A Calculus for Long-Running Transactions

Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A.Zamboni 7, I-40127 Bologna, Italy
{bocchi,laneve,zavattar}@cs.unibo.it

Abstract. We study *long-running transactions* in open component-based distributed applications, such as Web Services platforms. Long-running transactions describe time-extensive activities that involve several distributed components. Henceforth, in case of failure, it is usually not possible to restore the initial state, and firing a compensation process is preferable. Despite the interest of such transactional mechanisms, a formal modeling of them is still lacking. In this paper we address this issue by designing an extension of the asynchronous π-calculus with long-running transactions (and sequences) – the $\pi\mathsf{t}$-*calculus*. We study the practice of $\pi\mathsf{t}$-calculus, by discussing few paradigmatic examples, and its theory, by defining a semantics and providing a correct encoding of $\pi\mathsf{t}$-calculus into asynchronous π-calculus.

1 Introduction

Web Services technology intend to provide standard mechanisms for describing the interface and the services available on the web, as well as protocols for locating such services and invoking them (*cf.* WSDL standard [7]). A relevant feature, which is usually overlooked by Web Services, is the mechanism for their reuse when complex tasks are carried out. It is often the case, in business-to-business processes, to define new processes out of finer-grained subtasks that are likely available as Web Services. Therefore it is reasonable to forecast an extension of the Web Service standard, which supports the definition of complex services out of simpler ones – the so called *Web Services choreography*. Indeed, several proposals that describe Web Services choreography have been already set up: BPML [6] by BPMI.org, XLANG [18] and BizTalk [14] (a visual specification environment for XLANG) by Microsoft, WSFL [13] by IBM, BPEL4WS [9] by a consortium grouping BEA, IBM, Microsoft, and others), etc. The W3C Web Services Choreography Working Group is expected to present the Recommendation for the Web Services choreography specification for November 2003.

Most of these proposals use long-running transactions as a mechanism for describing *loosely-coupled* activities. On the contrary, traditional transactions in databases have been devised to compose *tightly-coupled* activities. These transactions are addressed by the keyword "ACID" to refer to the four properties that they guarantee – Atomicity, Consistency, Isolation, and Durability.

When the activities involved in a transaction are loosely-coupled, the ACID properties adapt badly. In particular, serializability (Isolation) requires that different activities have the same effect whether they are executed in sequence or in parallel. Usually, this is enforced by locking the resources used by each activity until the transaction commits. In the context of Web Services, the processes involved in a transaction may belong to different companies, and there is no chance to lock resources of other companies. Additionally, commercial transactions usually last long periods of time, even months, and it is not feasible and not reasonable to block resources so long. For similar reasons, Atomicity ("all or nothing"), and the two-phase commit protocol to enforce it, become impracticable in the case of long-term commercial transactions.

One immediately ends up in weakening the notion of rollback: in a business process, the provider might decide that rollback will not cancel all the operations carried out. The cancellation of an airplane booking, for instance, may lead to the payment of a fee; the interactions with non-transactional resources, which do not support an "absolute" mechanism of rollback, make failures extremely complicated, and to be dealt with ad-hoc ways. Overall, web transactions or, better, long-running transactions, miss serializability, due to the absence of locks on resources, and possess a lightweight notion of atomicity enforced by ad-hoc rollbacks, called *compensations*.

Despite the interest in long-running transactions, there is not yet a common agreement about their meaning. The above proposals for Web Services choreography have slightly different interpretations of long-running transactions, which may be hardly pointed out due to the informal nature of the documents, and to the complexity of parsing implementations (see the following subsection of related works).

In this paper we propose a formal approach to the description of long-running transactions. In particular, we consider Microsoft BizTalk, and the long-running transactions therein, and we define a formal model and an implementation. The other notions of long-running transactions could be defined as well in similar ways, and compared with the semantics of BizTalk. This is a considerable future work.

Long-running transaction in BizTalk have two associated activities: the *failure* process and the *compensation* process. There are two kinds of transactions: those without inner transactions and the others. The first case is simpler: if the transaction fails, the failure process is executed. In the second case, if a transaction with inner transactions fails, the compensations of the inner transactions must be executed before activating the failure process of the enclosing transaction. Namely, after failure, the compensations can be activated in any possible order, independently of the order in which the corresponding transactions completed. Therefore, the programmer must explicitly describe inter-dependencies among compensations, to avoid undesired schedules by the run-time system.

In our formal analysis we proceed as follows. We introduce a core language with BizTalk transactions, the $\pi\mathtt{t}$-calculus, and we define its operational semantics. It turns out that $\pi\mathtt{t}$-calculus is an extension of the asynchronous variant [4] of the π-calculus [15]. We then report some paradigmatic examples of long-term business activities. Afterwards we implement $\pi\mathtt{t}$-calculus in the asynchronous π-calculus, thus providing a further definition of the meaning of BizTalk transaction. Finally we demonstrate that this meaning conforms with the operational semantics of πt-calculus.

The choice to extend π-calculus with transactions, rather than other process calculi, is due to XLANG, which is the language implementing the orchestration services of BizTalk and whose definition has been strongly influenced by π-calculus. As regards our compilation, it is compositional, and therefore easily amenable to a distributed implementation. In this sense, our result may be read as a correctness proof of a distributed implementation of long-running transactions.

1.1 Related Works

Long-running transactions have been introduced in “data processing applications” in [12, 11], where they were called *saga*. Sagas are possibly nested processes with a monitor and a compensation. When a saga fails, if it doesn’t contain nested sagas, the compensation of the previously completed sagas are executed in the reverse order of composition. If the saga contains nested sagas, and a nested saga fails, the compensations of the successfully terminated sibling sagas are executed in the reverse order of commit. In this case, the monitor of the enclosing saga is notified of the abort.

