A trace semantics for long-running transactions

Michael Butler[1], Tony Hoare[2], and Carla Ferreira[3]

[1] School of Electronics and Computer Science, University of Southampton, UK,
mjb@ecs.soton.ac.uk
[2] Microsoft Research Cambridge, UK
[3] Department of Computer Science, Technical University of Lisbon

January 2005

Abstract. A long-running transaction is an interactive component of a distributed system which must be executed as if it were a single atomic action. In principle, it should not be interrupted or fail in the middle, and it must not be interleaved with other atomic actions of other concurrently executing components of the system. In practice, the illusion of atomicity for a long-running transaction is achieved with the aid of compensation actions supplied by the original programmer: because the transaction is interactive, familiar automatic techniques of check-pointing and rollback are no longer adequate. This paper constructs a model of long-running transactions within the framework of the CSP process algebra, showing how the compensations are orchestrated to achieve the illusion of atomicity. It introduces a method for declaring that a process is a transaction, and for declaring a compensation for it in case it needs to be rolled back after it has committed. The familiar operator of sequential composition is redefined to ensure that all necessary compensations will be called in the right order if a later failure makes this necessary. The techniques are designed to work well in a highly concurrent and distributed setting. In addition we define an angelic choice operation, implemented by speculative execution of alternatives; its judicious use can improve responsiveness of a system in the face of the unpredictable latencies of remote communication. Many of the familiar properties of process algebra are preserved by these new definitions, on reasonable assumptions of the correctness and independence of the programmer-declared compensations.

1 Introduction

Business transactions involve hierarchies of activities whose execution needs to be orchestrated. Business transactions typically involve interactions and coordination between multiple partners. Business transactions need to deal with faults that arise at any stage of execution. In standard atomic transactions, such as database transactions, rollback mechanisms are used to protect against faults by providing all or nothing atomicity for transactions [7]. In long-running business transactions, rollback is not always possible because parts of a transaction will have been committed or because parts of a transaction (e.g., communications with external agents) are inherently impossible to undo using any automatic technique. The only solution in principle is to ask the system designer to provide ways of compensating actions that cannot be undone automatically. A language for long-running transactions can provide constructs through which the application developer declares compensations for actions. The language will then orchestrate the compensations in the appropriate way to achieve the desired effect.

In the context of business transactions, Gray and Reuter [7] define a compensation as the action taken to recover from error or cope with a change of plan. Consider the following example: a client buys some books in an on-line bookstore

and the bookstore debits the client's account as the payment for the book order. The bookstore later realises that one of the books in the client's order is out of print. To compensate the client for this problem, the bookstore can credit the account with the amount wrongfully debited and send a letter apologising for their mistake. This example shows that compensation is more general than traditional rollback in database transactions. Compensation is important when a system cannot control everything, such as when interaction with other agents (including humans) is involved. Garcia-Molina and Salem [6] use compensation to define the concept of *sagas*. A saga partitions a long-running transaction into a sequence of several smaller subtransactions, where each of the subtransactions has an associated compensation. If one of the subtransactions in the sequence aborts, the compensation associated with those committed subtransactions is executed in reverse order.

This paper constructs a model of long-running transactions within the framework of the CSP process algebra [8], showing how the compensations are orchestrated to achieve the illusion of atomicity. Section 2 of this paper gives an introduction to the Compensating CSP language. Section 3 provides a description of the standard trace semantics of the sequential and the concurrent operators of CSP, slightly adapted to the needs of our model. The three following sections put together ideas from the standard semantics to construct the transaction processing model, and prove the relevant theorems.

Our compensation constructs are not intended to replace atomic transactions. Instead they extend transaction mechanisms to a higher level of granularity. The goal of our design is that shorter-running transactions should be nested inside longer-running transactions, so as to deal with many levels of granularity, from milliseconds to (say) months. Backtracking will be minimised, by use of compensations at the appropriate level of granularity, so as to preserve as much progress-to-date as possible. Where possible, basic activities of a long-running transaction could be implemented as atomic transactions with automatic rollback rather than explicit compensation.

