Dynamic Recovering of Long Running Transactions

Cátia Vaz[1,2], Carla Ferreira[2,3], and António Ravara[4⋆]

[1] DEETC, ISEL, Polytechnic Institute of Lisbon, Portugal
[2] CITI, FCT, New University of Lisbon, Portugal
[3] Dep. of Informatics, FCT, New University of Lisbon, Portugal
[4] SQIG, Instituto de Telecomunicações, and
Dep. of Mathematics, IST, Technical University of Lisbon, Portugal

Abstract. Most business applications rely on the notion of long running transaction as a fundamental building block. This paper presents a calculus for modelling long running transactions within the framework of the π-calculus, with support for compensation as a recovery mechanism. The underlying model of this calculus is the asynchronous polyadic π-calculus, with transaction scopes and dynamic installation of compensation processes. We add to the framework a type system which guarantees that transactions are unequivocally identified, ensuring that upon a failure the correct compensation process is invoked. Moreover, the operational semantics of the calculus ensures both installation and activation of the compensation of a transaction.

1 Introduction

Long Running Transactions (LRTs) are supported by most modern business applications. Due to the long running nature of business activities, traditional ACID transactions [1] are not suitable for them. Usually, LRTs are interactive and, consequently, cannot be check-pointed, thus cannot be based on locking, as usual for traditional ACID transactions. Instead, they use *compensations*, which are activities programmed to recover partial executions of transactions.

Several business activities specifications have been proposed, such as active service Corba [2], J2EE [3], WS-CAF [4] and WS-Coordination/Transaction [5], supporting the notion of LRTs. These transactions are a fundamental concept in building reliable distributed applications, namely when aggregating Web Services. Orchestration and Choreography languages, such as Microsoft XLANG [6] and its visual environment BizTalk, WS-BPEL [7], WS-CDL [8] and WSCI [8], support the definition of complex services in terms of interactions among simpler services. However, each specification language has a significative different interpretation of the notion of LRTs and compensable processes. Furthermore, this interpretation is often only informally defined by textual descriptions.

⋆ Partially supported by EC through project Sensoria and by FCT, through the Multiannual Funding Programme.

To proper model, and in particular, to be able to reason and ensure properties about system specifications supporting LRTs, one needs mathematical tools. Process calculi are one suitable tool, providing not only a description language, but a rigorous semantics as well, allowing the proof of relevant properties.

Several approaches use process calculi to rigorously define LRTs. Such approaches either rely on, and try to model closely, existing standards and technologies [9–11], or are language and technology independent, focusing on the main concepts associated to LRTs [12–16]. However, in most of these works the compensation mechanism is static, and moreover, their settings do not provide an important property: the guarantee of installation and of activation of a compensation. In Section 7 we present, and compare in detail with, related work.

This paper defines a formal calculus to model dynamic recovering of LRTs, which is language and technology independent. The calculus focuses on the following main concepts: compensable activities, compensation scope, dynamic installation of compensations, nesting interruptible processes and failure handlers. The calculus is built on the framework of π-calculus with a compensable transaction mechanism. Thus, it is an extension of the asynchronous polyadic π-calculus based on the fact that LRTs can be seen as interactive components, communicating asynchronously, within a distributed system.

One of the main contributions of this work is the *dynamic recovery mechanism* and, in particular, the properties this mechanism enjoys. Since in our calculus each interaction may have an associated compensation, the recovery process of a transaction is built incrementally. One of the original ideas of our calculus is that, when receiving a message (which triggers a certain process), a compensation may be automatically stored. Therefore the compensation of each transaction can be *dynamically* built, that is, the compensation processes that are executed when a transaction fails can depend on which sub-processes of the transaction were executed in a particular dynamic run of the process. Since in an asynchronous model after sending a message there are limited guarantees about the state of the receiver, namely the receiver may have been aborted after message receipt, we have chosen not to associate a compensation to the act of sending a message. Still, compensations can be installed within the execution of the transaction, without being associated to an interaction. Namely, it is possible to have "default" compensations within a process, *i.e.*, to associate compensations to transactional scopes. In case of failure, the stored compensations of a transaction are automatically activated. The origin of the failure can be internal or external, and the compensation activation is transparent in the sense that there is no need to specify it explicitly in the calculus.

Another contribution is the guarantee of *transaction soundness*, *i.e.*, once a (internal or external) failure occurs, the correct compensation is activated and will eventually occur. This is achieved via a type discipline which asserts that transactions are unequivocally identified. The type system, thus, not only guarantees type safety, but also transaction soundness.

Proofs of the results presented herein can be found in a technical report [17].

2 Dynamic Compensation Calculus

This paper presents a calculus for modelling long running transactions, with a compensation mechanism to recover from failures, named as dcπ-calculus. This mechanism allows to incrementally build the compensations of the transactions within each interaction. To achieve that, we associate to each input a process that defines the compensation to be stored upon message reception. As said before, we have chosen not to associate compensations with the sender of the message, since in an asynchronous context there are limited guarantees about the state of the receiver. Moreover, our model also supports the storing of compensations within a transaction, independently of interactions.

A transaction $t\,[E]$ encloses a process E within a transaction scope univocally identified by the transaction identifier t. In case of transaction failure, *i.e.*, abortion, all stored compensations of the transaction are activated and protected against external failures. The transaction $t\,[E]$ is the (unique) target of a failure signal $\overline{t}$, which can be internal or external to the transaction.

The calculus allows for nested transactions and failure handling in a nested way. While the abortion of a transaction is silent to its parent, it causes the abortion of all proper subtransactions and the activation of compensations installed either by the transaction or by all of its subtransactions.

The syntax of our language relies on: a countable set of channel names N, ranged over by $a, b, c, d, a_1, b_1, c_1, d_1, \ldots$; a countable set of transaction identifiers T, ranged over by $p, q, r, s, t, u, p_1, q_1, r_1, s_1, t_1, u_1, \ldots$; and natural numbers, ranged over by $i, j, k, i_1, j_1, k_1, \ldots$. The sets N and T are disjoint and identifiers $x, y, v, w, x_1, y_1, v_1, w_1, \ldots$ are used to refer to elements of both sets when there is no need to distinguish them. The tuple $\tilde{x}$ denotes a sequence $x_1 \cdots x_n$ of such identifiers, for some $n \geq 0$, and $\{\tilde{x}\}$ denotes the set of elements of that sequence.

Definition 1. *The grammar in Figure 1 defines the syntax of processes.*

Apart from the standard (asynchronous) π-calculus processes (inaction, output, parallel composition, scope restriction), we introduce: (1) the *transaction scope* $t\,[E]$ that behaves as process E until it receives on t a failure signal (which activates the stored compensations of E — see ahead); (2) the failure signal $\overline{t}$ that sends a message of failure to a transaction identified by t; (3) the *stored compensation* $\{P\}$, which has no activity until a failure occurs, becoming then the process P; (4) the *protected block* $\langle P \rangle$ that behaves as P and cannot be interrupted even if it occurs in the scope of a failing transaction. Furthermore, in process $a(\tilde{x})\%Q.P$ we associate with an input on a a *compensation process* Q. When an input action occurs, the associated compensation process is stored and becomes part of the recovery process of the transaction. As an example, consider the transactions:

$$\begin{aligned} C &\stackrel{\text{def}}{=} r\big[a(y)\%\mathbf{0}.(\overline{y}\langle\rangle \mid (z()\%(d(c).\mathbf{0}).Q + v()\%\mathbf{0}.\overline{r}))\big] \\ B &\stackrel{\text{def}}{=} q\big[\{\overline{r}\} \mid (\nu\, x)\,(\overline{x}\langle\rangle \mid (x()\%\overline{d}\langle w\rangle.\overline{a}\langle z\rangle + x()\%\mathbf{0}.\overline{a}\langle v\rangle))\big]. \end{aligned}$$