Web services languages, such as WSFL [13], XLANG [18], and BPEL [9], support long-running transactions with flexible failure managements. In these languages, an aborted transaction raises an exception that is catched by a suitable programmable handler. When the handler is omitted, a failure activates the compensation of the completed transactions in the reverse order of commit. This mismatches with BizTalk semantics.

Other contributions to the definition of long-running transactions arise in “web transaction protocols”, which define models for orchestrating loosely-coupled web services by means of a defined set of transaction messages. In particular, the W3C Tentative Hold Protocol (THP) [16], uses an architecture with two actors: clients and resources. The clients send temptative holds to resources, requesting information about existing holds, cancelling holds, and retrieving stored informations. The resources use a programmable “rule engine entity” for their management. Another protocol is the OASIS Business Transaction Protocol (BTP) [10], which supports nesting of transactions as in BizTalk. OASIS introduces the notion of cohesion to bear different outcomes from participants to a transaction. In partiular, a transaction may succeed even if some of its inner transactions fail, provided that a minimal number of sub-transactions have succeeded.

1.2 Structure of the Paper

In Section 2 we define the syntax and operational semantics of the πt-calculus. In Section 3 we report examples of long-running transactions described in πt-calculus. In Section 4 we discuss the encoding of πt-calculus into asynchronous

π-calculus. Section 5 contains some conclusive remarks and a comparison with the related literature.

2 Syntax and Semantics of πt-Calculus

2.1 The Syntax

The syntax of the πt-calculus uses a countable set of names $\mathcal{N}$ ranged over by x, y, u, v, ... Tuples of names are written $u_i^{i \in 1..p}$ or simply $\widetilde{u}$. In order to support process constant definitions, we assume given a set of process constants ranged over by K, K', ...

A process (or an agent) in πt-calculus is defined by the following syntax:

$$\begin{array}{llll}
P ::= & \mathsf{done} & \quad & \text{success} \\
 & \mid \mathsf{abort} & & \text{error} \\
 & \mid \overline{x}\langle \widetilde{u} \rangle & & \text{output} \\
 & \mid x(\widetilde{u}).P & & \text{input} \\
 & \mid P \mid P & & \text{parallel} \\
 & \mid P ; P & & \text{sequence} \\
 & \mid (x)P & & \text{new} \\
 & \mid K(\widetilde{u}) & & \text{invocation} \\
 & \mid \mathsf{t}(P,\, P,\, P,\, P) & & \text{transaction}
\end{array}$$

The new operator $(x)P$ and the input prefix operator $x(\widetilde{u}).P$ are binders for the names x and $\widetilde{u}$, respectively. We omit the standard definitions of *free variables* and *bound variables* of processes, noted $\mathsf{fv}(\cdot)$ and $\mathsf{bv}(\cdot)$, respectively. For each process constant K we assume given a single constant definition $K(\widetilde{u}) \stackrel{def}{=} P$ where $\widetilde{u}$ is a sequence of pairwise different names and P is a process. If the list of parameters is empty, we omit the surrounding parenthesis, e.g. we use K instead of $K()$.

The processes output, parallel, new, and invocation, are as usual [15]. To enlight the notation we sometime write $(x_1 \ldots x_n)P$ instead of $(x_1) \ldots (x_n)P$. As usual, we also abbreviate the parallel of P_i for $i \in I$, where I is a finite set, with $\prod_{i \in I} P_i$. The processes done and abort do nothing except manifesting a successful or erroneous termination – to be used inside a transactional context. The input process has an explicit continuation. The sequence process $P; Q$ forces a temporal order between the two operands: process Q will be activated only after successful completion of P. It is worth to notice that our sequential composition operator, even if similar to other operators of the tradition of process algebras (see e.g. the operator $\cdot$ of ACP [2]), is new due to the presence of two different kinds of process termination, represented by the processes done and abort respectively. As we will discuss in the following, these two processes are treated differently when they appear as first operand of the sequential composition operator.

The process $\mathsf{t}(P,\, F,\, B,\, C)$ defines a transaction; in particular, P is the main process – the *body* –, to be executed in a transactional way; F and B are, respectively, the *failure manager* and the *failure bag*, to be executed if a failure occurs; and C is the *compensation*, to be executed in case an enclosing transaction fails. Only the main process, the failure manager, and the compensation are considered in BizTalk specifications. The further argument – the "failure bag" – is a repository to store the compensations of the enclosed transactions that complete while the enclosing transaction is still running. This repository has been added for semantic reasons, to describe the behaviour of (successfully) terminating transactions.

Process contexts are processes with a hole inside. Contexts are ranged over by $\mathbf{C}[\,]$ and are defined by the following grammar:

$$\begin{aligned}\mathbf{C}[\,] ::= {} & [\,] \\ & \mid \mathbf{C}[\,] \mid P \\ & \mid \mathbf{C}[\,]; P \\ & \mid (x)\mathbf{C}[\,] \\ & \mid \mathsf{t}(\mathbf{C}[\,],\, P,\, P,\, P)\end{aligned}$$

We use $\mathbf{C}[P]$ to denote the process obtained by substituting the hole inside $\mathbf{C}[\,]$ with the process P.

2.2 Structural Congruence

Structural congruence, written $\equiv$, equates all processes we will never want to distinguish for any semantic reason.

