 2(c) departs from the standard one for Petri nets [16] because of transactions. In Petri nets, places are repositories of resources, called *tokens*, and transitions fetch and produce tokens.[2] Zero-safe nets have been introduced in [4] as a suitable transactional flavour of Petri nets: places are partitioned into ordinary and transactional ones (called *stable* and *zero*, respectively). Transition colors, introduced at the first step of the transformation are no longer relevant, because transaction orchestration is entirely demanded to zero places. Net configurations are called *markings* and can be viewed as pairs (S,Z)—corresponding to the multiset $S+Z$—where S is the multiset of stable resources and Z is the multiset of transactional resources. The key point is that stable tokens produced during a transaction are made available only at commit time, when all zero tokens have been consumed.

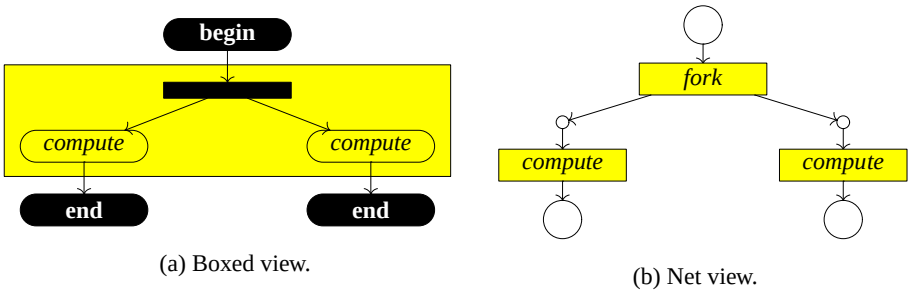
The operational semantics of ZS nets is defined by the two relations $\Rightarrow$ and $\to$ in Table 2, indexed by the set of transitions T. Transitions have the form $U \triangleright U'$, with U and U' multisets of stable and zero places. Rules FIRING and STEP are the ordinary ones for Petri nets, for the execution of one/many transition(s). However, sequences of steps differ from the ordinary transitive closure of $\to$: 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 COMMIT can be applied. We present ZS nets as pairs (T,S) where S is the initial marking, and denote $(S,\varnothing) \Rightarrow_T (S',\varnothing)$ by $S \Rightarrow_T S'$.

The moves $S \Rightarrow_T S'$ 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 $S \Rightarrow_T S'$ can be itself the parallel composition of several transactions (by rule STEP). We refer the interested reader to [5] for the characterization of the set of minimal transactions—in the sense that they cannot be decomposed into smaller transactions—that depends on the "token philosophy" under consideration: either *collective* or *individual* according to the classification in [12].

3 BizTalk with multiway transactions

BizTalk transactions possess exactly one entry and one exit point, meaning that *intra* thread coordination is mostly delegated to programmers (in contrast with *inter* thread

[2] For simplicity, here we leave aside that tokens can carry values and transitions can have parameters, but our results in Section 4 can be smoothly lifted to the valued case.



(a) Boxed view. (b) Net view.

Figure 3. A transaction with two exit points.

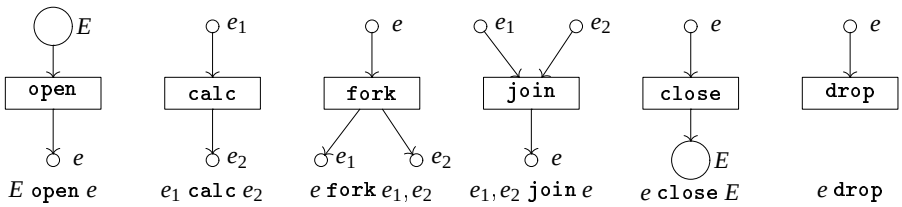


Figure 4. Basic shapes of extended BizTalk.

coordination due to message exchanges, which is demanded to the DTC process). The analogy between BizTalk flowgraphs and ZS nets, detailed in the previous section, suggests that transactions can be generalized by allowing multiple entry and exit points—yielding *multiway transactions*. In Figure 3, we picture a flowgraph with two exit points, as it should appear in BizTalk and its counterpart in ZS nets.

