Orchestrating Transactions in Join Calculus*

Roberto Bruni† Cosimo Laneve‡ Ugo Montanari§

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. In particular, our model: (1) extends Microsoft® BizTalk's short transactions; (2) it exploits an original commit algorithm, which is distributed; (3) it can deal with dynamically changing communication topology; (4) it is almost language-independent. In fact, the model is simply based on a two-level classification of resources, which should be easily conveyed to distributed calculi and languages, providing them with a powerful 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 (*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.

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 [16] is a system explicitly designed for process orchestrations. 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*. Short transactions consist of series of actions, while long and timed ones may group any combination of actions and transactions (up to two nesting levels). Since long and timed transactions require programmers to define the on-failure compensation algorithms, we focus on short transactions only, which retain automatic recovery.

We notice 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]. Actually, BizTalk flowcharts may be easily encoded into ZS nets, while the converse encoding seems not possible. The reason is that BizTalk boxes provide a somewhat "static" specification of transactions

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

†Dipartimento di Informatica, Università di Pisa, Corso Italia n. 40, 56125 Pisa, Italia. Tel: +39 050 2212768. Fax: +39 050 2212726. bruni@di.unipi.it **(corresponding author)**

‡Dipartimento di Scienze dell'Informazione, Università di Bologna, Italia, laneve@cs.unibo.it

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

A *transaction* can be considered as a collection of actions with the following properties:

Atomicity, A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

Consistency, A transaction is a correct transformation of the state. The action taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program.

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].

boundaries, while the state partitioning gives a "dynamic" notion of them, which depends on the runtime behaviour of processes (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 it 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.

We provide two implementations of ZS nets:

- the *centralized implementation*, reflecting BizTalk transaction implementation, which is modeled in a "flat" version of the join-calculus;
- the *distributed implementation*, which exploits the whole expressive power of the join-calculus for the dynamic definition of local transaction controllers.

The last implementation exploits a *distributed* 2PC *protocol* (a variant of the *decentralized* 2PC protocol [1]) that, up-to our knowledge, is original 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. Every process receiving a failure message also fails.

In the distributed 2PC protocol, the number of participants and their names are not statically fixed. For this reason, in the distributed 2PC protocol, the role of the coordinator is played by all participants. In particular, every participant builds 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.

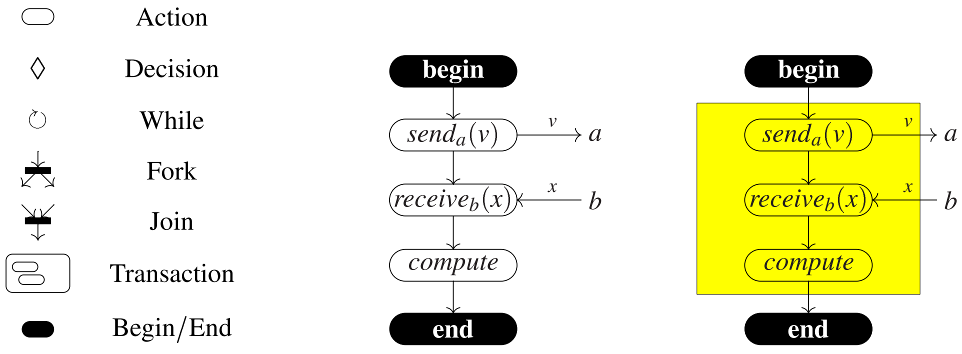
Synopsis. In Sections 2 and 3, we give a brief account of the transactional mechanism of BizTalk, and discuss its modeling in the theory of ZS nets, where also multiway transactions can be designed. In Section 4, we present the centralized and the distributed encoding of ZS nets in join calculus. Finally,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

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.

[1] Our actual understanding of BizTalk's transactional mechanism is mostly based on a series of documents available on the web through MSDN Library `http://msdn.microsoft.com/library` [14, 16].



(b) Simple flowchart.

(c) A transaction.

(a) Basic shapes.

Figure 1. Flowcharts in BizTalk Orchestration Visual Designer.

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 process cannot emit v on a and input x from b before 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.

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 the response from the DTC 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 transactions at the DTC. Actually, the process in Figure 2(a) shows up a further issue: it is required DTC become aware that the two transactions must commit together. This is achieved by making channels "*transactional*" (channels may be transactional or not), which amounts to joining them with twin COM+ components. These latter components, at run-time, report to DTC that the two transactions communicated.