Definition 1. Structural congruence, *written $\equiv$, is the least congruence over processes, which contains α-renaming, and the following equations:*

$$\begin{array}{lll} P \mid Q \equiv Q \mid P & P \mid (Q \mid R) \equiv (P \mid Q) \mid R & \\ (P;Q);R \equiv P;(Q;R) & & \\ (x)(y)P \equiv (y)(x)P & (x)(P \mid Q) \equiv P \mid (x)Q & x \notin \mathsf{fv}(P) \\ (x)(P;Q) \equiv (x)P;Q \quad x \notin \mathsf{fv}(Q) & & \end{array}$$

$$(\overline{x}\langle\tilde{u}\rangle \mid P);Q \equiv \overline{x}\langle\tilde{u}\rangle \mid P;Q$$

$$\begin{array}{ll} \mathsf{done} \mid P \equiv P & \mathsf{abort} \mid \mathsf{abort} \equiv \mathsf{abort} \\ \mathsf{done}; P \equiv P & \mathsf{abort}; P \equiv \mathsf{abort} \end{array}$$

$$K(\widetilde{v}) \equiv P\{^{\widetilde{v}}/_{\widetilde{u}}\} \qquad \text{if } K(\widetilde{u}) \overset{def}{=} P$$

$$\begin{array}{ll} \mathsf{t}((x)P,\, F,\, B,\, C) \equiv (x)\mathsf{t}(P,\, F,\, B,\, C) & x \notin \mathsf{fv}(F) \cup \mathsf{fv}(B) \cup \mathsf{fv}(C) \\ \mathsf{t}(\overline{x}\langle\widetilde{u}\rangle \mid P,\, F,\, B,\, C) \equiv \overline{x}\langle\widetilde{u}\rangle \mid \mathsf{t}(P,\, F,\, B,\, C) & \\ (\mathsf{t}(\mathsf{done},\, F,\, B,\, C) \mid P);P' \equiv \mathsf{t}(\mathsf{done},\, F,\, B,\, C) \mid (P;P') & \end{array}$$

The first group of equations is almost standard: let us discuss the not standard ones. Notice that the rule $(x)(P;Q) \equiv (x)P;Q$ if $x \notin \mathsf{fv}(Q)$, is necessary (in combination with α–renaming) to allow restrictions to float at top level. The equation $(\overline{x}\langle\tilde{u}\rangle \mid P);Q \equiv \overline{x}\langle\tilde{u}\rangle \mid P;Q$ floats outputs outside sequences, since they have no continuation. Equations $\mathsf{done} \mid P \equiv P$ and $\mathsf{done}; P \equiv P$ specify that done is the identity of parallel and sequence; $\mathsf{abort} \mid \mathsf{abort} \equiv \mathsf{abort}$ states that a process is aborted when all its parallel components are aborted; $\mathsf{abort}; P \equiv \mathsf{abort}$ specifies that an aborted process is such, regardless of the continuation. The equation $\mathfrak{t}(\overline{x}\langle\widetilde{u}\rangle \mid P, F, B, C) \equiv \overline{x}\langle\widetilde{u}\rangle \mid \mathfrak{t}(P, F, B, C)$ allows to move outputs from inside a transaction to the outside environment and vice-versa. The intended semantics is the following. If a transactional process emits a message, this message traverses the transaction boundary, until reaching the corresponding input. In case the transaction fails, recoveries for this output may be detailed inside the processes F and B. The equation $(\mathfrak{t}(\mathsf{done}, F, B, C) \mid P); P' \equiv \mathfrak{t}(\mathsf{done}, F, B, C) \mid (P; P')$ allows to float successfully terminated transactions outside parallels and sequences. We observe that $\mathfrak{t}(\mathsf{done}, F, B, C)$ is not equal to done because, if an enclosing transaction fails, then the compensation C must be fired to accomodate possible inconsistencies (see the next rule (T-DONE)).

2.3 The Reduction Relation

The reduction relation of $\pi\mathfrak{t}$-calculus is the least relation satisfying the rules in Table 1,

Table 1. The reduction rules of $\pi\mathfrak{t}$-calculus

(RED)	$\overline{x}\langle\widetilde{v}\rangle \mid x(\widetilde{u}).P \;\rightarrow\; P\{^{\widetilde{v}}/_{\widetilde{u}}\}$
(T-DONE)	$\mathfrak{t}(\mathfrak{t}(\mathsf{done}, F, B, C) \mid P, F', B', C') \;\rightarrow\; \mathfrak{t}(P, F', B' \mid C, C')$
(T-ABORT)	$\mathfrak{t}(\mathsf{abort}, F, B, C) \rightarrow B; F$
(CONTEXT)	$\dfrac{P \rightarrow P'}{\mathbf{C}[P] \rightarrow \mathbf{C}[P']}$
(LIFT)	$\dfrac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q \equiv Q'}{P \rightarrow Q}$

The rules (T-DONE) and (T-ABORT) deserve some discussion. (T-DONE) models the successful completion of a transaction $\mathfrak{t}(\mathsf{done}, F, B, C)$. In this case, the compensation C must be recorded in the failure bag of the enclosing transaction, if any, to account for possible failures of the latter. If the outer transaction fails, rule (T-ABORT) specifies that the failure manager must be executed *after* the compensation of every enclosed transaction.

2.4 Comparison with the π-Calculus

It is interesting to observe that the $\pi\mathfrak{t}$-calculus is essentially an extension of the π-calculus. Indeed, we can obtain the latter from the former simply by eliminating the sequence operator $P; Q$, the transactional process $\mathfrak{t}(\mathsf{done}, F, B, C)$, the process abort, and by interpreting the process done as the empty process $\mathsf{0}$ of the π-calculus. Actually it is also possible to encode the former into the latter, and we will study the encoding in section 4.

3 Examples

In the examples we use an extension of the calculus that comprises conditionals, boolean values, and boolean variables. Namely, we consider the operator:

$\mathsf{if}\ (a = k)\ \mathsf{then}\ P\ \mathsf{else}\ Q$

where $k = 0$ or $k = 1$, and a is a boolean variable. The semantics of the conditionals is the standard one.

3.1 Authentication

The first example describes a server that authenticates its clients exploiting a certification authority. The are four actors: the client (*Client*), the server (*Server*), the certification authority (*Auth*), and a law authority (*Law*) used to notify abuses.