Up-to our understanding, BizTalk limitation to single-threaded transaction shapes is motivated by implementation issues, rather than theoretical motivations. Indeed, rendezvous are essentially multiway transactions and can be programmed in BizTalk exploiting suitable compensation code. Zero-safe nets deal uniformly with multiway and single-threaded transactions, and hence the distinction is transparent to our encoding.

4 Encoding multiway transactions in join

In this section we present a *distributed* account of the 2PC protocol for Biztalk with multiway transactions. The distributed algorithm makes every participant acts as local coordinator. The implementation is provided with a distributed machine, the *distributed join calculus* [10,11]. Actually, since BizTalk flowgraphs may be converted into ZS nets, we find convenient to implement the latter ones of whom we have a neat understanding.

Without loss of generality, we restrict to nets made with the basic shapes in Figure 4 (which are as expressive as the general nets [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). We note that the ZS net in Figure 2(c) is not a basic ZS net. It is immediate to rewrite Figure 2(c) by adding four dummy *compute* shapes and considering *send* and

Syntax	$P \stackrel{def}{=} 0 \mid x\langle\tilde{u}\rangle \mid P \mid P \mid \mathsf{def}\ D\ \mathsf{in}\ P$	$D \stackrel{def}{=} J \triangleright P \mid D \wedge D$	$J \stackrel{def}{=} x\langle\tilde{u}\rangle \mid J \mid J$
Structural moves	$P \mid Q \equiv P, Q$	$\mathsf{def}\ D\ \mathsf{in}\ P \equiv D\sigma, P\sigma$ (σ renames defined names in D with fresh names)	$D \wedge D' \equiv D, D'$
Reduction	$J \triangleright P, J\rho \ \rightarrow\ J \triangleright P, P\rho$	(ρ only renames received names in J)	

Table 3. Syntax and semantics of the join calculus.

receive shapes as *fork* and *join*, respectively. We remark that for ordinary Petri nets (i.e. without zero places) a straightforward encoding in join is possible.

4.1 Brief intro to the join calculus

The join calculus [10] relies on a countable set of names ranged over by x, y, u, v, …; tuples of names are written $\tilde{u}$. Table 3 collects the syntax and the semantics of the join calculus. The syntax includes processes P, definitions D, and join-patterns J. A process P can be the inert process 0, a message $x\langle\tilde{u}\rangle$, a parallel composition of processes, or a defining process $\mathsf{def}\ D\ \mathsf{in}\ P$. A definition D is a conjunction of reaction rules $J \triangleright P$ that associate join-patterns J to *guarded processes* P; the intended meaning is that, whenever messages match the pattern J, these messages can be replaced with a copy of the guarded process P where the content of the messages substitute the formal parameters. The only binders of the calculus are join-patterns, but the scope of names appearing in a join-pattern depends on their position: the scope of formal parameters—called *received names*—is the guarded process; the scope of *defined names*—i.e., names carrying messages—is the main process of the definition and, recursively, all guarded processes of the definition.

The semantics of the join calculus relies on the *reflexive chemical abstract machine* model [2]. In this model a solution is a multiset of active definitions and processes (separated by comma). Dynamically new definitions may become active. Moves are distinguished between *structural* $\equiv$, which heat or cool processes, and *reductions* $\rightarrow$, which are the basic computational steps (disjoint reductions can be executed in parallel). In Table 3, the rules only address the part of the solution which actually moves.

The *distributed join calculus* is a distributed implementation of the join calculus, with notions of location, migration of processes, failure, and failure recovery [11]. In order to simplify the notation, the encoding in the sequel will be written in join calculus. Yet, the distributed versions shall be discussed informally.