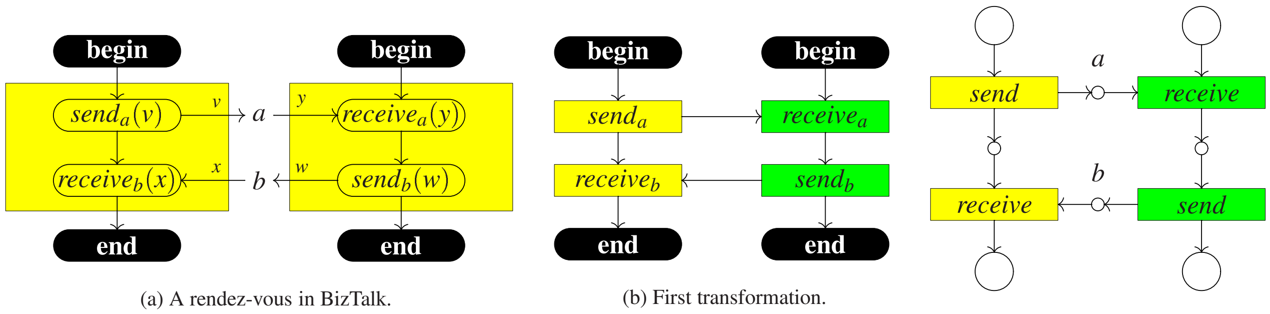
Built-in recovery is available in BizTalk for the so-called *short-lived* transactions only. In the following we disregard the other two kinds of transactions, i.e. *long-running* and *timed*, which rely on user-defined compensation code.

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: in the first step we draw transaction in a different manner dropping boxes; in the second step we introduce states.

First step: coloring basic shapes. In BizTalk, transactions are depicted by grouping basic shapes into (shadowed) boxes (Figures 1(b) and 2(a)). Short transactions can be neither intersected nor nested (nesting is admitted inside long/timed transactions). In alternative to using boxes, one may color basic shapes therein. Shapes included in different boxes have different colors; shapes which are outside transactions 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 channel link, if the channel is transactional then the circle is small, otherwise it is large. Begin and



(a) A rendez-vous in BizTalk.

(b) First transformation.

(c) Second transformation.

Figure 2. Alternative drawings of BizTalk transactions.

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

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

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

(COMMIT)
$$\frac{(S,\varnothing) \rightarrow_T (S',\varnothing)}{(S,\varnothing) \Rightarrow_T (S',\varnothing)}$$

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

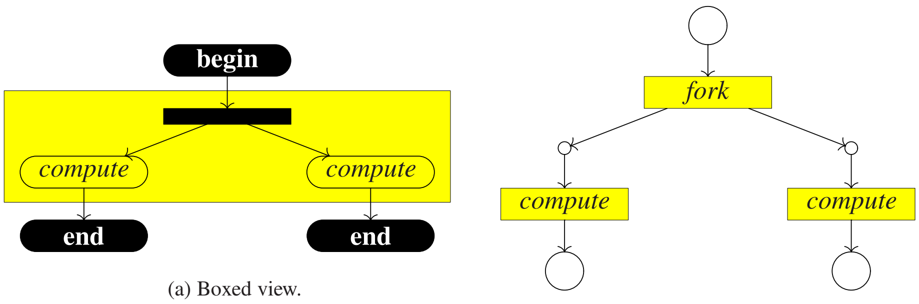
end shapes are regarded as stable places (a sort of entry/exit points). This second step is illustrated in Figure 2(c) for the flowchart in Figure 2(a), where channels are considered transactional.

The semantics of the net in Figure 2(c) departs from the standard one for Petri nets [15] because of transactions. In Petri nets, places are repositories of resources, called *tokens*, and transitions fetch and release resources.[2] Zero-safe nets have been introduced in [3] 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 formalized by the two relations $\Rightarrow$ and $\rightarrow$ 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, defining the execution of one/many transition(s). However, sequences of steps differ from the ordinary transitive closure of $\rightarrow$: The rule CONCATENATION composes zero tokens in series but stable tokens in parallel, so that stable tokens produced by the first step cannot be consumed by the second step. Transactions are step sequences which begin and end at stable markings, when COMMIT can be applied. We present ZS nets as pairs (T,S) where S is the initial marking, and abbreviate $(S,\varnothing) \Rightarrow_T (S',\varnothing)$ into $S \Rightarrow_T S'$.

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 say that S' is reachable from S if $S \Rightarrow_T^* S'$. We refer the interested reader to [5] for the characterization of the minimal transactions—in the sense that they cannot be decomposed into smaller transactions—that is dependent on the "token philosophy" under consideration: *collective* or *individual* [12]. Roughly, the collective philosophy assumes that all tokens in the same place are indistinguishable, while the individual philosophy says that tokens can have different histories and that, if so, it is important to distinguish them for keeping, e.g., causality information between threads.

[2] For simplicity, here we leave aside that resources 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.

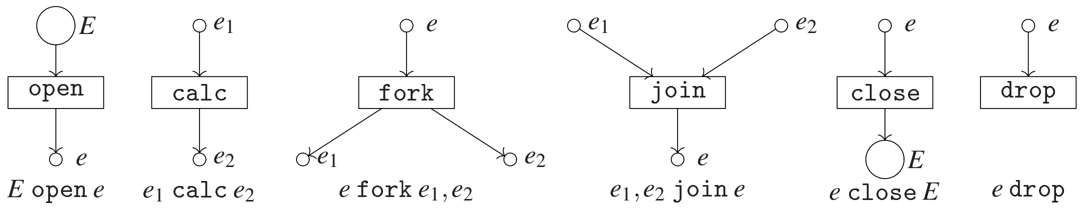


Figure 4. Basic shapes of extended BizTalk.

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 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 processes is motivated by implementation issues, rather than theoretical motivations. Indeed, two transactions exchanging messages on transactional channels form already a multiway transaction and are allowed in BizTalk.