We focus on the behaviour of the server: on reception of a request (which includes the identity of the client *id*, and its certificate *cert*), the server asks the certification authority to check the validity of the received certificate. Therefore, the server waits for an answer, that may be either *1* or *0*, depending on success or failure, respectively. In the case of success the server executes the required activity, on the contrary it communicates the abuse to the law authority. The following channels are used:

– *req* is the channel between *Client* and *Server*;
– *check* is the channel between *Server* and *Auth*;
– *resp* is a channel created by the *Main* process and passed to the *Server*: this channel is used to communicate the result (success or failure) of the certificate check (the same technique will be used also in the following examples);
– *ntf* is the channel between *Server* and *Law*;
– *abuse* is a local channel used to store data concerned with the abuse of certificates.

We are now in place to specify the above process in $\pi\mathsf{t}$-calculus:

$$\begin{array}{lll}
\mathit{Server} & = & (\mathit{abuse})\mathsf{t}(\mathit{Main}, \mathit{Fail}, \mathsf{done}, \mathsf{done}) \\
\mathit{Main} & = & \mathit{req}(\mathit{task}, \mathit{id}, \mathit{cert}). \\
 & & \quad (\mathit{resp})(\overline{\mathit{check}}\langle \mathit{id}, \mathit{cert}, \mathit{resp}\rangle \mid \\
 & & \qquad \mathit{resp}(a).\mathsf{if}\ (a = 1)\ \mathsf{then}\ \mathit{Execute}(\mathit{task})\ \mathsf{else}\ (\overline{\mathit{abuse}}\langle \mathit{id}, \mathit{cert}\rangle \mid \mathsf{abort}) \\
 & & \qquad) \\
\mathit{Fail} & = & \mathit{abuse}(\mathit{id}, \mathit{cert}).\overline{\mathit{ntf}}\langle \mathit{id}, \mathit{cert}\rangle
\end{array}$$

where *Execute* is a program constant (that we leave unspecified) representing the execution of the task. It is important to note that the transaction is used here merely as a facility for exception management.

3.2 Flight or Train Booking

This second example has been introduced to discuss that, when an abort process is reached, the failure process is not activated immediately, but the termination of parallel threads is waited. This is particularly useful in transactions, composed by concurrent activities, in order to give a chance to any activity to terminate successfully its task even if some other fails.

We consider a travel agency (*Travel*) that books a certain number of flights: each reservation is managed by a process *Reserve*. The reservations may succeed or fail. At the end of all the reservation subtasks, if at least one of them has failed, a failure process is started which reserves trains instead of failed flight reservations.

We exploit the following names:

- *bookF* is a channel shared between the travel agency and the flight company;
- *dest* and *resp* are names fresh for each process *Reserve*: *dest* indicates the destination while *resp* is the channel to be used to indicate the success or failure of the reservation (using *1* or *0*, respectively);
- *train* is a channel used to store, in the case of flight reservation failure, the request for an alternative train reservation;
- *bookT* is a channel shared between the travel agency and the train company.

We are now in place to specify the booking process:

$$\begin{aligned}
&\mathit{Travel} &&= \mathsf{t}(\mathit{Flight},\ \mathit{Train},\ \mathsf{done},\ \mathsf{done})\\
&\mathit{Flight} &&= \mathit{Reserve} \mid \mathit{Reserve} \mid \ldots \mid \mathit{Reserve}\\
&\mathit{Reserve} &&= (\mathit{dest})(\mathit{resp})(\ \overline{\mathit{bookF}}\langle \mathit{dest}, \mathit{resp}\rangle \mid\\
&&&\qquad\qquad \mathit{resp}(a).\mathsf{if}\ (a = 1)\ \mathsf{then}\ \mathsf{done}\ \mathsf{else}\ (\overline{\mathit{train}}\langle \mathit{dest}\rangle \mid \mathsf{abort})\ )\\
&\mathit{Train} &&= \mathit{train}(\mathit{dest}).(\overline{\mathit{bookT}}\langle \mathit{dest}\rangle \mid \mathit{Train})
\end{aligned}$$

As described above, the failure of one of the *Reserve* processes does not influence the concurrent activities; the failure process starts only on termination of all the *Reserve* processes. This is ensured by the rule (T-ABORT) that activates the failure bag and the failure manager only when the main process is (structurally congruent to) the process abort.

3.3 Flight and Hotel Booking

In this third example we decribe a transaction (*Journey*) composed of the sequence of two transactions: the first books a return flight ticket, the second reserves a hotel room for the nights between the arrival and the departure dates.

The first transaction has a compensation process which is responsible for cancelling the flight reservation. We use the two process constants *Ticket(dest,a,d)* and *Room(dest,a,d)* to represent the two transactions: *dest* is the destination while *a* and *d* are the arrival and departure dates, respectively. We consider the existence of three channels:

- *bookF* and *bookH* used to ask for a flight or a room booking, respectively. In both cases, the request may either succeed or fail;
- *cancelF* is used to ask for the cancellation of a flight reservation.

The booking requests may either succed or fail: also in this case we use the boolean values *1* and *0* to denote these two possible outcomes, respectively.