Let us assume that transaction B sends message z through channel a, after performing an internal choice through channel x. This choice also implies the

$$
\begin{array}{lll}
P, Q ::= & & \textit{Compensable Processes } (\mathcal{P}) \\
& \mathbf{0} & \text{(Inaction)} \\
\mid & \overline{t} & \text{(Failure)} \\
\mid & \overline{a}\,\langle \tilde{v} \rangle & \text{(Output)} \\
\mid & \sum_{i \in I} a_i(\tilde{x}_i)\%Q_i.P_i & \text{(Input guarded choice)} \\
\mid & !a(\tilde{x})\%Q.P & \text{(Input guarded replication)} \\
\mid & (P \mid Q) & \text{(Parallel composition)} \\
\mid & (\nu\, x)\, P & \text{(Restriction)} \\
\mid & \langle P \rangle & \text{(Protected block)} \\
\mid & t\,[E] & \text{(Transaction scope)} \\
 & & \\
E, F ::= & P & \textit{Execution Processes } (\mathcal{EP}) \\
\mid & \{P\} & \text{(Stored compensation)} \\
\mid & (E \mid F) & \text{(Parallel composition)} \\
\mid & (\nu\, x)\, E & \text{(Restriction)}
\end{array}
$$

Fig. 1: The syntax of processes.

installation of compensation $\overline{d}\langle w \rangle$. After receiving the message, transaction C must check it and decide how to proceed. Thus, since B sent message z, C installs compensation $d(c).\mathbf{0}$ and proceeds as $Q\{^z/_y\}$. Therefore, both transactions will evolve respectively to:

$$C' \stackrel{\text{def}}{=} r\big[\{d(c).\mathbf{0}\} \mid Q\{^z/_y\}\big] \qquad B' \stackrel{\text{def}}{=} q\big[\{\overline{r}\} \mid \{\overline{d}\langle w \rangle\}\big].$$

Moreover, our model gives the possibility to specify stored processes $\{P\}$, such as $\{\overline{r}\}$ in transaction C', which are stored as compensations within a transaction, independently of interactions. In fact, this feature allows the definition of "default" compensations of a transaction, *i.e.*, compensations associated to a transaction scope, which will be always activated upon a failure. To model the storing of compensations, we distinguish between compensable processes $(P, Q, ...)$ and execution processes $(E, F, ...)$, where the last ones allow the specification of stored compensations $(\{P\})$. We refer to both as processes whenever there is no need to distinguish them.

The level of granularity of the dynamic installation of compensations in our model is very flexible. The system designer can choose from defining a compensation process for each input action, to defining a unique compensation for each sequential process of inputs. The latter case can be achieved by associating the inaction process as a compensation to all input prefixes of the sequence except for one, which will be the overall compensation of the sequential process.

Notice that there is no causality information about installed compensations, *i.e.*, all installed compensations are executed in parallel. This design choice was influenced by the fact that in practice compensations typically use forward recovery [18]. Also, on the occurrence of a transaction failure, its stored compensations will be activated and placed within a protected block. This block ensures that compensations cannot be interrupted while subprocess of a failing transaction. However, a transaction inside a protected block can fail, when it explicitly re-

ceives a failure message. Similarly to stored compensations, protected blocks can also be defined independently of the occurrence of failures. This feature would allow to have protected blocks by "default".

Consider the previous transactions B', C' and the failure message $\overline{q}$. Suppose that $P = \overline{q} \mid B' \mid C'$. Then, P will evolve to a process $P' = \langle \overline{r} \rangle \mid \langle\, \overline{d}\, \langle w \rangle \rangle \mid C'$, *i.e.*, the stored compensations of B' have been activated and placed within protected blocks. Notice that the evolution of process P' will also imply the failure of transaction C' and the activation of its respective compensation process.

In the following, we omit the sum symbol if the indexing set is singular; term $\Pi_{i \in I} E_i$ abbreviates the parallel composition of the processes E_i with $i \in I$, where I is a finite set of indexes; term $(\nu\, \tilde{x})\, E$ abbreviates $(\nu\, x_1) \cdots (\nu\, x_n)\, E$, for some $n \geq 1$; and as usual, the operator ν binds tighter than the operator $\mid$. We also abbreviate $a(x)\%0$, $a()$ and $\overline{a}\, \langle\rangle$ with $a(x)$, a and $\overline{a}$, respectively.

3 Example

In this section we describe an ordering system. We can think of it as a web shop that accepts orders from clients. Whenever a client submits an order, the system must take care of payment and order packing. We model the system as depicted in Figure 2, which for simplicity only considers one client and one order.

$$\textsf{OrderTransaction} \stackrel{\text{def}}{=} (\nu\, info)\, (\textsf{Client} \mid \textsf{Bank}) \mid \textsf{Shop} \mid \textsf{Warehouse}$$

$$\begin{aligned}\textsf{Client} \stackrel{\text{def}}{=}\ & (\nu\, u)\, u\big[(\nu\, ctl, msg)\, (\overline{ord}\langle ctl, msg\rangle \mid \\ & ctl(s).(\overline{ctl}\langle u\rangle \mid (\nu\, x)\, (\overline{x} \mid (x.\overline{s} + x.info(q).\overline{q} + x.(msg(done) \mid info(q)))))\big]\end{aligned}$$

$$\begin{aligned}\textsf{Shop} \stackrel{\text{def}}{=}\ & !\ ord(ctl, msg).\\ & (\nu\, s)\, s\big[\overline{ctl}\langle s\rangle \mid ctl(u)\%\overline{u}.(\nu\, bOk, pOk)\, (\textsf{Charge} \mid \textsf{Pack} \mid bOk.pOk.\overline{msg}\langle done\rangle)\big]\end{aligned}$$

$$\begin{aligned}\textsf{Charge} \stackrel{\text{def}}{=}\ & (\nu\, r)\, r\big[\{\overline{s}\}\big] \mid (\nu\, ctl, msg)\, (\overline{op}\langle ctl, msg\rangle \mid \\ & ctl(q)\%\overline{q}.\overline{ctl}\langle r\rangle \mid msg(x).(\overline{x} \mid (valid\%msg(refunded).\overline{bOk} + invalid.\overline{r})))\big]\end{aligned}$$

$$\begin{aligned}\textsf{Pack} \stackrel{\text{def}}{=}\ & (\nu\, p)\, p\big[\{\overline{s}\}\big] \mid (\nu\, ctl, msg)\, (\overline{pkg}\langle ctl, msg\rangle \mid \\ & ctl(t)\%\overline{t}.\overline{ctl}\langle p\rangle \mid msg(x).(\overline{x} \mid (packed\%msg(unpacked).\overline{pOk} + unavail.\overline{p})))\big]\end{aligned}$$

$$\begin{aligned}\textsf{Bank} \stackrel{\text{def}}{=}\ & !\ op(ctl, msg).(\nu\, q)\, q\big[\overline{ctl}\langle q\rangle \mid ctl(r)\%\overline{r}.\\ & (\overline{info}\langle q\rangle \mid (\nu\, x)\, (\overline{x} \mid (x\%\overline{msg}\langle refunded\rangle.\overline{msg}\langle valid\rangle + x.\overline{msg}\langle invalid\rangle)))\big]\end{aligned}$$

$$\begin{aligned}\textsf{Warehouse} \stackrel{\text{def}}{=}\ & !\ pkg(ctl, msg).(\nu\, t)\, t\big[\overline{ctl}\langle t\rangle \mid \\ & ctl(p)\%\overline{p}.(\nu y)(\overline{y} \mid (y\%\overline{msg}\langle unpacked\rangle.\overline{msg}\langle packed\rangle + y.\overline{msg}\langle unavail\rangle))\big]\end{aligned}$$

Fig. 2: Ordering system example.