ZS nets deal uniformly with multiway and single-threaded transactions, and therefore the distinction is transparent to our encodings.

4 Encoding multiway transactions in join

In this section we present two accounts of the 2PC protocol for Biztalk with multiway transactions: a centralized one and a distributed one. The centralized algorithm reflects BizTalk approach, where transactions are orchestrated by the DTC. 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 remark that for ordinary Petri net transitions (i.e. without zero places) a direct encoding in join is obviously possible.

We notice 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 *receive* shapes as *fork* and *join*, respectively.

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, $\ldots$; tuples of names are written $\widetilde{u}$. 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 $x\langle\widetilde{u}\rangle$, a parallel composition of processes, or a defining process def D in P. A definition D is a conjunction of reaction rules $J \triangleright P$ that associate join-

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	$P \mid Q \equiv P, Q$	$\mathsf{def}\ D\ \mathsf{in}\ P \equiv D\sigma, P\sigma$	$D \wedge D' \equiv D, D'$
		(σ renames defined names in D with fresh names)	
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.

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.

A process P is *flat* if all guarded processes inside P have no defining process. The *flat join calculus* is the subcalculus of flat processes (the property of being flat is preserved under reduction). Therefore, in flat join calculus, no fresh definition can be dynamically generated. In fact, the flat join calculus is equivalent to high-level nets (see [8]), with D and P playing the roles of the set of transitions and of the initial marking, respectively (message $x\langle\tilde{u}\rangle$ models a token with value $\tilde{u}$ in the place x).

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

4.2 Centralized 2PC

We begin by translating ZS nets in flat join calculus. To this aim, we extend the language with integers and pattern matching on integers. Namely, join patterns may take integer values, and the corresponding guarded process may be triggered provided the matching messages carry the same values. The following encoding accounts for ZS nets conveying unvalued messages only. The general case follows by taking additional parameters into account.

Definition 4.1 (Flat encoding) *The flat join-calculus process associated to a* ZS *net* (T,S) *is* $\mathsf{def}\ [\![T]\!]\ \mathsf{in}\ [\![S]\!]$, *where*

$$\begin{array}{rcl} [\![T]\!] & = & Cont \wedge Fail \wedge (\bigwedge_{t \in T} \{[t]\}) \\ [\![S]\!] & = & gen\langle 0\rangle \mid \{[S]\} \end{array}$$

with $\{[t]\}$, $\{[S]\}$ *defined in Figure 5, and Cont, Fail defined in Figure 6. In the following, channels which encode stable and zero tokens are addressed as* stable *and* zero *channels, respectively.*

When a transaction begins (firing a transition which does not require zero tokens) a positive integer κ is picked up. This is the *name of the transaction*, and every transition of the transaction is tagged with this name. In this way, only threads in the same transaction (i.e., transitions that have picked up the same transaction name) can interact.

Transaction names correspond to controllers $state\langle\kappa,n\rangle$, which record the number n of zero tokens that must be consumed to commit. New controllers may be dynamically released by the generator $gen\langle\kappa\rangle$ with the transition

$$gen\langle\kappa\rangle \triangleright gen\langle\kappa+1\rangle \mid state\langle\kappa+1,0\rangle$$

where κ is the last generated transaction name and $\kappa+1$ is the fresh transaction name. Zero tokens are then labeled with the name κ of the transaction they belong to. A transition willing to increase/decrease the number of zero tokens must inform the controller $state\langle\kappa,n\rangle$ to update the value n. However, to augment the parallelism among transitions of the same transaction κ, we use "credit" tokens $cdt\langle\kappa\rangle$ and "debit" tokens $dbt\langle\kappa\rangle$. For instance, for a fork transition e `fork` e_1, e_2 this is achieved through two reactions:

$$\begin{array}{c} state\langle\kappa,n\rangle \triangleright state\langle\kappa,n+1\rangle \mid cdt\langle\kappa\rangle \\ e\langle\kappa\rangle \mid cdt\langle\kappa\rangle \triangleright e_1\langle\kappa\rangle \mid e_2\langle\kappa\rangle \end{array}$$

Similarly, for a join transition e_1, e_2 `join` e, we have

$$\begin{array}{c} e_1\langle\kappa\rangle \mid e_2\langle\kappa\rangle \triangleright e\langle\kappa\rangle \mid dbt\langle\kappa\rangle \\ state\langle\kappa,n+1\rangle \mid dbt\langle\kappa\rangle \triangleright state\langle\kappa,n\rangle \end{array}$$

$$\begin{aligned}
\{[E\ \mathtt{open}\ e]\} &= E\langle\rangle \,|\, cdt\langle\kappa\rangle \triangleright e\langle\kappa\rangle \,|\, fetched\langle\kappa,E\rangle & \{[E+S]\} &= E\langle\rangle \,|\, \{[S]\} \\
\{[e\ \mathtt{fork}\ e_1,e_2]\} &= e\langle\kappa\rangle \,|\, cdt\langle\kappa\rangle \triangleright e_1\langle\kappa\rangle \,|\, e_2\langle\kappa\rangle & \{[\varnothing]\} &= \varnothing \\
\{[e_1,e_2\ \mathtt{join}\ e]\} &= e_1\langle\kappa\rangle \,|\, e_2\langle\kappa\rangle \,|\, \triangleright e\langle\kappa\rangle \,|\, dbt\langle\kappa\rangle \\
\{[e_1\ \mathtt{calc}\ e_2]\} &= e_1\langle\kappa\rangle \triangleright e_2\langle\kappa\rangle \\
\{[e\ \mathtt{close}\ E]\} &= e\langle\kappa\rangle \triangleright frozen\langle\kappa,E\rangle \,|\, dbt\langle\kappa\rangle \\
\{[e\ \mathtt{drop}]\} &= e\langle\kappa\rangle \triangleright dbt\langle\kappa\rangle
\end{aligned}$$