4.2 Distributed 2PC

In the following encoding, we extend the join calculus with the data type *SET*, for finite sets and the standard operations of emptyset $\varnothing$, union $\cup$, and difference $\setminus$. Join patterns are extended with pattern matching on sets. We also use the operation $\mathsf{release}$ which takes a set of channels and sends an (empty) message on every channel in the set. We address channels which encode stable and zero tokens as stable and zero channels, respectively. Intuitively, a message $x\langle\tilde{u}\rangle$ models a token with value $\tilde{u}$ in the place x.

$$
\begin{aligned}
[\![E \ \texttt{open}\ e]\!] &= E\langle\rangle \triangleright \mathsf{def}\ \mathsf{D}\ \mathsf{in}\ e\langle put, \{lock\}\rangle \,|\, state\langle\{E\}\rangle \\
[\![e_1 \ \texttt{calc}\ e_2]\!] &= e_1\langle p,\ell\rangle \triangleright e_2\langle p,\ell\rangle \\
[\![e \ \texttt{fork}\ e_1, e_2]\!] &= e\langle p,\ell\rangle \triangleright \mathsf{def}\ \mathsf{D}\ \mathsf{in}\ e_1\langle p, \ell\cup\{lock\}\rangle \,|\, e_2\langle put, \ell\cup\{lock\}\rangle \,|\, state\langle\varnothing\rangle \\
[\![e_1, e_2 \ \texttt{join}\ e]\!] &= e_1\langle p_1,\ell_1\rangle \,|\, e_2\langle p_2,\ell_2\rangle \ \triangleright e\langle p_1, \ell_1\cup\ell_2\rangle \,|\, p_2\langle \ell_2\cup\ell_1, \varnothing\rangle \\
[\![e \ \texttt{close}\ E]\!] &= e\langle p,\ell\rangle \triangleright p\langle \ell, \{E\}\rangle \\
[\![e \ \texttt{drop}]\!] &= e\langle p,\ell\rangle \triangleright p\langle \ell, \varnothing\rangle
\end{aligned}
$$

where D is the following definition:

$$
\begin{aligned}
\mathsf{D} \stackrel{def}{=}\ & state\langle H\rangle \,|\, put\langle \ell, F\rangle \triangleright commit\langle \ell\setminus\{lock\}, \ell, \{lock\}, F, H\rangle \\
\wedge\ & state\langle H\rangle \triangleright failed\langle\rangle \,|\, \mathsf{release}\langle H\rangle \\
\wedge\ & commit\langle \{l\}\cup\ell, \ell', \ell'', F, H\rangle \triangleright commit\langle \ell, \ell', \ell'', F, H\rangle \,|\, l\langle \ell', lock, fail\rangle \\
\wedge\ & commit\langle \ell, \ell', \ell'', F, H\rangle \,|\, lock\langle \ell''', l, f\rangle \triangleright commit\langle \ell\cup(\ell'''\setminus\ell'), \ell'\cup\ell''', \ell''\cup\{l\}, F, H\rangle \\
\wedge\ & commit\langle \varnothing, \ell, \ell, F, H\rangle \triangleright \mathsf{release}\langle F\rangle \\
\wedge\ & commit\langle \ell, \ell', \ell'', F, H\rangle \,|\, fail\langle\rangle \triangleright failed\langle\rangle \,|\, \mathsf{release}\langle H\rangle \\
\wedge\ & failed\langle\rangle \,|\, put\langle \ell, F\rangle \triangleright failed\langle\rangle \\
\wedge\ & failed\langle\rangle \,|\, lock\langle \ell, l, f\rangle \triangleright failed\langle\rangle \,|\, f\langle\rangle \\
\wedge\ & failed\langle\rangle \,|\, fail\langle\rangle \triangleright failed\langle\rangle
\end{aligned}
$$

Figure 5. The encoding in the join calculus.

Definition 1 (Reflexive encoding). *The join calculus process of a* ZS *net* (T,S) *is* $\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ \{[S]\}$, *where* $[\![T]\!] = \wedge_{t\in T}[\![t]\!]$ *with* $[\![t]\!]$ *defined in Figure 5, and* $\{[S]\}$ *defined inductively as follows:* $\{[\varnothing]\} = 0$ *and* $\{[E+S]\} = E\langle\rangle \,|\, \{[S]\}$.

A new thread is started either when (1) *a transition* $\texttt{open}(E,e)$ *occurs*, or when (2) *a thread forks*. This is evidenced by controller definitions D in the join calculus agents $[\![E \ \texttt{open}\ e]\!]$ and $[\![e \ \texttt{fork}\ e_1, e_2]\!]$. Every thread has at most one active zero token at any time. A thread may terminate when its associated zero token has been consumed, i.e. when a `join`, or `close`, or `drop` is executed on that token. Zero channels carry two informations: (a) a channel name to notify the thread completion; (b) the *synchronization set* of the thread, namely a set of pointers to companion threads which belong to the same transaction. This synchronization set is built by recording thread pointers every time a fork or a join occurs (see the encodings of `fork` and `join`).