In this example all transaction identifiers are restricted and access to them by other transactions is given through scope extrusion. Thus, if we had different

clients, we would be able to ensure transaction context separation and proper cancellation. Note also that transactions are started with a three-way handshake. Client starts to send two private names, *ctl* for control and *msg* for other messages, and waits that the receiver sends a transaction identifier through *ctl*. The receiver receives the private names, starts a new transaction, sends its identifier through *ctl* and waits that the client sends also its transaction identifier.

In this scenario, the client submits an order to the shop and waits for order confirmation. The shop receives the message and starts two transactions, charging the client and packing the order. The payment is done by the bank, therefore the shop sends a message to the bank and it starts a new transaction. Similarly, the shop sends a message to the warehouse and a new packing transaction starts. Notice that the client may cancel either the shop transaction or the bank transaction. In both cases, the system stops and compensation transactions are executed. If charging and packing are successfully accomplished, the shop sends a message to the client and terminates the transaction.

The compensations are incrementally built. For example, when the bank starts to interact with the shop, it installs the compensation $\overline{r}$ and, when it validates the payment, it installs the compensation $\overline{msg}\langle refunded\rangle$. Whenever a failure occurs, the first compensation ensures that the charge subtransaction is cancelled and the second one informs that the client is being refunded. It is interesting to note that if the bank fails before charging validation, only the first compensation is executed. This is an important feature of dynamic installation mechanism.

In this example, there are also "default" compensations of transactions, namely in the subtransactions charge and pack. For instance, when the shop receives the client order, it starts the subtransaction charge. This subtransaction, by default, installs the compensation $\overline{s}$. This compensation ensures that the shop transaction is cancelled if charge fails.

As noted before, the client can cancel both shop transaction and bank transaction. If the client cancels the bank transaction, we must ensure that shop knows about it. Therefore, the compensation $\overline{r}$ installed by the bank cancels the charge subtransaction. Thus, it is ensured that the client does not get the goods for free. A possible execution of a successful transaction and an execution where the client cancels the transaction can be seen in Appendix A.

In this example we have also nested transactions. The shop transaction includes two inner transactions, the charging transaction and the packing transaction. If the shop transaction fails, then both inner transactions will also fail and, most important, we do not need to explicitly model it. Nesting also allows us to separate different inner transactions and restrict the action of external agents. In the example, the bank can only interrupt the inner transaction identified by r and, although we choose not to, the shop could try another payment method without failing.

Finally, we highlight the importance of keeping installed compensations after the end of a transaction. Suppose that the bank transaction ends successfully, but that the warehouse fails the packing. We must be able to compensate the

Scope extension laws

$$(\nu\, x)\, \mathbf{0} \equiv \mathbf{0} \qquad E \mid (\nu\, z)\, F \equiv (\nu\, z)\, (E \mid F) \quad \text{if} \quad z \notin \mathsf{fn}(E)$$
$$(\nu\, z)\, (\nu\, w)\, E \equiv (\nu\, w)\, (\nu\, z)\, E \qquad t\, [(\nu\, y)\, E] \equiv (\nu\, y)\, t\, [E] \quad \text{if} \quad t \neq y$$
$$\langle (\nu\, x)\, P \rangle \equiv (\nu\, x)\, \langle P \rangle$$

Protected block, stored and termination laws

$$\langle\langle P \rangle\rangle \equiv \langle P \rangle \qquad \langle P \mid Q \rangle \equiv \langle P \rangle \mid \langle Q \rangle \qquad \{\langle P \rangle\} \equiv \{P\} \qquad \langle \mathbf{0} \rangle \equiv \mathbf{0} \qquad \{\mathbf{0}\} \equiv \mathbf{0}$$

Fig. 3: Structural congruence relation.

charging transaction, *i.e.*, we must refund the client. In the next Section, we will discuss how our calculus supports this behaviour.

4 Operational Semantics

We define an operational semantics by means of a reduction relation on execution processes, making use of a structural congruence relation, following the usual approach of Milner *et al.* [19].

As for bindings, in processes $a(\tilde{x})\%Q.P$, $!a(\tilde{x})\%Q.P$, and $(\nu\, \tilde{x})\, E$, the occurrences of names and transaction identifiers of $\tilde{x}$ are bound in the subprocesses P, Q and E. Furthermore, we use the standard notions of free names of processes and of α-equivalence. We write $\mathsf{bn}(E)$ (respectively $\mathsf{fn}(E)$) for the set of names that are bound (respectively free) in a process E.

Definition 2. *Structural congruence $\equiv$ is the smallest congruence relation on execution processes satisfying the α-conversion law, the abelian monoid laws for parallel and inaction, and the laws in Figure 3.*

The scope laws are standard. The law $\langle\langle P \rangle\rangle = \langle P \rangle$ reflects the intended semantics of a protected block being already protected. The law $\langle P \mid Q \rangle = \langle P \rangle \mid \langle Q \rangle$ flattens nested protected blocks. The law $\{\langle P \rangle\} = \{P\}$ reflects a feature of the calculus, namely a stored block cannot be executed until its extraction into a protected block, which occurs within a failure. The termination laws are straightforward.

The dynamic behaviour of processes is defined by a reduction relation in which we must take into account some aspects of the transaction scope behaviour. For instance, when we send a message through the transaction identifier, the transaction should fail and all its stored compensations should be activated. Therefore, we have to extract the stored compensations and place them in a protected block. To extract stored compensations of a transaction scope, we use the function extr which is defined as follows.

Definition 3. *Function $\mathsf{extr} : \mathcal{EP} \longmapsto \mathcal{P}$ for extracting stored compensations, is inductively defined in Figure 4.*

With the definition of extr, we can always write a process $\mathsf{extr}(E)$ with the form expressed in the following lemma.

$$
\begin{array}{rlrl}
\mathsf{extr}(\mathbf{0}) &= \mathbf{0} & \mathsf{extr}(\{P\}) &= \langle P \rangle \\
\mathsf{extr}(\overline{t}) &= \mathbf{0} & \mathsf{extr}(\langle P \rangle) &= \langle P \rangle \\
\mathsf{extr}(\overline{a}\,\langle \tilde{v} \rangle) &= \mathbf{0} & \mathsf{extr}(t\,[E]) &= \mathsf{extr}(E) \\
\mathsf{extr}(\textstyle\sum_{i \in I} a_i(\tilde{x}_i)\%Q_i.P_i) &= \mathbf{0} & \mathsf{extr}(E \mid F) &= \mathsf{extr}(E) \mid \mathsf{extr}(F) \\
\mathsf{extr}(!a(\tilde{x})\%Q.P) &= \mathbf{0} & \mathsf{extr}((\nu\, x)\, E) &= (\nu\, x)\ \mathsf{extr}(E)
\end{array}
$$

Fig. 4: Extraction function.

$$
\begin{array}{rl}
C[\![\bullet]\!] & ::= \bullet \mid (\nu\, x)\, C[\![\bullet]\!] \mid C[\![\bullet]\!] \mid E \mid \langle C[\![\bullet]\!] \rangle \mid t\,[C[\![\bullet]\!]] \\
D[\![\bullet, \bullet]\!] & ::= C[\![\bullet]\!] \mid C[\![\bullet]\!]
\end{array}
$$

Fig. 5: Execution contexts and double execution contexts.