We are now in place to present the formal specification of the *Journey* process:

$$\begin{array}{lll} \mathit{Journey} & = \mathsf{t}(\mathit{Ticket}(\mathit{dest}, a, d); \mathit{Room}(\mathit{dest}, a, d), \mathsf{done}, \mathsf{done}, \mathsf{done}) & \\ \mathit{Ticket}(\mathit{dest}, a, d) & = (\mathit{resp})\ \mathsf{t}(\ \overline{\mathit{bookF}}\langle \mathit{dest}, a, d, \mathit{resp}\rangle & \\ & \qquad \mid \mathit{resp}(\mathit{ack}).\mathsf{if}\ (\mathit{ack} = 1)\ \mathsf{then\ done\ else\ abort}, & \\ & \qquad \mathsf{done}, \mathsf{done}, \overline{\mathit{cancelF}}\langle \mathit{dest}, a, d\rangle\ ) & \\ \mathit{Room}(\mathit{dest}, a, d) & = (\mathit{resp})\ \mathsf{t}(\ \overline{\mathit{bookH}}\langle \mathit{dest}, a, d, \mathit{resp}\rangle & \\ & \qquad \mid \mathit{resp}(\mathit{ack}).\mathsf{if}\ (\mathit{ack} = 1)\ \mathsf{then\ done\ else\ abort}, & \\ & \qquad \mathsf{done}, \mathsf{done}, \mathsf{done}\ ) & \end{array}$$

Notice that an equivalent behaviour is obtained placing the flight cancellation request $\overline{\mathit{cancelF}}\langle \mathit{dest}, a, d\rangle$ as failure of the second transaction. However, the process specification we have reported supports better modularity because all the activities related to flight reservation, as well as cancellation, are all inside the same transaction.

4 The Encoding of the $\pi\mathsf{t}$-Calculus into the Asynchronous π-Calculus

In this section we demonstrate that $\pi\mathsf{t}$-calculus may be encoded into the asynchronous π-calculus. We recall that the latter is indeed a subcalculus of the former (see section 2.4), therefore we avoid any redefinition. The encoding is a partial function $[\![-]\!]^{-;-}_{-;-}$. The process $[\![P]\!]^{z,w}_{x,y}$, such that $x, y, z, w \notin \mathsf{fv}(P)$, uses the channels x, y, z and w respectively to signal successful termination without compensations, erroneous termination without compensations, successful termination with compensation and erroneous termination with compensation. The termination of P could have compensations if there are transactional contexts defined in it.

In the definition below we use the constants M and N^{zw}_{xy} defined as follows:

$$\begin{array}{l} M(c, c', c'') = c(x, y, z, w).\ (x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R)(\\ \qquad\qquad \overline{c'}\langle x_L y_L z_L w_L\rangle \mid \overline{c''}\langle x_R y_R z_R w_R\rangle \\ \qquad\qquad \mid N^{zw}_{xy}(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) \\ \qquad) \end{array}$$

$$\begin{array}{l} N^{zw}_{xy}(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) = \\ \qquad x_L().(x_R().\overline{x}\langle\rangle \mid y_R().\overline{y}\langle\rangle \mid z_R(c_R).\overline{z}\langle c_R\rangle \mid w_R(c_R).\overline{w}\langle c_R\rangle) \\ \qquad \mid y_L().(x_R().\overline{y}\langle\rangle \mid y_R().\overline{y}\langle\rangle \mid z_R(c_R).\overline{w}\langle c_R\rangle \mid w_R(c_R).\overline{w}\langle c_R\rangle) \\ \qquad \mid z_L(c_L).(\ x_R().\overline{z}\langle c_L\rangle \mid y_R().\overline{w}\langle c_L\rangle \\ \qquad\qquad \mid z_R(c_R).(c''')(\overline{z}\langle c'''\rangle \mid M(c''', c_L, c_R)) \\ \qquad\qquad \mid w_R(c_R).(c''')(\overline{w}\langle c'''\rangle \mid M(c''', c_L, c_R))) \\ \qquad \mid w_L(c_L).(x_R().\overline{w}\langle c_R\rangle \mid y_R().\overline{w}\langle c_R\rangle \\ \qquad\qquad \mid z_R(c_R).(c''')(\overline{w}\langle c'''\rangle \mid M(c''', c_L, c_R)) \\ \qquad\qquad \mid w_R(c_R).(c''')(\overline{w}\langle c'''\rangle \mid M(c''', c_L, c_R))) \end{array}$$

The purpose of M and N^{zw}_{xy} is discussed next. Actually we focus on M, because N^{zw}_{xy} is similar. The process $M(c, c', c'')$ multiplexes the compensation request coming from c towards c' and c'', which are the channels for invoking compensations of two parallel subprocesses, let us call them L and R. The subprocesses L and R may return four different kinds of signals, namely x_L, y_L, z_L, w_L and x_R, y_R, z_R, w_R, with sixteen different combinations. The interesting combinations are when L returns on z_L or w_L, and R returns on z_R or w_R, namely when L and R return a compensation to activate in case of failure. In these cases a new multiplexer must be triggered, henceforth the recursive invocation of M. We remark that, for the correctness of M and N, it is necessary there are exactly two messages: one on channels x_L, y_L, z_L, w_L and the other on x_R, y_R, z_R, w_R.

Definition 2. *The process* $[\![P]\!]^{z,w}_{x,y}$*, such that* $x, y, z, w \notin \mathsf{fv}(P)$*, is defined by the equations below (the definition of* M_P *is the last one). We always assume that new names introduced by the encoding never clash with free names of the encoded process.*

$$[\![\mathit{done}]\!]^{z,w}_{x,y} = \overline{x}\langle\rangle \qquad [\![\mathit{abort}]\!]^{z,w}_{x,y} = \overline{y}\langle\rangle$$

$$[\![\overline{u}\langle\widetilde{v}\rangle]\!]^{z,w}_{x,y} = \overline{u}\langle\widetilde{v}\rangle \mid \overline{x}\langle\rangle \qquad [\![u(\widetilde{v}).P]\!]^{z,w}_{x,y} = u(\widetilde{v}).[\![P]\!]^{z,w}_{x,y} \qquad (x, y, z, w \notin \widetilde{v})$$

$$[\![(u)P]\!]^{z,w}_{x,y} = (u)[\![P]\!]^{z,w}_{x,y} \qquad (u \notin x, y, z, w)$$