We comment the definition D in Figure 5. The channel *state* records the stable channels (if any) that must be released in case of failure. The *state* message is created when the thread starts and it is consumed when the thread terminates either with success or with failure. Successful termination occurs when the active zero token of the thread has been consumed—a message $put\langle \ell, F\rangle$ has been emitted and

$$state\langle H\rangle \,|\, put\langle \ell, F\rangle \triangleright commit\langle \ell\setminus\{lock\}, \ell, \{lock\}, F, H\rangle$$

has been executed. In this case a *commit* message is produced. We assume that a thread cannot (nondeterministically) decide to fail anymore once *commit* has been emitted. This is a standard assumption: when the thread reaches this state, it logs the relevant infos. However, if another thread involved in the same transaction fails, the *commit* state must still be able to handle the failure.

A thread in $state\langle H\rangle$ may fail. This is modelled by a nondeterministic rule yielding the internal state $failed\langle\rangle$. (In real implementations, failures are e.g. triggered by time-

outs of the communication with a specific other participant or of the whole transaction—see timed transactions in BizTalk.) In the state $failed\langle\rangle$, any thread in the same transaction is informed about the failure.

The commit protocol is a distributed variant of the 2PC, where the role of the coordinator is played by all participants. Up-to our knowledge, this algorithm is original to our contribution. It differs from the *decentralized* 2PC [1] because the number of participants and their names are not statically fixed. The protocol is detailed below.

The commit state is modeled by the *commit* message, carrying values $\langle \ell, \ell', \ell'', H, F\rangle$:

1. ℓ records the set of threads to which the signal of successful completion (i.e. the analogous of the **yes** vote) must still be sent;
2. ℓ' stores the synchronization set of the thread. The successful completion signal must be sent to every item of ℓ'. During the protocol, this set is augmented with the synchronization sets of the threads participating to the same transaction, until when it is *transitively closed*. Initially, ℓ' contains the pointers to the threads with whom there has been a direct interaction (a fork or a join), together with a pointer to itself;
3. ℓ'' records the participants who have already sent the success signal to the thread. Initially, ℓ'' contains the pointer to itself, thus avoiding self-addressed messages;
4. F and H store the stable channels to be released in case of successful and unsuccessful completion, respectively. In our case, they may be empty sets or singletons.

The *distributed* 2PC is based on the following steps performed by every thread:

1. **first phase**. The thread sends a request message to every thread in its own synchronization set. This task is performed by the rule

$$commit\langle \{l\} \cup \ell, \ell', \ell'', F, H\rangle \ \triangleright\ commit\langle \ell, \ell', \ell'', F, H\rangle \mid l\langle \ell', lock, fail\rangle.$$

   The request message carries the synchronization set of the thread, together with the names for signaling the successful or unsuccessful completion.
2. **second phase**. The thread collects the messages sent by other threads and updates its own synchronization set. The name used to carry these information is *lock*, and the rule collecting the synchronization sets is:

$$commit\langle \ell, \ell', \ell'', F, H\rangle \mid lock\langle \ell''', l, f\rangle \ \triangleright\ commit\langle \ell \cup (\ell''' \setminus \ell'), \ell' \cup \ell''', \ell'' \cup \{l\}, F, H\rangle.$$

   A request message will be also sent to the new items in the synchronization set. Namely, the first and the second phases work in parallel.
3. When the synchronization set is transitively closed, namely when the union of synchronization sets of threads in ℓ' is equal to the set ℓ'' of successful termination messages, the commit protocol for the local thread terminates and the messages on channels in F may be emitted.
4. In case of a failure, the thread transits in the state *failed*, releases the stable channels in H, and replies to every *lock* message with a failure answer:

$$\begin{aligned} commit\langle \ell, \ell', \ell'', F, H\rangle \mid fail\langle\rangle \ &\triangleright\ failed\langle\rangle \mid \mathsf{release}\langle H\rangle \\ failed\langle\rangle \mid lock\langle \ell, l, f\rangle \ &\triangleright\ failed\langle\rangle \mid f\langle\rangle \end{aligned}$$

The proof of correctness of the distributed 2PC is split in two steps: (part 1) we show that if all coordinators are ready to commit, then all frozen resources will be released (assuming fairness); (part 2) we strengthen the result by dealing with failures of transactions.