Lemma 1. *Let E be an execution process. Then* $\mathsf{extr}(E)$ *is structurally congruent to a process of the form* $(\nu\, \tilde{y})\, \Pi_{i \in I} \langle P_i \rangle$.

Interactions can happen in different execution contexts. Since all our interactions are binary, we also introduce double contexts, *i.e.*, two execution contexts that can interact. The grammar in Figure 5 generates execution and double execution contexts.

Definition 4. *The grammar in Figure 5 inductively defines* execution contexts*, denoted by $C[\![\bullet]\!]$, and* double execution contexts*, denoted by $D[\![\bullet, \bullet]\!]$.*

Applying a double execution context $D[\![\bullet, \bullet]\!]$ with two holes $\bullet$ to two processes E and F produces the process obtained by replacing the left hole with E and the right hole with F, *i.e.*, $D[\![E, F]\!]$.

Definition 5. *The reduction relation $\rightarrow$ is the least relation satisfying the rules of Figure 6.*

Some rules of the reduction relation deserve an explanation. The rule R-CONG allows reduction to happen inside arbitrary execution contexts. Rules R-COM and R-REP allow communication in a double execution context, *i.e.* within two execution contexts, while storing the associated compensations in the respective execution contexts. As usual, arities of names must be respected within communications. We ensure this with a type system, presented in the next Section. Transaction failures are modelled by rules R-RECOVER-IN, used when the failure message is internal to the transaction, and R-RECOVER-OUT, otherwise. Moreover, in rule R-RECOVER-OUT the transaction and failure message can occur in different execution contexts. Note also that in rules R-RECOVER-IN and R-RECOVER-OUT, when we have a protected block within the transaction, it will not be interrupted by the definition of extr.

Our semantics allows to compensate a completed transaction since our compensations are not discarded. This design choice was influenced by an expected feature of compensation handlers: the possibility to compensate transactions partially executed or completed transactions, where by completed transaction

$$\text{(R-COM)}\quad \frac{D \text{ does not bind } a \qquad a = a_j \text{ for some } j \in I}{D[\![\overline{a}\,\langle\tilde{v}\rangle \,,\, \sum_{i\in I} a_i(\tilde{x}_i)\%Q_i.P_i]\!] \;\rightarrow\; D[\![\mathbf{0}, \{Q_j\{\tilde{v}/\tilde{x}_j\}\} \mid P_j\{\tilde{v}/\tilde{x}_j\}]\!]} \qquad \text{(R-CONG)}\quad \frac{E \rightarrow E'}{C[\![E]\!] \rightarrow C[\![E']\!]}$$

$$\text{(R-REP)}\quad \frac{D \text{ does not bind } a}{D[\![\overline{a}\langle\tilde{v}\rangle, !a(\tilde{x})\%Q.P]\!] \rightarrow D[\![\mathbf{0}, \{Q\{\tilde{v}/\tilde{x}\}\} \mid P\{\tilde{v}/\tilde{x}\} \mid !a(\tilde{x})\%Q.P]\!]} \qquad \text{(R-RECOVER-IN)}\quad \frac{C \text{ does not bind } t}{t\,[C[\![\,\overline{t}\,]\!]] \rightarrow \mathsf{extr}(C[\![\mathbf{0}]\!])}$$

$$\text{(R-STRUCT)}\quad \frac{E' \equiv E \quad E \rightarrow F \quad F \equiv F'}{E' \rightarrow F'} \qquad \text{(R-RECOVER-OUT)}\quad \frac{D \text{ does not bind } t}{D[\![\,\overline{t}, t\,[E]]\!] \rightarrow D[\![\mathbf{0}, \mathsf{extr}(E)]\!]}$$

Fig. 6: Reduction rules.

we mean a transaction with no more active inputs, excluding transaction identifiers. This feature would not have problems of scalability in a real system, since it could be associated to our calculus a distributed garbage collection [20] to remove compensations of transactions no longer reachable.

In Appendix A we provide partial reductions for two executions of the example described in Section 3.

5 Uniqueness of Transaction Identifiers

To guarantee transaction soundness, *i.e.*, upon a transaction failure the correct compensation is activated, we define a (simple) type system, adding some conditions and rules to the basic type system of the π-calculus [21]. The defined type system will provide transaction soundness by (statically) verifying that transactions are unequivocally identified, *i.e.*, that transactions identifiers are unique in a process.

Firstly, it is necessary to identify the *set of transaction identifiers* that occur in a process E, denoted by $\mathsf{ti}(E)$. Namely, in each $t[E]$, the displayed occurrence of t is a *transaction identifier*, which should be unique, *i.e.*, a given process cannot have two transaction scopes with the same identifier. Consequently, we must ensure for example that in the case of the input guarded replication $!a(\tilde{x})\%Q.P$, the transaction identifiers cannot occur free. This leads to the definition of the *set of free transaction identifiers* of a process E, denoted by $\mathsf{fti}(E)$ and defined as $\mathsf{fti}(E) = \mathsf{ti}(E) \setminus \mathsf{bn}(E)$.

Types distinguish between transaction identifiers and channels.

Definition 6. *The grammar in Figure 7 defines the syntax of types.*

Let Γ be a partial function from channel names to channel types and from transaction identifiers to transaction types. We write $\tilde{T}$ for a tuple $T_1, \ldots, T_n$ of types and $\tilde{v} : \tilde{T}$ for a sequence $v_1 : T_1, \ldots, v_n : T_n$ of labelled types. The comma in $\Gamma, x : T$ denotes disjoint union.

$$
\begin{array}{lll}
T ::= & & \textit{Types} \\
& \mathsf{tr} & \textit{transaction types} \\
\mid & \mathsf{ch}(T_1, \ldots, T_n),\ n \geq 0 & \textit{channel types} \\
\\
\Gamma ::= \emptyset \mid \Gamma, x : T & & \textit{Type environments}
\end{array}
$$

Fig. 7: The syntax of types.

$$
\text{(T-NIL)}\ \ \Gamma \vdash 0
\qquad
\text{(T-PAR)}\ \ \frac{\Gamma \vdash E \quad \Gamma \vdash F \quad \mathsf{fti}(E) \cap \mathsf{fti}(F) = \emptyset}{\Gamma \vdash E \mid F}
\qquad
\text{(T-RES)}\ \ \frac{\Gamma, x\colon T \vdash E}{\Gamma \vdash (\nu\, x)\, E}
$$

$$
\text{(T-INP)}\ \ \frac{\forall_{i \in I} (\Gamma(a_i) = ch_i(\tilde{T}_i) \quad \Gamma, \tilde{x}_i : \tilde{T}_i \vdash P_i \mid Q_i \quad \{\tilde{x}_i\} \cap (\mathsf{fti}(P_i) \cup \mathsf{fti}(Q_i)) = \emptyset)}{\Gamma \vdash \sum_{i \in I} a_i(\tilde{x}_i)\%Q_i.P_i}
$$

$$
\text{(T-REP)}\ \ \frac{\Gamma(a) = \mathsf{ch}(\tilde{T}) \quad \Gamma, \tilde{x} : \tilde{T} \vdash P \mid Q \quad \mathsf{fti}(P) = \emptyset = \mathsf{fti}(Q)}{\Gamma \vdash\, !a(\tilde{x})\%Q.P}
\qquad
\text{(T-OUT)}\ \ \Gamma, a\colon \mathsf{ch}(\tilde{T}), \tilde{v}\colon \tilde{T} \vdash \overline{a}\,\langle \tilde{v} \rangle
$$

$$
\text{(T-STORED)}\ \ \frac{\Gamma \vdash P}{\Gamma \vdash \{P\}}
\qquad
\text{(T-BLOCK)}\ \ \frac{\Gamma \vdash P}{\Gamma \vdash \langle P \rangle}
\qquad
\text{(T-TRANS-ID)}\ \ \Gamma, t\colon \mathsf{tr} \vdash \overline{t}
\qquad
\text{(T-SCOPE)}\ \ \frac{\Gamma \vdash E \quad \Gamma(t) = \mathsf{tr} \quad t \notin \mathsf{fti}(E)}{\Gamma \vdash\ t\,[E]}
$$

Fig. 8: Type system.