$$\begin{aligned}
[\![P; Q]\!]^{z,w}_{x,y} = (x', y', z', w')(\ & [\![P]\!]^{z',w'}_{x',y'} \\
& \mid x'().[\![Q]\!]^{z,w}_{x,y} \\
& \mid y'().\overline{y}\langle\rangle \\
& \mid z'(c).M_Q(c, z, w) \\
& \mid w'(c).\overline{w}\langle c\rangle \\
)&
\end{aligned}$$

$$\begin{aligned}
[\![P \mid Q]\!]^{z,w}_{x,y} = (x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R)(& \\
& [\![P]\!]^{z_L,w_L}_{x_L,y_L} \mid [\![Q]\!]^{z_R,w_R}_{x_R,y_R} \\
& \mid N^{zw}_{xy}(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) \\
)&
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{t}(P,\ F,\ B,\ C)]\!]^{z,w}_{x,y} = (x_1, y_1, z_1, w_1)(& \\
& [\![P]\!]^{z_1,w_1}_{x_1,y_1} \\
& \mid x_1().(c)(\overline{z}\langle c\rangle \mid c(x_1 y_1 z_1 w_1).[\![C]\!]^{z_1,w_1}_{x_1,y_1}) \\
& \mid y_1().[\![B; F]\!]^{z,w}_{x,y} \\
& \mid z_1(c).(c')(\overline{z}\langle c'\rangle \mid c'(x_1 y_1 z_1 w_1).[\![C]\!]^{z_1,w_1}_{x_1,y_1}) \\
& \mid w_1(c).(x', y', z', w') \\
& \qquad (x_L, y_L, z_L, w_L) \\
& \qquad (x_R, y_R, z_R, w_R)(\\
& \qquad\quad \overline{c}\langle x_L y_L z_L w_L\rangle \mid [\![B]\!]^{z_R,w_R}_{x_R,y_R} \\
& \qquad\quad \mid N^{z'w'}_{x'y'}(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) \\
& \qquad\quad \mid x'().[\![F]\!]^{z,w}_{x,y} \\
& \qquad\quad \mid y'().\overline{y}\langle\rangle \\
& \qquad\quad \mid z'(c').M_F(c', z, w) \\
& \qquad\quad \mid w'(c').\overline{w}\langle c'\rangle \\
& \qquad) \\
)&
\end{aligned}$$

$$[\![K(u_1,\ldots,u_n)]\!]^{z;w}_{x,y} = K_\pi(u_1,\ldots,u_n,x,y,z,w)$$

$$K_\pi(v_1,\ldots,v_n,x,y,z,w) = [\![P]\!]^{z;w}_{x;y} \qquad (\textit{assuming}\ \ K(v_1,\ldots,v_n) \overset{def}{=} P)$$

where the definition of the constant M_P is the following:

$$\begin{aligned} M_P(c,z,w) = (x',y',z',w')(\ & [\![P]\!]^{z',w'}_{x',y'} \\ & \mid x'().\overline{z}\langle c\rangle \\ & \mid y'().\overline{w}\langle c\rangle \\ & \mid z'(c').(c'')(\overline{z}\langle c''\rangle \mid M(c'',c,c')) \\ & \mid w'(c').(c'')(\overline{w}\langle c''\rangle \mid M(c'',c,c')) \\ &) \end{aligned}$$

Let us clarify the behaviour of $[\![\cdot]\!]^{z,w}_{x,y}$, according to the shape of the argument.

If the argument is done, a signal on x is emitted, meaning the successful termination without compensations.

If the argument is $\overline{u}\langle\widetilde{v}\rangle$, the successful termination signal is in addition with the message on u.

If the argument is abort, a signal on y is emitted, representing the failure without compensation.

If the argument is $u(\tilde{v}).Q$ or $(x)Q$ the encoding is homomorphic and successes and failures are passed to the encoding of Q.

If the argument is $P; Q$, the encoding of P is executed and, in case of successful completion, the encoding of Q is performed afterwards (we observe that the encoding of Q is always underneath an input). In case there are compensations, the process M_Q is called, as discussed above.

If the argument is $P \mid Q$, the encodings of P and Q are performed in parallel and the results are collected by the agent N.

If the argument is $\mathsf{t}(P,\ F,\ B,\ C)$, the encoding of the body P is executed. There are several cases. In case of success (with or without compensations), the process $[\![\mathsf{t}(P,\ F,\ \mathsf{done},\ C)]\!]^{z;w}_{x,y}$ emits on z the compensation triggering the encoding of C (inner compensations are discarded). In case of failure without compensations, we must perform the encoding of the agent B *before* the failure manager F. In case of failure with compensations then the failure manager F must be executed *after* the other compensations. Remark that this last case is very similar to the sequential composition.

If the argument is $K(u_1,\ldots,u_n)$, we use a twin constant K_π that carries four additional arguments, namely the channels for signaling success and failure. The definition of K_π is the expected encoding of the definition of K.

4.1 Correctness of the Encoding

We assess the correctness of the encoding of $\pi\mathsf{t}$-calculus with respect to the semantics defined in section 2. Since the previous encoding yields π-calculus agents that show up several deadlocked subprocesses (see the definitions of sequence, parallel, or transaction), we require an extensional semantics that is, as far as the π-calculus is considered, insensitive to deadlocked processes. To this aim we introduce *(weak) barbed equivalence* [17].