In the following theorems let σ_i, $i \in \mathbb{N}$, be the renaming that indexes with i all the defined names in D. We write $A\langle\rangle$, when A is an empty set or a singleton $\{a\}$: In the former case it means $\mathsf{0}$, in the latter case it represents $a\langle\rangle$. Moreover, we let a *symmetric lock covering* be a finite family $\{\ell_i\}_{i\in I}$ such that $\ell_i \subseteq \{lock_j \mid j \in I\}$, with $lock_j \in \ell_i$ if and only if $lock_i \in \ell_j$ for all $i, j \in I$.

Theorem 1 (Correctness of the distributed 2PC, part 1). *Let $P = |_{i\in I} commit_i\langle \ell_i, \ell_i \cup \{lock_i\}, \{lock_i\}, F_i, H_i\rangle$, where $\{\ell_i\}_{i\in I}$ is a symmetric lock covering, and let n be the cardinality of I. The process* $\mathsf{def}\ \bigwedge_{i\in I} \mathsf{D}\sigma_i\ \mathsf{in}\ P$ *is strongly confluent, in the sense that it always converges after a finite number of steps bound by $O(n^2)$ to the configuration*

$$\mathsf{def} \bigwedge_{i\in I} \mathsf{D}\sigma_i \ \mathsf{in} \left(\mathop{|}_{i\in I} F_i\langle\rangle \right) .$$

Though the number of steps is $O(n^2)$, they are concurrently executed by n threads, thus the number of steps for each thread is linear in n. More precisely, in the worst case where all threads participate to the same transaction, each thread must (asynchronously) send and receive exactly n messages to commit.

The second theorem states that, when several coordinators fail, then all the participants to their transactions also fail. To determine the coordinators which participate to a transaction, we compute the transitive closure of the synchronization sets.

Theorem 2 (Correctness of the distributed 2PC, part 2). *Let $\{\ell_i\}_{i\in I}$ be a symmetric lock covering and let $P = |_{i\in I} P_i$, such that P_i may be one of the followings: (1) $P_i = commit_i\langle \ell_i, \ell_i \cup \{lock_i\}, \{lock_i\}, F_i, H_i\rangle$; or (2) $P_i = state_i\langle H_i\rangle \,|\, e_i\langle put_i, \ell_i\rangle$; or (3) $P_i = failed_i\langle\rangle \,|\, e_i\langle put_i, \ell_i\rangle \,|\, H_i\langle\rangle$. Let $L \subseteq I$ be the least set such that*

1. *if $P_i = state_i\langle H_i\rangle \,|\, e_i\langle put_i, \ell_i\rangle$ then $i \in L$;*
2. *if $P_i = failed_i\langle\rangle \,|\, e_i\langle put_i, \ell_i\rangle \,|\, H_i\langle\rangle$ then $i \in L$; and*
3. *L is transitively closed, namely if $i \in L$ and $lock_j \in \ell_i$, then also $j \in L$.*

The process $\mathsf{def}\ \bigwedge_{i\in I} \mathsf{D}\sigma_i\ \mathsf{in}\ P$ *is strongly confluent and converges to*

$$\mathsf{def} \bigwedge_{i\in I} \mathsf{D}\sigma_i \ \mathsf{in} \left(\mathop{|}_{i\in I\setminus L} F_i\langle\rangle \right) \,|\, \left(\mathop{|}_{i\in L} H_i\langle\rangle \right) \,|\, G$$

where G is a parallel composition of messages on zero channels and failed messages with indexes in L (garbage).

The set G in Theorem 2 collects the active zero tokens of threads and failed messages of failed threads. These messages survive to the failure and may be consumed by future transactions, making fail any of them. In a real implementation, one may design an explicit garbage collection of these zero tokens.