Definition 7. *The rules in Figure 8 inductively define the type system.*

Due to the requirement for uniqueness of transaction identifiers, we must also ensure that the identifier of a transaction cannot be defined by instantiation. The identifier must be defined within the transaction definition, where it is decided if it is *public*, *i.e.*, all processes know this identifier and can cancel the transaction, or *protected*, *i.e.*, the transaction can only be cancelled by itself or by other processes with permission, that can be given for instance through scope extrusion. This feature is assured in the type rules T-INP and T-REP.

This type system is consistent with the operational semantics and ensures that transactions are unequivocally identified.

Theorem 1 (Subject Reduction). *Let $\Gamma \vdash E$ and $E \rightarrow E'$. Then $\Gamma \vdash E'$.*

To state the uniqueness property of the transaction identifiers, we need first to define the predicate $\mathsf{unq}(E)$, which verifies if the transaction identifiers of a process are unique.

Definition 8. *The predicate $\mathsf{unq}(E)$ on processes is inductively defined in Figure 9.*

$$
\begin{array}{rl}
\mathsf{unq}(0) & \\
\mathsf{unq}(\overline{t}\,\langle\rangle) & \\
\mathsf{unq}(\overline{a}\,\langle\tilde{v}\rangle) & \\
\mathsf{unq}(\sum_{i\in I} a_i(\tilde{x}_i)\%Q_i.P_i) & \text{if } \mathsf{unq}(P_i) \text{ and } \mathsf{unq}(Q_i) \text{ and } \mathsf{fti}(P_i)\cap\mathsf{fti}(Q_i)=\emptyset, \text{ for all } i\in I \\
\mathsf{unq}(!a(\tilde{v})\%Q.P) & \text{if } \mathsf{fti}(P)=\emptyset \text{ and } \mathsf{fti}(Q)=\emptyset \\
\mathsf{unq}(\langle P\rangle) & \text{if } \mathsf{unq}(P) \\
\mathsf{unq}(\{P\}) & \text{if } \mathsf{unq}(P) \\
\mathsf{unq}(E\mid F) & \text{if } \mathsf{unq}(E) \text{ and } \mathsf{unq}(F) \text{ and } \mathsf{fti}(E)\cap\mathsf{fti}(F)=\emptyset \\
\mathsf{unq}((\nu\, x)\, E) & \text{if } \mathsf{unq}(E) \\
\mathsf{unq}(t[E]) & \text{if } \mathsf{unq}(E) \text{ and } t\notin\mathsf{fti}(E)
\end{array}
$$

Fig. 9: Uniqueness predicate.

We are now in a position to show that transaction identifiers are unique in well-typed processes.

Theorem 2 (Soundness). *Let $\Gamma\vdash E$. Then $\mathsf{unq}(E)$ holds.*

Notice that our type system also assures type safety, that is, each name respects arity: if the name w has arity n then each occurrence of $\overline{a}\,\langle x_1,\ldots,x_k\rangle$ and $a(x_1,\ldots,x_k)$ is well-formed only if $k=n$. Processes not satisfying this are *errors*. Let $\rightarrow^*$ denote the reflexive and transitive closure of $\rightarrow$.

Definition 9 (Error processes).
$$
\begin{array}{l}
\mathit{Error}=\{E \mid (E\rightarrow^* D[\![\overline{a_j}\langle\tilde{v}\rangle, \sum_{i\in I} a_i(\tilde{x}_i)\%Q_i.P_i]\!] \text{ and } |\tilde{v}|\neq|\tilde{x}_j|) \text{ or} \\
\qquad\qquad (E\rightarrow^* D[\![\overline{a}\langle\tilde{v}\rangle, !a(\tilde{x})\%Q.P]\!] \text{ and } |\tilde{v}|\neq|\tilde{x}|)\}
\end{array}
$$

Theorem 3 (Type Safety). *Let $\Gamma\vdash E$. Then $E\notin \mathit{Error}$.*

6 Properties of the Recovery Mechanism

In this section we formally state the main distinctive features of the proposed calculus: the assurance of both installation and activation of process compensations. In the proposed calculus the compensations of a transaction are defined dynamically, *i.e.*, since compensations are associated to input prefixes, they are incrementally installed within the execution of the transaction. Therefore, the compensation processes that are executed after a transaction failure can depend on which subprocesses of the transaction were executed until that moment. Even more, compensations can also be associated to transaction scopes, allowing the existence of "default" compensations.

As mentioned before, a transaction may fail in two different ways: the failure can be raised by internal or external messages. In both cases, stored compensations are activated. As an example, consider the following transaction:

$$\mathbf{E}\stackrel{\text{def}}{=}\ t\,[\{Q_1\}\mid a(x)\%Q_2.b(y)\%Q_3.R\mid \overline{c}\,\langle z\rangle]$$

If transaction t fails before receiving a message through name a, the process Q_1 will be its compensation process. Alternatively, if a transaction failure occurs immediately after receiving a message through the name a, the process $Q_1 \mid Q_2$ will be its compensation process.

The following propositions assert that installed compensations of a transaction are automatically activated whenever a failure occurs.

Proposition 1. *Let $E = t\left[C[\![\,\bar{t}\,]\!]\right]$ be a typable process, and C a context that does not bind t. If $E \to E'$ by applying the* R-RECOVER-IN *rule with respect to the subterm $\bar{t}$, then $E' = \mathsf{extr}(C[\![\mathbf{0}]\!])$.*

Proposition 2. *Let $E = D[\![t\,[F]\,, \bar{t}\,]\!]$ be a typable process, and D a context that does not bind t. If $E \to E'$ by applying the* R-RECOVER-OUT *rule to the subterms $t\,[F]$ and $\bar{t}$, then $E' = D[\![\mathsf{extr}(F), \mathbf{0}]\!]$.*

Consider the previous transaction E and the following:

$$\mathsf{F} \stackrel{\text{def}}{=} \; p\left[\overline{a}\,\langle v\rangle \mid \overline{b}\,\langle z\rangle \mid c(w)\%S_1.P_1\right]$$

In order to exemplify the dynamic definition of the compensation processes of both transactions, we will consider a particular execution of $E \mid F$.

Whenever an interaction occurs, we must ensure that the associated compensation process is installed. Since an input can occur in two situations, input guarded choice and input guarded replication, in the following theorems we state that in both cases the compensations are installed. Initially, the compensation processes of transaction E and F are Q_1 and $\mathbf{0}$, respectively. Suppose that a message is sent through a and E receives it. Then, in transaction E a new compensation process is installed, and its compensation becomes $Q_1 \mid Q_2$. If transaction E receives a message of transaction F through b, the compensation process of E will be appended with process Q_3, *i.e.*, it will become $Q_1 \mid Q_2 \mid Q_3$.