The inspiration of this paper derives from the transaction processing features of Microsoft Biztalk [11], IBMs WSFL [10], IBM's Business Process Beans [4], Structured Activity Compensation [3] and the OASIS draft standard for BPEL4WS [5]. However no attempt has been made to model the particular semantics of any of these languages.

2 Compensating CSP

The behaviour of an interactive process (typically denoted $P, Q, \ldots$) can be recorded as a sequential trace (typically denoted $p, q, \ldots$) of all its environmentally observable actions (typically denoted $A, B, \ldots$), and of certain special internal actions (like $\checkmark$, indicating successful termination of a process). For example, the trace $\langle A, B, \checkmark \rangle$ is a behaviour of the process $A;B$ that executes action A, then action B and then terminates successfully. In the CSP process algebra, processes are modelled using such traces [8]. The traces of composite processes, such as a sequential composition $(P;Q)$ or a parallel composition $(P \parallel Q)$, are defined in terms of the traces of their constituent processes. The trace model means that each action that occurs cannot be anything but atomic in the two usual senses: (1) it either occurs as a whole, or it does not occur at all; (2) it occurs either wholly before or wholly after every other action.

If a long-running transaction actually fails before successfully completing, the effect must be as if it had not occurred at all. In a conventional (short running) transaction system, the effect of the transaction can be undone at any time by restoring a checkpoint of local state that has been taken before its start. But a long-running transaction may have interacted with the real world before failing,

and the real world cannot be check-pointed. To solve this problem, the programmer of the original transaction is asked to provide for each fine-grained action A a compensation action (often called A°); its occurrence after the action A will restore the world to a state which is an acceptable approximation to the state that it had before the start of the transaction. Thus the primitive component of a long-running transaction can be written $A \div A^\circ$, where A is a fine-grained atomic action, and A° is its compensation, which will be invoked if a failure later in the transaction makes it necessary. Since a complete transaction P is an atomic action at a coarser level of granularity, it too may be declared to have its own compensation, for example $P \div Q$. The coarse-grained compensation Q over-rides the fine-grained compensations declared inside P.

An implementation of a transaction processing system must ensure that on failure of a transaction, all the necessary atomic compensations are performed in an appropriate order to compensate for the effect of everything that has actually happened so far. For example, if a failure occurs after sequential execution of the two fine-grained actions $\langle A, B\rangle$, the compensations should occur in the reverse order $\langle B^\circ, A^\circ\rangle$. To model this strategy, we distinguish between standard processes $P, Q, \ldots$), and compensable processes ($PP, QQ, \ldots$). We represent a behaviour ($pp, qq, \ldots$) of a compensable process using a pair of sequential traces with a forward part and a compensation part. For example, the trace pair $(\langle A, B, \checkmark\rangle, \langle B^\circ, A^\circ, \checkmark\rangle)$ is a behaviour of the process $(A \div A^\circ)$; $(B \div B^\circ)$. Sequential composition of compensable processes is redefined in a non-standard way to ensure that the compensations for all actions performed will be accumulated in the reverse order to their original performance. Parallel composition of compensable processes is defined so that compensations for performed actions will be accumulated in parallel.

Failure of a transaction is signified by another special symbol $!$, which appears like $\checkmark$ at the end of a trace. The intended effect of the $!$ event is to throw an interrupt. For example, the primitive process $THROW$ which fails immediately contains the trace $\langle ! \rangle$. In a purely sequential process, the exception causes an immediate disruption to the flow of control. An interrupt handler may be used to catch interrupts: in $P \rhd Q$, an interrupt raised by P triggers execution of the handler Q. In parallel processes, the whole group of parallel processes may fail when one of the processes throws an exception and all the other processes are willing to disrupt their flow of control and yield to the exception. A process that is ready to terminate (indicated by $\checkmark$) is also willing to yield to an interrupt. A process may also yield at mid points in its execution, indicated by the special symbol $?$ which again appears at the end of a trace. Parallel composition is defined so that $!$ in one process synchronises with $!$, $\checkmark$ or $?$ in another process and the combined event is $!$. A compensation pair $P \div Q$ is always willing to yield to an interrupt either before starting P or immediately after completing P. For example, $A \div A^\circ$ will contain the compensable behaviours $(\langle ? \rangle, \langle \checkmark \rangle)$ and $(\langle A, \checkmark\rangle, \langle A^\circ, \checkmark\rangle)$.