Definition 3. *Let $P \downarrow x$ be the least relation satisfying the rules below.*

$$
\begin{array}{ll}
\overline{x}\langle\widetilde{u}\rangle \downarrow x & \\
P \mid Q \downarrow x & \textit{if } P \downarrow x \textit{ or } Q \downarrow x \\
P; Q \downarrow x & \textit{if } P \downarrow x \\
(y)P \downarrow x & \textit{if } P \downarrow x \textit{ and } x \neq y \\
\mathsf{t}(P,\, F,\, B,\, C) \downarrow x & \textit{if } P \downarrow x \\
P \downarrow x & \textit{if } Q \downarrow x \textit{ and } P \equiv Q
\end{array}
$$

If $P \downarrow x$ we say that P has a barb *on x.*

Notice that, this definition of barb is different from the standard one [17] because we are closing the relation by structural equivalence. The usual definition by induction on the syntax is hard in our case because of the sequence operator. We remark that the above barb does not allow to discriminate between the processes done and abort, even though a transaction context behaves differently when filled with them. Actually this is not an issue since the barbed congruence semantics (not discussed in this paper) is enough discriminating to separate done and abort.

Definition 4. *A* (weak) barbed bisimulation *is a symmetric binary relation $\mathcal{S}$ between agents such that $P \mathrel{\mathcal{S}} Q$ implies:*

1. *If $P \to P'$ then $Q \to^* Q'$ and $P' \mathrel{\mathcal{S}} Q'$.*
2. *If $P \downarrow x$ for some x, then $Q \to^* Q'$ and $Q' \downarrow x$.*

P is barbed bisimilar to Q, written $P \stackrel{\bullet}{\approx} Q$, if there exists some barbed bisimulation $\mathcal{S}$ such that $P \mathrel{\mathcal{S}} Q$.

For instance, $\mathsf{done} \stackrel{\bullet}{\approx} \mathsf{abort} \stackrel{\bullet}{\approx} (x)\overline{x}\langle u\rangle$. The definitions of barb and barbed equivalence coincide with those of π-calculus when processes are restricted to π-calculus ones.

The correctness of the encoding $[\![P]\!]^{z,w}_{x,y}$ is formalized by the following result.

Theorem 1. *Let P be a $\pi\mathsf{t}$-calculus process. Then*

1. *$P \downarrow u$ if and only if $[\![P]\!]^{z,w}_{x,y} \downarrow u$ and $u \notin \{x, y, z, w\}$.*
2. *If $P \to Q$ then $[\![P]\!]^{z,w}_{x,y} \to^* \stackrel{\bullet}{\approx} [\![Q]\!]^{z,w}_{x,y}$ (provided that x, y, z, w do not clash with $\mathsf{fv}(P)$ and $\mathsf{fv}(Q)$).*
3. *If $[\![P]\!]^{z,w}_{x,y} \to Q$ then there is R such that $P \to^* R$ and $Q \stackrel{\bullet}{\approx} [\![R]\!]^{z,w}_{x,y}$.*

Proof. (Sketch) (1) Since "$\downarrow$" encompasses structural equivalence, one ends up at demonstrating that, if $P \equiv Q$ and $[\![P]\!]^{z,w}_{x,y} \downarrow u$, then $[\![Q]\!]^{z,w}_{x,y} \downarrow u$. This is mostly a straightforward analysis, except for the structural rules $(\overline{x}\langle\tilde{u}\rangle \mid P); Q \equiv \overline{x}\langle\tilde{u}\rangle \mid P; Q$ and $(\mathsf{t}(\mathsf{done},\, F,\, B,\, C) \mid P); P' \equiv \mathsf{t}(\mathsf{done},\, F,\, B,\, C) \mid (P; P')$. Both cases follow by the definition of the encoding and by a careful analysis whether P is amenable to done or not.

(2) We analyze the basic reductions. Among them, the difficult case is (T-DONE) because of the management of the failure bag. Let us discuss this case in detail. On one side we have:

$$\mathsf{t}(\mathsf{t}(\mathsf{done},\, F,\, B,\, C) \mid P,\, F',\, B',\, C') \;\to\; \mathsf{t}(P,\, F',\, B' \mid C,\, C')$$

On the other side we have:

$$
\begin{array}{l}
[\![\mathsf{t}(\mathsf{t}(\mathsf{done},\, F,\, B,\, C) \mid P,\, F',\, B',\, C')]\!]^{z,w}_{x,y} \\
\quad = (x_1, y_1, z_1, w_1)([\![\mathsf{t}(\mathsf{done},\, F,\, B,\, C) \mid P]\!]^{z_1,w_1}_{x_1,y_1} \mid T^{xyzw}_{x_1y_1z_1w_1}(F', B', C')) \\
\qquad\qquad\qquad\qquad\qquad (T^{xyzw}_{x_1y_1z_1w_1}(F', B', C') \text{ is the manager of transaction} \\
\qquad\qquad\qquad\qquad\qquad \text{that may be grabbed from definition 2)} \\[1em]
\quad = (x_1, y_1, z_1, w_1)(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R)(\\
\qquad [\![\mathsf{t}(\mathsf{done},\, F,\, B,\, C)]\!]^{z_L,w_L}_{x_L,y_L} \mid [\![P]\!]^{z_R,w_R}_{x_R,y_R} \mid N^{z_1,w_1}_{x_1,y_1} \\
\qquad (x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) \mid T^{xyzw}_{x_1y_1z_1w_1}(F', B', C')) \\[1em]
\quad \rightarrow (x_1, y_1, z_1, w_1)(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R)(\\
\qquad (c)(\overline{z_L}\langle c\rangle \mid c(x'y'z'w').[\![C]\!]^{z',w'}_{x',y'} \mid [\![P]\!]^{z_R,w_R}_{x_R,y_R} \\
\qquad \mid N^{z_1,w_1}_{x_1,y_1}(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) \mid T^{xyzw}_{x_1y_1z_1w_1}(F', B', C')) \\[1em]
\quad \rightarrow (x_R, y_R, z_R, w_R)([\![P]\!]^{z_R,w_R}_{x_R,y_R} \mid \\
\qquad\quad (x_1, y_1, z_1, w_1)(c)(x_R().\overline{z_1}\langle c\rangle \mid y_R().\overline{w_1}\langle c\rangle \\
\qquad\qquad \mid z_R(c_R).(c''')(\overline{z_1}\langle c'''\rangle \mid M(c''', c, c_R)) \\
\qquad\qquad \mid w_R(c_R).(c''')(\overline{w_1}\langle c'''\rangle \mid M(c''', c, c_R)) \\
\qquad\qquad \mid c(x'y'z'w').[\![C]\!]^{z',w'}_{x',y'} \mid T^{xyzw}_{x_1y_1z_1w_1}(F', B', C')))
\end{array}
$$