Though definitions in D are generated for orchestrating threads, their task only last for the time strictly necessary to successfully conclude the transaction. A completeness result between the operational semantics of ZS nets and our distributed encoding can thus be stated as below, where D plays the role of the dynamically generated controls.

Theorem 3 (Correctness). *If* $S_1 \Rightarrow_T S_2$, *then* $\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ \{[S_1]\} \rightarrow^* \mathsf{def}\ [\![T]\!] \wedge D\ \mathsf{in}\ \{[S_2]\}$, *where the names defined in* D *appear neither in* $[\![T]\!]$ *nor in* $\{[S_2]\}$.

Theorem 3 handles the case where only successful computations are considered.

When failures occur, it is still possible to automatically normalize the reached state into a correct configuration. To this aim we define the normalization of a solution $\mathcal{S}$, noted as $norm(\mathcal{S})$, as the solution yielded by executing rules of coordinators D until termination, plus the release operations. (It is easy to check that these evaluations terminate.) The normalization process yields configurations which reflect stable configurations of ZS net. These are obtained by making all the active transactions failed, and rolling back the initial state (except for transactions that are already committing). In this normalized configuration, every remaining message on zero channels must be considered garbage since the corresponding thread is failed.

Theorem 4 (Completeness). *If* $\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ \{[S_1]\} \rightarrow^* P$, *then* $norm(P) = \mathsf{def}\ [\![T]\!] \wedge D\ \mathsf{in}\ \{[S_2]\} \mid Q$, *for some* S_2 *such that* $S_1 \Rightarrow_T^* S_2$ *and: (1)* $D = \wedge_{i \in I} \mathsf{D}\sigma_i$; *(2)* Q *is a parallel composition of messages of the form* $failed_i\langle\rangle$ *and* $e\langle put_i, \ell_i\rangle$, *such that* $e\langle put_i, \ell_i\rangle \in Q$ *implies* $failed_i\langle\rangle \in Q$. *(Roughly,* Q *keeps track of all failed coordinators and controls.)*

It is worth noting that the encoding of Definition 1 characterizes threads according to the semantic interpretation of zs nets based on the individual token philosophy. Namely, threads are collected in a transaction provided they effectively interact together.

A running implementation of the reflexive encoding in Definition 1 has been developed by Hernan Melgratti, currently PhD student in Pisa. For more information and the full code we refer the interested reader to the URL `http://www.di.unipi.it/~melgratt/D2PC/implementation.html`.

4.3 Distributing places and coordinators

The join calculus processes encoding ZS nets in Definition 1 may be easily distributed by attaching (sub)definitions to locations. In particular, locations of coordinators D are mandatory: every coordinator is located in a different location. As regards places of the net, the distribution task is left to the ZS net programmer, which must suitably partition them in separate locations.

In Figure 6 we illustrate a ZS net modeling two transactions communicating on a transactional channel. Figure 7 illustrates a join process for the net in Figure 6 with a possible distribution of definitions. In particular, the sender and the receiver are in different locations, called `locSend` and `locRec`, respectively, and new threads are created as sub-locations.

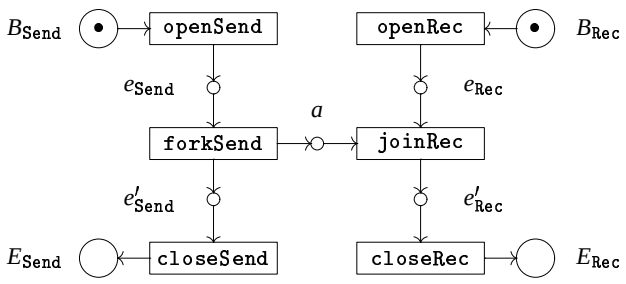


Figure 6. Net modeling of a distributed transactions.