Proposition 3. *Let $E = D[\![\overline{a}_j\,\langle\tilde{y}\rangle\,, \sum_{i\in I} a_i(\tilde{x_i})\%Q_i.R_i]\!]$ be a typable process. If $E \to E'$ by applying the* R-COM *rule to the subterms $\sum_{i\in I} a_i(\tilde{x}_i)\%Q_i.R_i$ and $\overline{a}_j\,\langle\tilde{y}\rangle$, then $E' = D[\![\mathbf{0}, R_j\{\tilde{y}/\tilde{x}_j\} \mid \{Q_j\{\tilde{y}/\tilde{x}_j\}\}]\!]$.*

Proposition 4. *Let $E = D[\![\overline{a}\,\langle\tilde{y}\rangle\,, !a(\tilde{x})\%Q.R]\!]$ be a typable process. If $E \to E'$ by applying the* R-REP *rule to the subterms $!a(\tilde{x})\%Q.R$ and $\overline{a}\,\langle\tilde{y}\rangle$, then $E' = D[\![\mathbf{0}, !a(\tilde{x})\%Q.R \mid R\{\tilde{y}/\tilde{x}\} \mid \{Q\{\tilde{y}/\tilde{x}\}\}]\!]$.*

7 Related Work

There are other approaches that use process calculi toward the formalization of LRTs and their compensation mechanisms. In this section we briefly compare our calculus $\mathtt{dc}\pi$ with them and in Table 1 we present a succinct comparison.

Bocchi *et al.* introduced πt-calculus [9], which is inspired by BizTalk, and consists of an extension of asynchronous polyadic π-calculus [22] with the notion of transactions. However, the compensation of each transaction is statically

defined, *i.e.*, the compensations are not incrementally built. In our calculus, we have included a dynamic recovery mechanism.

The cJoin calculus [14] is an extension of Join calculus [23] with primitives for representing transactions. As in πt-calculus, the compensation mechanism of this calculus is statically defined. In contrast to our calculus, completed transactions cannot be compensated, *i.e.*, after a transaction completes, compensations are discarded. Therefore, in cJoin, only running transactions can be compensated whenever interrupted.

Butler and Ferreira [10] propose the StAC language, which is inspired by BPBeans. The language includes the notion of compensation pair, similar to the sagas concept defined by Gargia-Molina and Salem [24]. In StAC, a LRT is seen as a composition of one or more sub-transactions, where each of them has an associated compensation. In contrast to our calculus, StAC is flow composition based and includes explicit operators for running or discarding installed compensations. As well, compensating CSP [12], denoted by cCSP, and Sagas calculi [13] are also composition flow based, namely they adopt a centralized coordination mechanism. They have similar operators but different compensation policies. However these three approaches are conceptually different from ours, as they are flow based and do not provide mobility.

Laneve and Zavattaro define a calculus named webπ [15] which is an extension of asynchronous polyadic π-calculus with a timed transaction construct. An untimed version of webπ, known as webπ_∞ was proposed by Mazzara and Lanese [16]. Although our calculus shares some syntax similarities with both calculi, we have followed different principles. Namely, in both calculi the nested transactions are flattened. Thus these calculi do not provide nested failure because the failure of a transaction does not cause the abortion of proper subtransactions. This is a substantial difference with respect to our calculus, since it implies that the internal transaction cancelling must be explicitly programmed within the specification. Another difference is that in these calculi completed transactions cannot be compensated. As in πt-calculus the compensation mechanism is statically defined. Furthermore, they assume that transactions are unequivocally identified, whereas in our approach a type system ensures this feature in order to guarantee transaction soundness.

Guidi *et al.* [11] propose an extension of SOCK [25], which is inspired by WSDL and BPEL. This calculus includes explicit primitives for dynamic handler installation, such as fault and compensation handlers and automatic failure notification. They assert correctness properties for their calculus, namely the expected behaviour of a scope, the correct termination upon a failure, the correct behaviour of communications and guarantee of fault activation. Our approach is different in the sense that both installation and activation of compensations are transparent to the user, *i.e.*, they occur implicitly within interactions. Thus, making the syntax clear and simpler. Similar to $\texttt{web}\pi_\infty$, they only assume that transactions are unequivocally identified, lacking support to formally ensuring this feature. Another difference is that in our calculus we use a type discipline

Table 1: Comparison of compensation mechanisms of interaction based calculi.

Calculi	Asynchronous calculus	Compensation construction	Compensation installation	Completed transactions	Nested failure
πt-calc.	yes	static	implicitly	not compensable[a]	yes
cJoin	yes	static	implicitly	not compensable	yes
Webπ	yes	static	implicitly	not compensable	no
Web$\pi\infty$	yes	static	implicitly	not compensable	no
SOCK	no	dynamic	explicitly[b]	compensable	yes
dcπ-calc.	yes	dynamic[c]	implicitly	compensable	yes

[a] Except if is an inner transaction of a failing transaction.
[b] With explicit primitives and recurring to prioritization.
[c] Also supports static compensation constructions.

to ensure soundness and the installation and activation of transaction compensations.

8 Conclusion

In this paper we have proposed a calculus for reasoning about long running transactions, language and technology independent. We have built our calculus on the framework of asynchronous polyadic π-calculus. One of our main contributions is the recovery mechanism, which is based on compensations and supports the specification of both dynamic and static compensations. This is a first effort to further study the expressiveness of different policies, namelly dynamic versus static. Another contribution is a type discipline that ensures both calculus soundness and safety. Finally, we have defined a compensation semantics that ensures both installation and activation of transaction compensations.

The expressiveness of our calculus was demonstrated with a case study. It was shown that is possible to model deeply connected transactions in a comprehensive way. Notice that in the case study the overall effect of the execution of compensations is equivalent to the non execution of the transactions (in the sense that every received request is later cancelled). However, in a more complex scenario, it is hard to assert such behaviour. Thus, one of our future research directions is to study compensation soundness given a notion of transaction equivalence.

References

1. Gray, J.: The transaction concept: Virtues and limitations (invited paper). In: VLDB, IEEE Computer Society (1981) 144–154
2. OMG: Additional Structuring Mechanisms for the OTS Specification 1.0. (September 2002)
3. Sun Microsystems: J2EE Activity Service for Extended Transactions. (March 2004)
4. OASIS: Web Services Composite Application Framework (WS-CAF). (2005)

5. Microsoft, IBM, BEA: WS-Coordination/WS-Transaction Specification. (2005)
6. Thatte, S.: XLANG: Web services for business process design. Technical report, Microsoft Corporation (2001)
7. OASIS: Web Services Business Process Execution Language Version 2.0. (April 2007)
8. Kavantzas, N., Olsson, G., Michkinsky, J., Chapman, M.: Web services choreography description language. Technical report, Oracle Corporation (2003)
9. Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In Najm, E., Nestmann, U., Stevens, P., eds.: FMOODS. Volume 2884 of LNCS., Springer (2003) 124–138
10. Butler, M.J., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In Nicola, R.D., Ferrari, G.L., Meredith, G., eds.: COORDINATION. Volume 2949 of LNCS., Springer (2004) 87–104
11. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: 8th International Conference on Application of Concurrency to System Design, IEEE Computer Society (2008) 190–199
12. Butler, M.J., Hoare, C.A.R., Ferreira, C.: A trace semantics for long-running transactions. In Abdallah, A.E., Jones, C.B., Sanders, J.W., eds.: 25 Years Communicating Sequential Processes. Volume 3525 of LNCS., Springer (2004) 133–150
13. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In Palsberg, J., Abadi, M., eds.: POPL, ACM (2005) 209–220
14. Bruni, R., Melgratti, H.C., Montanari, U.: Nested commits for mobile calculi: Extending join. In Lévy, J.J., Mayr, E.W., Mitchell, J.C., eds.: IFIP TCS, Kluwer (2004) 563–576
15. Laneve, C., Zavattaro, G.: Foundations of web transactions. In Sassone, V., ed.: FoSSaCS. Volume 3441 of LNCS., Springer (2005) 282–298
16. Mazzara, M., Lanese, I.: Towards a unifying theory for web services composition. In Bravetti, M., Núñez, M., Zavattaro, G., eds.: WS-FM. Volume 4184 of LNCS., Springer (2006) 257–272
17. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. Technical report, CITI http://pwp.net.ipl.pt/cc.isel/cvaz/dcpi.pdf.
18. JBoss: Web Service Transactions Programmers Guide. (April 2007)
19. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I and II. Inf. Comput. **100**(1) (1992) 1–77
20. Veiga, L., Ferreira, P.: Asynchronous complete distributed garbage collection. In: IPDPS, IEEE Computer Society (2005)
21. Vasconcelos, V.T., Honda, K.: Principal typing-schemes in a polyadic π-calculus. In: 4th CONCUR. Volume 715 of LNCS., Springer (August 1993) 524–538
22. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press (2001)
23. Fournet, C., Gonthier, G.: The reflexive cham and the join-calculus. In: POPL. (1996) 372–385
24. Garcia-Molina, H., Salem, K.: Sagas. In Dayal, U., Traiger, I.L., eds.: SIGMOD Conference, ACM Press (1987) 249–259
25. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: : A calculus for service oriented computing. In Dan, A., Lamersdorf, W., eds.: ICSOC. Volume 4294 of LNCS., Springer (2006) 327–338