Therefore we are reduced to prove that

$$
\begin{array}{l}
(x_1, y_1, z_1, w_1)(c)(x_R().\overline{z_1}\langle c\rangle \mid y_R().\overline{w_1}\langle c\rangle \\
\quad \mid z_R(c_R).(c''')(\overline{z_1}\langle c'''\rangle \mid M(c''', c, c_R)) \\
\quad \mid w_R(c_R).(c''')(\overline{w_1}\langle c'''\rangle \mid M(c''', c, c_R)) \\
\quad \mid c(x'y'z'w').[\![C]\!]^{z',w'}_{x',y'} \mid T^{xyzw}_{x_1y_1z_1w_1}(F', B', C'))
\end{array}
\quad \dot{\approx}\; T^{xyzw}_{x_Ry_Rz_Rw_R}(F', B' \mid C, C')
$$

This follows by a close inspection of all the possible cases.

(3) The proof consists of picking some representative π-calculus processes of the evaluation of $[\![P]\!]$, and demonstrating that intermediate processes are barbed bisimilar to the representatives. Representatives are processes where no "bureaucratic reactions" is possible (these are the reactions due to the encoding). Representatives are proved bisimilar to encodings of $\pi\mathsf{t}$-calculus processes in a way similar to that reported for the case (2).

This theorem is the basic result to relate barbed bisimulation in π-calculus and in $\pi\mathsf{t}$-calculus. It is well-known that this equivalence is not very interesting because its discriminating power is weak. Nevertheless, our intended application of such result is to infer barbed bisimulation congruence of $\pi\mathsf{t}$-calculus – which, on the contrary, is an interesting semantics – from barbed bisimulation congruence of the encoded agents in π-calculus. This is a considerable result for our calculus that requires a weighty effort (e.g. we should exploit an equivalent labelled semantics characterizations [17]) that we leave for future work.

5 Conclusions

Long-running transactions have recently received a renewed interest with the advent of Web Services-based business interactions. Indeed, these transactions are considered a valuable tool for business process modeling. In this paper, we have formalized and studied the notion of long-running transactions incorporated in Microsoft BizTalk [14], a visual environment for business process modeling.

We notice the absence, to the best of our knowledge, of formal specifications and analysis of transactions in Web Services-based business process modeling. The unique published work we are aware of is [5], devoted to the investigation of ACID (short-lived) transactions in the context of BizTalk.

As future work, we plan to investigate more complex mechanisms for composing transactions; in particular, in our calculus a transaction obtained as parallel composition of sub-transactions waits for the termination of all these transactions before terminating itself. Other interesting composition operators, see e.g. the *pick* constructur of XLANG [18], allows for the execution of one subcomponent only, chosen according to the occurrence of some specific event.

Acknowledgement

We thank the anonymous referees for their fruitful comments and Greg Meredith and Tony Hoare for the insightful discussions about the topics treated in this paper. This paper has been partially supported by Italian Ministry of University – MURST 40% – Progetto NAPOLI, and by a grant by Microsoft Research.

References

1. A. Ankolekar, M. Bursten, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, N. Paolucci, T. Payne, K. Sycara, H. Zeng, (2001). DAML-S: Semantic Markup for Web Services. In International Semantic Web Working Symposium 2001.
2. J.C.M. Baeten and W.P. Weijland. Process algebra. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press 1990.
3. T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, D. Fensel, R. Fikes, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, L.A. Stein. DAML+OIL. [http://www.daml.org/2001/03/daml+oil-index], 2001.
4. G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, INRIA, Sophia–Antipolis,1992.
5. R. Bruni, C. Laneve, U. Montanari. Orchestrating Transactions in Join Calculus. In proc. of CONCUR'02, LNCS 2421, pp. 321 - 337, 2002.
6. Business Process Modeling Language (BPML). [www.bpmi.org]
7. E. Christensen, F. Curbera, G. Meredith and S. Weerawarana. Web Services Description Language (WSDL 1.1). [www.w3.org/TR/wdsl], W3C, Note 15, 2001.
8. T.D.S. Coalition. DAML-S: Web service description for the semantic web. In Proc. of ISWC, 2002.

9. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte and S. Weerawarana. Business Process Execution Language for Web Services (BPEL4WS 1.0). [http://www-106.ibm.com/developerworks/webservices/library/ ws-bpel/], 2002.
10. S. Dalal, S. Temel, M. Little, M. Potts, J, Webber. Coordinating Business Transactions on the Web. IEEE Internet Computing, January-February 2003.
11. H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem. Modeling Long-Running Activities as Nested Sagas. IEEE Bulletin of the Technical Committee on Data Engineering, 14 (1), 1991.
12. H. Garcia-Molina, K. Salem. Sagas. In Proc. of SIGMOD International Conference on Management of Data, pp. 249–259, 1987.
13. F. Leymann. Web Services Flow Language (WSFL 1.0). [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], Member IBM Academy of Technology, IBM Software Group, 2001.
14. Microsoft BizTalk Server. [http://www.microsoft.com/biztalk/default.asp], Microsoft Corporation.
15. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77. Academic Press, 1992.
16. J. Roberts, K. Srinivasan. Tentative Hold Protocol Part 1: White Paper. W3C Note 28 November 20001. [http://www.w3.org/TR/tenthold-1/]
17. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.
18. S. Thatte. XLANG: Web Services for Business Process Design. [http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.