$$
\begin{array}{l}
\mathsf{def}\ \ \texttt{locSend} : [\ B_{\texttt{Send}}\langle\rangle \triangleright \mathsf{def}\ \texttt{sub1} : [\mathsf{D}\ ,\ \mathit{state}\langle\{B_{\texttt{Send}}\}\rangle\ ]\ \mathsf{in}\ e_{\texttt{Send}}\langle \mathit{put}, \{\mathit{lock}\}\rangle \\
\qquad\qquad\qquad \wedge\, e_{\texttt{Send}}\langle p, \ell\rangle \triangleright\ \mathsf{def}\ \texttt{sub2} : [\mathsf{D}\ ,\ \mathit{state}\langle\varnothing\rangle\ ] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ e'_{\texttt{Send}}\langle p, \ell \cup \{\mathit{lock}\}\rangle \,|\, a\langle \mathit{put}, \ell \cup \{\mathit{lock}\}\rangle \\
\qquad\qquad\qquad \wedge\, e'_{\texttt{Send}}\langle p, \ell\rangle \triangleright\ p\langle \ell, \{E_{\texttt{Send}}\}\rangle\ , \\
\qquad\qquad\qquad B_{\texttt{Send}}\langle\rangle\ ] \\
\qquad \wedge\ \texttt{locRec} : [\ B_{\texttt{Rec}}\langle\rangle \triangleright \mathsf{def}\ \texttt{sub3} : [\ \mathsf{D},\ \mathit{state}\langle\{B_{\texttt{Rec}}\}\rangle\ ]\ \mathsf{in}\ e_{\texttt{Rec}}\langle \mathit{put}, \{\mathit{lock}\}\rangle \\
\qquad\qquad\qquad \wedge\, e_{\texttt{Rec}}\langle p_1, \ell_1\rangle \,|\, a\langle p_2, \ell_2\rangle \triangleright e'_{\texttt{Rec}}\langle p_1, \ell_1 \cup \ell_2\rangle \,|\, p_2\langle \ell_2 \cup \ell_1, \varnothing\rangle \\
\qquad\qquad\qquad \wedge\, e'_{\texttt{Rec}}\langle p, \ell\rangle \triangleright\ p\langle \ell, \{E_{\texttt{Rec}}\}\rangle\ , \\
\qquad\qquad\qquad B_{\texttt{Rec}}\langle\rangle\ ] \\
\mathsf{in}\ 0
\end{array}
$$

Figure 7. Reflexive encoding of a distributed transactions.

5 Compensation and nesting

There are two issues which have been expressly left aside: programmable compensations and nested transactions.

Regarding compensation, the fact is that ZS-nets implement standard compensations of database transactions as defined in Table 1, assuming that tokens can be fetched or produced, but not "updated." If large databases must be managed or if shared memory is taken into account, then it is often the case that some trade-off can be negotiated between the atomicity assumption and efficiency, so that suitable compensation code is associated to programs to avoid keeping track of all memory modifications internal to the transaction (that should be rolled back in case of failure). In our distributed encoding, compensation can be easily accomodated replacing the default continuation H for failure (i.e. the stable tokens that initiated the transaction) by the address of the compensation code, which in general can be any join calculus process that the programmer associates with net transitions. Roughly, each thread can execute a suitable algorithm $\mathsf{compensate}\langle H\rangle$ instead of $\mathsf{release}\langle H\rangle$.

For what concern nesting, the issue has not been yet considered for ZS-nets, but the intuition is that resources should be classified on different levels so that stable tokens are *tokens of level 0*, zero tokens are *tokens of level 1*, and so on. Then, at the generic

level i the resources of levels $j < i$ are seen as stable for that level. Note that in the case of ZS-nets, the maximum number of levels is statically assigned, which should suffice for modeling the two-level nesting of BizTalk. A much more interesting framework can be obtained by applying the zero-safe approach to the join calculus, where the nesting of definition and reflection can be used to dynamically generate new levels, a topic that we leave to future research.

6 Concluding remarks

Transactional aspects should play a relevant role in global computing, though the ordinary concept of transaction must be stretched out to deal with issues of distributed computations, such as asynchrony, dynamically changing topology and mobility (a number of weakenings of the ACID properties for orchestrating distributed processes has been recently discussed in [9]). To this respect, both the design and implementation phases of process orchestration should be supported by well developed, easy-to-use tools. In particular, visual formalisms and prototyping platforms should be at disposal of system analysts, and well assessed programming paradigms be at disposal of developers.