A Example Executions

In this section we provide two possible executions for the **OrderTransaction** example presented in Section 3. We also sketch the reduction path for both executions.

A successfully execution is depicted in Figure 10, where the implicit creation of **Charge** and **Pack** subtransactions is denoted with the keyword *create*. Note also that in Figure 10, the inner executions of each transaction are omitted.

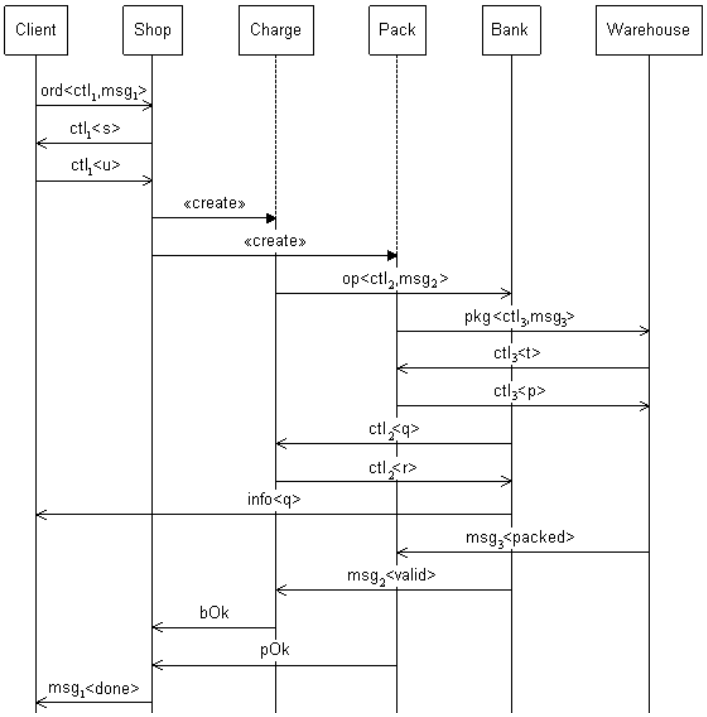

Fig. 10: An execution of the **OrderTransaction** successfully completed.

Transactions **Client** and **Shop** start by performing a handshake, where they exchange transaction identifiers. Then, **Charge** and **Pack** are started and they perform also handshakes with **Bank** and **Warehouse** transactions, respectively. These two handshakes can be interleaved as depicted in Figure 10.

Let $\mathbf{OrderTransaction}_1$ denote the resulting process after the mentioned handshakes. We are able to

$$\mathbf{OrderTransaction} \longrightarrow^* \mathbf{OrderTransaction}_1,$$

by applying successively the operational semantics rules presented in Section 4, with

$$\mathsf{OrderTransaction}_1 \stackrel{\text{def}}{=} (\nu\, t, p, ctl_3, msg_3)\,((\nu\, info, r, q, ctl_2, msg_2)\,((\nu\, s, u, ctl_1, msg_1)\,(\mathsf{Client}_1 \mid \mathsf{Shop}_1) \mid \mathsf{Bank}_1) \mid \mathsf{Warehouse}_1)$$

$$\mathsf{Client}_1 \stackrel{\text{def}}{=} u\big[(\nu\, x)\,(\overline{x} \mid (x.\overline{s} + x.info(q).\overline{q} + x.(msg_1(done) \mid info(q))))\big]$$

$$\mathsf{Shop}_1 \stackrel{\text{def}}{=} s\big[\{\overline{u}\} \mid (\nu\, bOk, pOk)\,(\mathsf{Charge}_1 \mid \mathsf{Pack}_1 \mid bOk.pOk.\overline{msg_1}\langle done\rangle)\big] \mid \mathsf{Shop}$$

$$\mathsf{Charge}_1 \stackrel{\text{def}}{=} r\big[\{\overline{s}\} \mid \{\overline{q}\}| msg_2(x).(\overline{x} \mid (valid\%msg_2(refunded).\overline{bOk} + invalid.\overline{r}))\big]$$

$$\mathsf{Bank}_1 \stackrel{\text{def}}{=} q\big[\{\overline{r}\} \mid \overline{info}\langle q\rangle| (\nu\, x)\,(\overline{x} \mid (x\%\overline{msg_2}\langle refunded\rangle.\overline{msg_2}\langle valid\rangle + x.\overline{msg_2}\langle invalid\rangle))\big] \mid \mathsf{Bank}$$

$$\mathsf{Pack}_1 \stackrel{\text{def}}{=} p\big[\{\overline{s}\} \mid \{\overline{t}\}| msg_3(x).(\overline{x} \mid (packed\%msg_3(unpacked).\overline{pOk} + unavail.\overline{p}))\big]$$

$$\mathsf{Warehouse}_1 \stackrel{\text{def}}{=} t\big[\{\overline{p}\} \mid (\nu y)(\overline{y}| (y\%\overline{msg_3}\langle unpacked\rangle.\overline{msg_3}\langle packed\rangle + y.\overline{msg_3}\langle unavail\rangle))\big] \mid \mathsf{Warehouse}.$$

Names have been changed by α-conversions because of the scope extrusion within the handshakes. Note also that there are already installed compensations, e.g., **Charge** has installed $\{\overline{s}\}$ and $\{\overline{q}\}$ as compensations. If a failure occurs, these compensations ensure that other transactions are notified. For **Charge**, both **Shop** and **Bank** should be notified about its failure.