Figure 5. Definition of $\{[t]\}$ and $\{[S]\}$.

$$\begin{aligned}
Cont = \quad & gen\langle\kappa\rangle \triangleright gen\langle\kappa+1\rangle \,|\, state\langle\kappa+1,0\rangle \\
\wedge \quad & state\langle\kappa,n\rangle \triangleright state\langle\kappa,n+1\rangle \,|\, cdt\langle\kappa\rangle \\
\wedge \quad & state\langle\kappa,n+1\rangle \,|\, dbt\langle\kappa\rangle \triangleright state\langle\kappa,n\rangle \\
\wedge \quad & cdt\langle\kappa\rangle \triangleright dbt\langle\kappa\rangle \\
\wedge \quad & state\langle\kappa,0\rangle \triangleright commit\langle\kappa\rangle \\
\wedge \quad & commit\langle\kappa\rangle \,|\, frozen\langle\kappa,E\rangle \triangleright commit\langle\kappa\rangle \,|\, E\langle\rangle \\
\wedge \quad & commit\langle\kappa\rangle \,|\, fetched\langle\kappa,E\rangle \triangleright commit\langle\kappa\rangle
\end{aligned}$$

(a) Controller and commit.

$$\begin{aligned}
Fail = \quad & state\langle\kappa,n\rangle \triangleright failed\langle\kappa,n\rangle \\
\wedge \quad & failed\langle\kappa,n+1\rangle \,|\, cdt\langle\kappa\rangle \triangleright failed\langle\kappa,n\rangle \\
\wedge \quad & failed\langle\kappa,n+1\rangle \,|\, dbt\langle\kappa\rangle \triangleright failed\langle\kappa,n\rangle \\
\wedge \quad & failed\langle\kappa,n\rangle \,|\, frozen\langle\kappa,E\rangle \triangleright failed\langle\kappa,n\rangle \\
\wedge \quad & failed\langle\kappa,n\rangle \,|\, fetched\langle\kappa,E\rangle \triangleright failed\langle\kappa,n\rangle \,|\, E\langle\rangle
\end{aligned}$$

(b) Failure.

Figure 6. Shapes for transaction controllers.

Controllers generate as many $cdt\langle\kappa\rangle$ tokens as those needed during the transaction. The transition $cdt\langle\kappa\rangle \triangleright dbt\langle\kappa\rangle$ recovers surplus tokens. When the controller is in the state $state\langle\kappa,0\rangle$, it means that the transaction κ is terminated with success and the rule

$$state\langle\kappa,0\rangle \triangleright commit\langle\kappa\rangle$$

is used for entering the success state. The message $commit\langle\kappa\rangle$ says that stable tokens produced by the thread κ can now be released. To this purpose, a repository *frozen* is employed to record tokens of the form $frozen\langle\kappa,E\rangle$, and the rule

$$commit\langle\kappa\rangle \,|\, frozen\langle\kappa,E\rangle \triangleright commit\langle\kappa\rangle \,|\, E\langle\rangle$$

is used to release them.

Rules in *Fail* manage failure cases. We use the repository *fetched*, analogous to *frozen*, for tokens which must be released in case of failure. A message $fetched\langle\kappa,E\rangle$ records the stable tokens E that have been fetched by the transaction κ (see the encoding of $E\,\mathtt{open}e$). Nondeterministically, a transaction can fail by means of

$$state\langle\kappa,n\rangle \triangleright failed\langle\kappa,n\rangle.$$

In the failure state *failed*, tokens $fetched\langle\kappa,E\rangle$ are released, and tokens $frozen\langle\kappa,E\rangle$ are garbage collected.

Remark 4.2 *Every solution yielded while evaluating the agent of Definition 4.1 satisfies the following properties:*

1. *there is exactly one message on the channel gen, say $gen\langle\bar\kappa\rangle$;*
2. *there is no message carrying values κ, with $\kappa > \bar\kappa$;*
3. *for every positive $\kappa \leq \bar\kappa$ there is exactly one message*
   (a) *either of the form $state\langle\kappa,n\rangle$ or $failed\langle\kappa,n\rangle$. In this case, the total number of messages in $cdt\langle\kappa\rangle$, $dbt\langle\kappa\rangle$, and $e\langle\kappa\rangle$ is n;*
   (b) *or of the form $commit\langle\kappa\rangle$. In this case, the number of messages in $cdt\langle\kappa\rangle$, $dbt\langle\kappa\rangle$, and $e\langle\kappa\rangle$ is 0.*