The proposal in this paper aims to fill the above gaps. In fact, it extends in many directions the concept of transaction; it comes equipped with both a graphical presentation and a concurrent semantics; it can be implemented with a fully distributed commit, which is novel to the transaction community (to the best of our knowledge). Our proposal relies on the zero-safe approach, which promotes an abstract notion of transaction that can find application to many concurrent and distributed calculi, because it is based on the following three assumptions: (1) states are multisets (e.g. of resources or messages); (2) rules can concurrently rewrite parts of the state; and (3) transactions are defined by typing resources (as stable or zero).

The first two features are common to most calculi (e.g., π, spi, join, ambients); the third one is the key for having a basic, uniform, and formal notion of transaction. Roughly, to import transactions from the ZS approach, we just partition names into stable and zero. Then configurations are separated into those containing messages on stable names only, called *stable*, and the other ones, and the operational semantics can be given in the style of Definition 2. We plan to apply this technique to the join calculus, with the aim to exploiting reflection for modeling nested transactions. The foundations and pragmatics of this "transactional" join calculus will be the argument of future work.

The distributed 2PC looks suitable for orchestrating web transactions, because it is fully distributed and it does not require the election of a global coordinator. The second feature is relevant, because when the participants are e.g. threads running on servers of different companies it is often the case that not all the companies can agree on the choice of a unique coordinator.

To conclude, let us remark that the modeling of failures in our encoding refers to the impossibility of completing a transaction in the correct way, not to the failure of the distributed join interpreter. Dealing with failures in this broader sense is an ambitious goal that is out of the scope of the present paper and that we leave to future work.

Acknowledgements. We would like to thank Satish Thatte for his kind and patient clarification of several aspects related to the design of transactions in BizTalk. We also would like to thank the anonymous referees for their constructive comments and criticisms.

References

1. P.A. 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. R. Bruni, C. Laneve, and U. Montanari. Centralized and distributed orchestration of transactions in the join-calculus. Technical Report, University of Pisa. To appear. `http://www.di.unipi.it/~bruni/publications/TRbis2002.ps.gz`
4. R. Bruni and U. Montanari. Zero-safe nets, or transition synchronization made simple. *Proc. of EXPRESS'97*, vol. 7 of *Elect. Notes in Th. Comput. Sci.* Elsevier Science, 1997.
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. *Advances in Petri Nets: Unifying Petri Nets*, vol. 2128 of *Lect. Notes in Comput. Sci.*, pp. 380–426. Springer, 2001.
7. R. Bruni and U. Montanari. Zero-safe net models for transactions in linda. *Proc. of ConCoord 2001*, vol. 54 of *Elect. Notes in Th. Comput. Sci.*, 2001.
8. M. Buscemi and V. Sassone. High-level Petri nets as type theories in the Join calculus. *Proc. of FoSSaCS 2001*, vol. 2030 of *Lect. Notes in Comput. Sci.*, pp. 104–120. Springer, 2001.
9. A. Coratella, M. Felder, R. Hirsch, and E. Rodriguez. A framework for analyzing mobile transaction. *Journal of Database Management*, 12(13), 2001.
10. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the Join calculus. *Proc. of POPL'96*, pp. 372–385. ACM Press, 1996.
11. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. *Proc. of CONCUR'96*, vol. 1119 of *Lect. Notes in Comput. Sci.*, pp. 406–421. Springer, 1996.
12. R.J. van Glabbeek and G.D. Plotkin. Configuration structures. *Proc. of LICS'95*, pp. 199–209. IEEE Computer Society Press, 1995.
13. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
14. T. Hoare. *A Model for Long-Running Transactions in XLANG*. Preliminary draft, 2001.
15. B. Laskey and J. Parker. Microsoft biztalk server 2000: Building a reverse auction with biztalk orchestration, 2001. Microsoft Corporation. Available at [18].
16. W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer, 1985.
17. U. Roxburgh. Biztalk orchestration: Transactions, exceptions, and debugging, 2001. Microsoft Corporation. Available at [18].
18. URL `http://msdn.microsoft.com/library/en-us/dnbiz/html/bizorchestr.asp`.