Since this is a successful execution, the **Bank** and the **Warehouse** confirm transaction success to the **Shop** and the **Client** is notified by the **Shop**, as depicted in Figure 10. Thus, let $\mathsf{OrderTransaction}_2$ denote the final process since

$$\mathsf{OrderTransaction}_1 \longrightarrow^* \mathsf{OrderTransaction}_2,$$

by applying successively the operational semantics rules presented in Section 4, with

$$\textsf{OrderTransaction}_2 \stackrel{\text{def}}{=} (\nu\, t, p, ctl_3, msg_3)\,((\nu\, info, r, q, ctl_2, msg_2)\ ((\nu\, s, u, ctl_1, msg_1)\,(\textsf{Client}_2 \mid \textsf{Shop}_2) \mid \textsf{Bank}_2) \mid \textsf{Warehouse}_2)$$

$$\textsf{Client}_2 \stackrel{\text{def}}{=} u\big[\mathbf{0}\big]$$

$$\textsf{Shop}_2 \stackrel{\text{def}}{=} s\big[\{\overline{u}\} \mid (\nu\, bOk, pOk)\,(\textsf{Charge}_2 \mid \textsf{Pack}_2)\big] \mid \textsf{Shop}$$

$$\textsf{Charge}_2 \stackrel{\text{def}}{=} r\big[\{\overline{s}\} \mid \{\overline{q}\} \mid \{msg_2(refunded)\}\big]$$

$$\textsf{Pack}_2 \stackrel{\text{def}}{=} p\big[\{\overline{s}\} \mid \{\overline{t}\} \mid \{msg_3(unpacked)\}\big]$$

$$\textsf{Bank}_2 \stackrel{\text{def}}{=} q\big[\{\overline{r}\} \mid (\{\overline{msg_2}\langle refunded\rangle\}\big] \mid \textsf{Bank}$$

$$\textsf{Warehouse}_2 \stackrel{\text{def}}{=} t\big[\{\overline{p}\} \mid \{\overline{msg_3}\langle unpacked\rangle\}\big] \mid \textsf{Warehouse}.$$

In spite of $\textsf{OrderTransaction}_2$ not being congruent with the inaction process $\mathbf{0}$, note that it is not reducible. Moreover, there are none active inputs (excluding transaction identifiers) inside any of the transaction scopes. Thus, we may say that transactions are completed.

An example of an execution where the client cancels the transaction can be seen in Figure 11, where the keyword *cancel* denotes the implicit cancelling of subtransactions.

As in the previous execution, this one starts by performing the handshakes. After that the **Bank** and the **Warehouse** notify the **Shop** by confirming the transaction. Let $\textsf{OrderTransaction}_3$ denote the resulting process just before the client cancels the **Shop** transaction. Thus,

$$\textsf{OrderTransaction} \longrightarrow^{*} \textsf{OrderTransaction}_3$$

by applying successively the operational semantics rules presented in Section 4, with

$$\textsf{OrderTransaction}_3 \stackrel{\text{def}}{=} (\nu\, t, p, ctl_3, msg_3)\,((\nu\, info, r, q, ctl_2, msg_2)\ ((\nu\, s, u, ctl_1, msg_1)\,(\textsf{Client}_3 \mid \textsf{Shop}_3) \mid \textsf{Bank}_3) \mid \textsf{Warehouse}_3)$$

$$\textsf{Client}_3 \stackrel{\text{def}}{=} u\big[\overline{s}\big]$$

$$\textsf{Shop}_3 \stackrel{\text{def}}{=} s\big[\{\overline{u}\} \mid (\nu\, bOk, pOk)\,(\textsf{Charge}_3 \mid \textsf{Pack}_3 \mid bOk.pOk.\overline{msg_1}\langle done\rangle)\big] \mid \textsf{Shop}$$

$$\textsf{Charge}_3 \stackrel{\text{def}}{=} r\big[\{\overline{s}\} \mid \{\overline{q}\} \mid \{msg_2(refunded)\} \mid \overline{bOk}\big]$$

$$\textsf{Bank}_3 \stackrel{\text{def}}{=} q\big[\{\overline{r}\} \mid \overline{info}\langle q\rangle \mid \{\,\overline{msg_2}\langle refunded\rangle\}\big] \mid \textsf{Bank}$$

$$\textsf{Pack}_3 \stackrel{\text{def}}{=} p\big[\{\overline{s}\} \mid \{\overline{t}\} \mid \{msg_3(unpacked)\} \mid \overline{pOk}\big]$$

$$\textsf{Warehouse}_3 \stackrel{\text{def}}{=} t\big[\{\overline{p}\} \mid \{\overline{msg_3}\langle unpacked\rangle\}\big] \mid \textsf{Warehouse}.$$

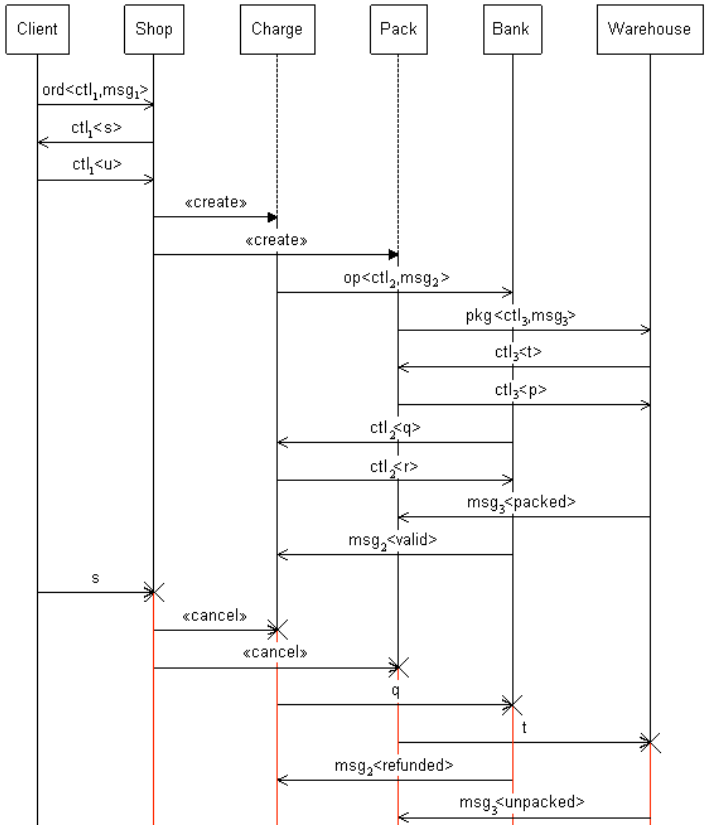

Fig. 11: An execution of the **OrderTransaction** with compensated transactions.

Note that all compensations are installed and that, after the client cancels the **Shop** transaction, compensations start to be activated. E.g., by applying the operational semantics rules presented in Section 4,

$$\mathbf{OrderTransaction}_3 \longrightarrow^* \mathbf{OrderTransaction}_4$$

where

$$\mathbf{OrderTransaction}_4 \stackrel{\text{def}}{=} (\nu\, t, p, ctl_3, msg_3)\,((\nu\, info, r, q, ctl_2, msg_2)\ ((\nu\, s, u, ctl_1, msg_1)\,(\mathbf{Client}_4 \mid \mathbf{Shop}_4) \mid \mathbf{Bank}_3) \mid \mathbf{Warehouse}_3)$$

$$\mathbf{Client}_4 \stackrel{\text{def}}{=} u\big[\mathbf{0}\big]$$

$$\mathbf{Shop}_4 \stackrel{\text{def}}{=} \langle \overline{u} \mid (\nu\, bOk, pOk)\,(\mathbf{Charge}_3 \mid \mathbf{Pack}_3)\rangle \mid \mathbf{Shop}$$

$$\mathbf{Charge}_4 \stackrel{\text{def}}{=} \langle \overline{s} \mid \overline{q} \mid msg_2(refunded)\rangle$$

$$\mathbf{Pack}_4 \stackrel{\text{def}}{=} \langle \overline{s} \mid \overline{t} \mid msg_3(unpacked)\rangle.$$

the compensations of transaction **Shop** and subtransactions **Charge** and **Pack** are activated. Then the compensations of the **Bank** and of the **Warehouse** are also activated and, by applying the operational semantics rules presented in Section 4, we are able to see the compensations taking effect.