Below we find convenient to represent flat join calculus solutions as pairs (D,M), where D is a flat definition and M is a parallel composition of messages. The *observable* of a parallel composition of messages M, in notation $obs(M)$, is the subprocess retaining all stable messages $E\langle\rangle$ in M. The normalization of a solution (D,M), noted as $norm(D,M)$, is the solution yielded by executing rules of shape

$$\begin{aligned}
& Fail \\
& \wedge\, commit\langle\kappa\rangle \,|\, frozen\langle\kappa,E\rangle \triangleright commit\langle\kappa\rangle \,|\, E\langle\rangle \\
& \wedge\, commit\langle\kappa\rangle \,|\, fetched\langle\kappa,E\rangle \triangleright commit\langle\kappa\rangle
\end{aligned}$$

until termination. (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). One may prove that, if $([\![T]\!],[\![S]\!]) \to^* ([\![T]\!],M)$, then $norm([\![T]\!],M)$ is such that messages in the solutions concern stable or zero channels, and channels *gen*, *failed*, and *commit*. No message is carried by channels $fetched$, $frozen$, *state*, *cdt*, or *dbt*. In this normalized configuration, every remaining message on zero channels must be considered garbage since the corresponding thread is failed.

The Proposition below shows that normalization can be performed at any step without compromising reachability of (normalized) states.

Proposition 4.3 (Normalization is harmless) *Let (T,S) be a* ZS *net, and let* def $[\![T]\!]$ in $[\![S]\!] \to^* ([\![T]\!],M)$. *If $([\![T]\!],M) \to^* ([\![T]\!],M')$, then $norm([\![T]\!],M) \to^* norm([\![T]\!],M')$.*

The following results establish the correctness and completeness of the encoding in Definition 4.1.

Theorem 4.4 (Correctness) *Let $S_1 \Rightarrow_T S_2$. Then, for any normalized solution $([\![T]\!],M_1)$ satisfying the properties in Remark 4.2, and with $obs(M_1) = \{[S_1]\}$, there is M_2 such that $obs(M_2) = \{[S_2]\}$ and $([\![T]\!],M_1) \to^* ([\![T]\!],M_2)$.*

Proof (Sketch). We label $\to_T$ with pairs (S,m), where

- S is the multiset of stable tokens which stay idle during the whole computation;
- m is the number of `open` and `fork` operations which are performed during the computation.

We remark that $\to_T$ actually consists of several elementary steps in sequence and/or in parallel, each of them may contain an `open` or a `fork` operation.

By rule induction, we demonstrate that

$$(S+S'_1,Z_1) \xrightarrow{S,m}_T (S+S'_2,Z_2)$$

implies, for any κ, $([\![T]\!],M) \to^* ([\![T]\!],M')$, where

$$\begin{aligned} M = \quad & \{[S+S'_1]\} \,|\, (|_{e\in Z_1}\, e\langle\kappa\rangle) \,|\, (|_{i\in 1..m}\, cdt\langle\kappa\rangle) \\ M' = \quad & \{[S]\} \,|\, (|_{E\in S'_1}\, fetched\langle\kappa,E\rangle) \,|\, (|_{E\in S'_2}\, frozen\langle\kappa,E\rangle) \\ & |\, (|_{e\in Z_2}\, e\langle\kappa\rangle) \,|\, (|_{i\in 1..m+|Z_1|-|Z_2|}\, dbt\langle\kappa\rangle) \end{aligned}$$

We notice that M represents the minimal set of resources needed to perform the sequence of reductions. Of course, $([\![T]\!],M\,|\,P) \to^* ([\![T]\!],M'\,|\,P)$, for every P.

Since $S_1 \Rightarrow_T S_2$, there exist S, S'_1, S'_2 and m such that

$$(S_1,\varnothing) = (S+S'_1,\varnothing) \xrightarrow{S,m}_T (S+S'_2,\varnothing) = (S_2,\varnothing)$$

Hence, let $([\![T]\!],M_1)$ be normalized, satisfy Remark 4.2, and with $obs(M_1) = \{[S_1]\}$. Then $M_1 = \{[S_1]\}\,|\,gen\langle\bar\kappa\rangle\,|\,G$, for some $\bar\kappa$ and "garbage" G, and we may execute the following sequence of reductions

$$\begin{aligned} & ([\![T]\!],\{[S_1]\}\,|\,gen\langle\bar\kappa\rangle\,|\,G) \\ & \to ([\![T]\!],\{[S_1]\}\,|\,gen\langle\bar\kappa+1\rangle\,|\,state\langle\bar\kappa+1,0\rangle\,|\,G) \\ & \to^* ([\![T]\!],\ \{[S_1]\}\,|\,gen\langle\bar\kappa+1\rangle\,|\,state\langle\bar\kappa+1,m\rangle \\ & \qquad |(|_{i\in 1..m}\, cdt\langle\bar\kappa+1\rangle)\,|\,G\ ) \end{aligned}$$

By the arguments above, it follows that $([\![T]\!],M'_1\,|\,G) \to^* ([\![T]\!],M'_2\,|\,G)$, where

$$\begin{aligned} M'_1 = \quad & \{[S_1]\}\,|\,gen\langle\bar\kappa+1\rangle\,|\,state\langle\bar\kappa+1,m\rangle\,|(|_{i\in 1..m}\, cdt\langle\bar\kappa+1\rangle) \\ M'_2 = \quad & \{[S]\}\,|\,gen\langle\bar\kappa+1\rangle\,|\,state\langle\bar\kappa+1,m\rangle\,|(|_{i\in 1..m}\, dbt\langle\bar\kappa+1\rangle) \\ & |\,(|_{E\in S'_1}\, fetched\langle\bar\kappa+1,E\rangle)\,|\,(|_{E\in S'_2}\, frozen\langle\bar\kappa+1,E\rangle) \end{aligned}$$

To the solution $([\![T]\!],M'_2\,|\,G)$, we apply the following sequence of rules:

1. m times the rule $state\langle\kappa,n+1\rangle\,|\,dbt\langle\kappa\rangle \triangleright state\langle\kappa,n\rangle$;
2. the rule $state\langle\kappa,0\rangle \triangleright commit\langle\kappa\rangle$;
3. $|S'_1|$ times the rule
$$commit\langle\kappa\rangle\,|\,fetched\langle\kappa,E\rangle \triangleright commit\langle\kappa\rangle \ ;$$
4. $|S'_2|$ times the rule
$$commit\langle\kappa\rangle\,|\,frozen\langle\kappa,E\rangle \triangleright commit\langle\kappa\rangle\,|\,E\langle\rangle \ .$$

We therefore obtain the normalized configuration

$$([\![T]\!],\{[S_2]\}\,|\,gen\langle\bar\kappa+1\rangle\,|\,commit\langle\bar\kappa+1\rangle\,|\,G)$$

whose observable is clearly $\{[S_2]\}$. □

We remark that transactions may be executed with the same degree of concurrency both in (T,S) and in $([\![T]\!],[\![S]\!])$. Although this aspect is not explicitly set out in our results, where only reachability is considered, it has a great impact in the design of the encoding. For example, credit tokens are not generated by-need, because otherwise all transitions `open` and `fork` in the transaction κ would be sequentialized while requesting $cdt\langle\kappa\rangle$.

Theorem 4.5 (Completeness) *Let $([\![T]\!],M_1)$ be normalized and satisfy the properties in Remark 4.2. Let $norm([\![T]\!],M_1) \to^* norm([\![T]\!],M'_2) = ([\![T]\!],M_2)$. Then $obs(M_1) \Rightarrow^*_T obs(M_2)$.*

Proof (Sketch). The proof consists of several steps:

1. The derivation $norm([\![T]\!],M_1) \to^* norm([\![T]\!],M'_2)$ can possibly execute several transactions at a time. We show that there is an equivalent derivation which interleaves transactions.
2. We prove that failing transactions can be eliminated without affecting reachability.
3. We proceed by induction on the number of transactions. In the basic case we extract the sequence of reductions that are images of transitions in T and prove inductively that the same step can be performed on the net. The inductive case is analogous.

□

4.3 Distributed 2PC

A more efficient encoding of ZS nets relies on the full join calculus. In particular, dynamically active definitions allow to implement

- local names and coordinators for transactions (rather than global ones);
- a distributed commit (rather than a centralized one);
- on-demand updatings of local managers.

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 release which takes a set of channels and sends an (empty) message on every channel in the set.

Definition 4.6 (Reflexive encoding) *The join calculus process of a* ZS *net* (T,S) *is*

$$\mathsf{def}\ [\![T]\!]^{\mathrm{R}}\ \mathsf{in}\ \{[S]\}$$

where $[\![T]\!]^{\mathrm{R}} = \wedge_{t\in T}[\![t]\!]^{\mathrm{R}}$ *with* $[\![t]\!]^{\mathrm{R}}$ *defined in Figure 7 (and* $\{[S]\}$ *as in Figure 5). As in Definition 4.1, we address channels encoding stable and zero tokens as stable and zero channels, respectively.*

A new thread is started either when (1) *a transition* $\mathtt{open}(E,e)$ *occurs*, or when (2) *a thread forks*. This is evidenced by controller definitions D in the join calculus agent. Every thread has at most one active zero token at any time. A thread may terminate when its associated zero token has been consumed—a `join`, or `close`, or `drop` is executed. 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 7. 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 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 triggered by time-outs.) In the state $failed\langle\rangle$, any thread in the same transactions 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 represented by the *commit* message, which carries the 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 set of threads to which the successful completion signal has been already sent. Initially, ℓ'' contains the pointer to itself, thus avoiding self-addressed messages;
4. F and H store the stable channels which must 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
   $$commit\langle \{l\}\cup\ell,\ell',\ell'',F,H\rangle \ \triangleright \quad commit\langle \ell,\ell',\ell'',F,H\rangle \,|\, l\langle \ell',lock,fail\rangle.$$
   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:

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

where D is the following definition:

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

Figure 7. The encoding in the join calculus.

$$
\begin{aligned}
&commit\langle\ell,\ell',\ell'',F,H\rangle \,|\, lock\langle\ell''',l,f\rangle \ \triangleright \\
&\qquad commit\langle\ell\cup(\ell'''\setminus\ell'),\ell'\cup\ell''',\ell''\cup\{l\},F,H\rangle.
\end{aligned}
$$

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 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. This task is achieved by:

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

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.

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 0, in the latter case it represents $a\langle\rangle$. Moreover, we let a *symmetric lock partitioning* 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 4.7 (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 partitioning. Then*

$$\mathsf{def}\ \bigwedge_{i\in I}\mathsf{D}\sigma_i\ \mathsf{in}\ P$$

is strongly confluent, in the sense that it always converges after a finite number of steps to the configuration

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

Moreover, the number of steps is bound by $O(n^2)$.

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 4.8 (Correctness of the distributed 2PC, part 2) *Let $P = |_{i\in I}\, P_i$, such that P_i may be one of the followings:*

- *$commit_i\langle\ell_i,\ell_i\cup\{lock_i\},\{lock_i\},F_i,H_i\rangle$; or*
- *$state_i\langle H_i\rangle \,|\, e_i\langle put_i,\ell_i\rangle$; or*
- *$failed_i\langle\rangle \,|\, e_i\langle put_i,\ell_i\rangle \,|\, H_i\langle\rangle$;*

where $\{\ell_i\}_{i\in I}$ is a symmetric lock partitioning. 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 $P_i = failed_i\langle\rangle \mid e_i\langle put_i, \ell_i\rangle \mid H_i\langle\rangle$ then $i \in L$; and*

3. *L is transitively closed, namely if $i \in L$ and $lock_j \in \ell_i$, then also $j \in L$.*

Then, the process

$$\mathsf{def}\ \bigwedge_{i\in I} \mathsf{D}\sigma_i\ \mathsf{in}\ P$$

is strongly confluent and converges to

$$\mathsf{def}\ \bigwedge_{i\in I} \mathsf{D}\sigma_i\ \mathsf{in}\ (\ \prod_{i\in I\setminus L} F_i\langle\rangle) \mid (\ \prod_{i\in L} H_i\langle\rangle) \mid G$$

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

The set G in Theorem 4.8 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 the theorem below, where D plays the role of the dynamically generated controls.

Theorem 4.9 (Correctness) *If $S_1 \Rightarrow_T S_2$, then*

$$\mathsf{def}\ [\![T]\!]^{\mathrm{R}}\ \mathsf{in}\ \{[S_1]\} \to^* \mathsf{def}\ [\![T]\!]^{\mathrm{R}} \wedge D\ \mathsf{in}\ \{[S_2]\}$$

where the names defined in D appear neither in $[\![T]\!]^{\mathrm{R}}$ nor in $\{[S_2]\}$.

Proof (Sketch). The proof is analogous to the one for the centralized algorithm. We reconsider $\to_T$ labeled with pairs (S,m), where S are the idle stable tokens and m is the total number of `open` and `fork` transitions fired.

By rule induction we prove that if

$$(S+S_1',Z) \xrightarrow{S,m}_T (S+S_2',Z')$$

where $Z = \{e_1,\cdots,e_k\}$ and $Z' = \{e'_{i_1},\cdots,e'_{i_h}\}$ then, for any symmetric lock partitioning $\{\ell_i\}_{i\in 1..k}$ and for any k sets of stable places $H_1,\ldots,H_k$ with $|H_i| \leq 1$ for $i \in 1..k$, we get

$$\begin{aligned}&\mathsf{def}\ [\![T]\!]^{\mathrm{R}} \wedge (\textstyle\bigwedge_{i\in 1..k} \mathsf{D}\sigma_i)\ \mathsf{in}\ \{[S+S_1']\} \mid M\\ &\qquad \to^*\ \mathsf{def}\ [\![T]\!]^{\mathrm{R}} \wedge (\textstyle\bigwedge_{i\in 1..k+m} \mathsf{D}\sigma_i)\ \mathsf{in}\ \{[S]\} \mid M' \mid M''\end{aligned}$$

where

- $M = |_{i\in 1..k}(e_i\langle put_i,\ell_i\rangle \mid state\langle H_i\rangle)$,
- $M' = |_{j\in 1..h}(e'_{i_j}\langle put_{i_j},\ell'_{i_j}\rangle \mid state\langle H_{i_j}\rangle)$, and
- $M'' = |_{j\in L}(state_j\langle H_j\rangle \mid put_j\langle \ell'_j, F_j\rangle)$, where $L = (1..k+m)\setminus\{i_1,\cdots,i_h\}$, $S_1' = \bigcup_{j\in L} H_j$ and $S_2' = \bigcup_{j\in L} F_j$.

for a suitable symmetric lock partitioning $\{\ell'_j\}_{j\in 1..k+m}$ and sets H_i, F_i.

Since $S_1 \Rightarrow_T S_2$, there exist S, S_1', S_2' and m such that

$$(S_1,\varnothing) = (S+S_1',\varnothing) \xrightarrow{S,m}_T (S+S_2',\varnothing) = (S_2,\varnothing)$$

By the arguments above, letting $Z = Z' = \varnothing$, we have that $M = M' = 0$ and $M'' = |_{i\in 1..m}(state_i\langle H_i\rangle \mid put_i\langle \ell'_i, F_i\rangle)$. Therefore, by firing the rule (in $\mathsf{D}\sigma_i$)

$$\begin{aligned}&state_i\langle H_i\rangle \mid put_i\langle \ell'_i, F_i\rangle \triangleright\\ &\qquad\qquad commit_i\langle \ell'_i\setminus\{lock_i\}, \ell'_i, \{lock_i\}, F_i, H_i\rangle\end{aligned}$$

for all $i \in 1..m$, every thread commits, and we are in condition for applying the correctness result for the distributed commit and conclude the proof. □

Theorem 4.9 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, as in the centralized encoding. In this case, the normalization process consists of repeated applications of reaction rules of coordinators D until termination.

Theorem 4.10 (Completeness) *If $\mathsf{def}\ [\![T]\!]^{\mathrm{R}}\ \mathsf{in}\ \{[S_1]\} \to^* P$, then $norm(P) = \mathsf{def}\ [\![T]\!]^{\mathrm{R}} \wedge D\ \mathsf{in}\ \{[S_2]\} \mid Q$, for some S_2 such that $S_1 \Rightarrow_T^* S_2$ and:*

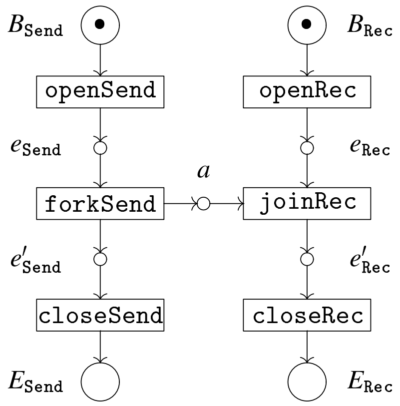
- *$D = \wedge_{i\in I} \mathsf{D}\sigma_i$;*
- *Q is a parallel composition of messages of the form $failed_i\langle\rangle$ and $e\langle put_i,\ell_i\rangle$, such that $e\langle put_i,\ell_i\rangle \in Q$ implies $failed_i\langle\rangle \in Q$.*

In Theorem 4.10, Q keeps track of all failed coordinators and controls.

It is worth noting that the encoding of Definition 4.6 characterizes threads according to the individual token philosophy. Namely, threads are collected in a transaction provided they effectively interact together. On the contrary, the encoding of Definition 4.1 can group disjoint threads into the same transaction.

4.4 Distributing places and coordinators

The join calculus process encoding ZS nets in Definition 4.6 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.



(a) Net modeling.

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

(b) Reflexive encoding.

Figure 8. A distributed transactions.

In Figure 8(a) we illustrate a ZS net modeling two transactions communicating on a transactional channel. Figure 8(b) illustrates a join process for the net in Figure 8(a) 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.

5 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 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;
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, powerful, and formal characterization of transactions. Roughly, to import transactions from the ZS approach, it suffices to partition names into stable and zero. Then configurations are separated into those containing messages on stable names only, called *stable configurations*, and the other ones. To such configurations, one defines an operational semantics 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, implementations, and pragmatics of this "transactional" join calculus will be the argument of future research.

References

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

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

[3] R. Bruni and U. Montanari. Zero-safe nets, or transition synchronization made simple. In C. Palamidessi and J. Parrow, editors, *Proceedings of EXPRESS'97, 4th workshop on Expressiveness in Concurrency*, volume 7 of *Elect. Notes in Th. Comput. Sci.* Elsevier Science, 1997.

[4] R. Bruni and U. Montanari. Executing transactions in zero-safe nets. In M. Nielsen and D. Simpson, editors, *Proceedings of ICATPN 2000, 21st Int. Conf. on Application and*

Theory of Petri Nets, volume 1825 of *Lect. Notes in Comput. Sci.*, pages 83–102. Springer Verlag, 2000.

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

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

[7] R. Bruni and U. Montanari. Zero-safe net models for transactions in linda. In U. Montanari and V. Sassone, editors, *Proceedings of ConCoord 2001, International Workshop on Concurrency and Coordination*, volume 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. In F. Honsell and M. Miculan, editors, *Proceedings of FoSSaCS 2001, Foundations of Software Science and Computation Structures*, volume 2030 of *Lect. Notes in Comput. Sci.*, pages 104–120. Springer Verlag, 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. In *Proceedings of POPL'96, 23rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[11] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR'96, 7th International Conference on Concurrency Theory*, volume 1119 of *Lect. Notes in Comput. Sci.*, pages 406–421. Springer Verlag, 1996.

[12] R. v. Glabbeek and G. Plotkin. Configuration structures. In D. Kozen, editor, *Proceedings of LICS'95, 10th Annual IEEE Symposium on Logics in Computer Science*, pages 199–209. IEEE Computer Society Press, 1995.

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

[14] B. Laskey and J. Parker. Microsoft biztalk server 2000: Building a reverse auction with biztalk orchestration, 2001. Microsoft Corporation. Available at `http://msdn.microsoft.com/library/en-us/dnbiz/html/bizorchestr.asp`.

[15] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1985.

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