A complete transaction is formed from a compensable process PP by enclosing PP in a transaction block $[PP]$. This converts PP back into a standard process. The standard behaviours of a transaction block $[PP]$ are defined in terms of the compensable behaviours of PP. Successful forward traces of PP represent successful completion of the whole transaction. The compensations are no longer needed, and they are discarded. The failed traces of PP need to involve actual execution of the compensations. The intention in forming a complete transaction from a compensable process is that, in the case of failure, the compensations will cancel all the forward actions, leaving only a trace containing no observable actions as a result. We introduce a framework for proving that a transaction either does nothing, because its forward actions will have been cancelled, or completes successfully. This is the fundamental principle for a process algebra that models long-running transactions. In these proofs, we will assume that any trace is equivalent to one in which any

Standard processes:

$P, Q ::= A$		(atomic action)
	$\mid P \ ; \ Q$	(sequential composition)
	$\mid P \ \Box \ Q$	(choice)
	$\mid P \parallel Q$	(parallel composition)
	$\mid SKIP$	(normal termination)
	$\mid THROW$	(throw an interrupt)
	$\mid YIELD$	(yield to an interrupt)
	$\mid P \rhd Q$	(interrupt handler)
	$\mid [PP]$	(transaction block)

Compensable processes:

$PP, QQ ::= P \div Q$		(compensation pair)
	$\mid PP \ ; \ QQ$	
	$\mid PP \ \Box \ QQ$	
	$\mid PP \parallel QQ$	
	$\mid SKIPP$	
	$\mid THROWW$	
	$\mid YIELDD$	

Fig. 1. Syntax of Compensating CSP

action and its following compensation have been cancelled. The unrealism of this abstraction should be mitigated in engineering practice, by ensuring that failures with less desirable compensations are adequately rare.

External choice ($P \ \Box \ Q$) is defined in our model as the union of the traces of the alternatives P and Q, just as in CSP. In implementation, the choice is made between P and Q according to whichever of them is the first to be able to start. This choice operation is often used to mitigate the unpredictable variations in latency that are characteristic of remote interactions on the world wide web. In a transaction processing system, further improvement is possible, by delaying the choice until the first of P and Q have not only started but completed; the actions of the other are then just compensated. This strategy is a kind of speculative execution; it has been called optimistic scheduling in distributed system simulation. Its definition is the final achievement of this paper.

To keep the semantic definitions simple in this paper, we have avoided supporting synchronised communication between parallel processes. Synchronisation in parallel process blocks is limited to joint execution of compensations, joint termination and joint interruption. Dealing with synchronised communication is a desirable longer term aim.

The syntax of compensating CSP is summarised in Figure 1. Figure 2 presents a transaction for processing of customer orders in the compensating CSP language. The first step in the transaction is a compensation pair. The primary action of this pair is to accept the order and deduct the order quantity from the inventory database. The compensation action simply adds the order quantity back to the total in the inventory database. After an order is received from a customer, the order is packed for shipment, and a courier is booked to deliver the goods to the customer. The *PackOrder* process packs each of the items in the order in parallel. Each *PackItem* activity can be compensated by a corresponding *UnpackItem*. Simultaneously with the packing of the order, a credit check is performed on the customer. The credit check is performed in parallel because it normally succeeds, and in this normal case the company does not wish to delay the order unnecessarily. In the case that a credit check fails, an interrupt is thrown causing the transaction to

$$
\begin{aligned}
\mathit{OrderTransaction} &= [\,\mathit{ProcessOrder}\,] \\
\mathit{ProcessOrder} &= (\mathit{AcceptOrder} \div \mathit{RestockOrder})\,;\ \mathit{FulfillOrder} \\
\mathit{FulfillOrder} &= \mathit{BookCourier} \div \mathit{CancelCourier} \ \|\ \\
&\quad\ \mathit{PackOrder} \ \|\ \\
&\quad\ \mathit{CreditCheck}\ ;\ (\ \mathit{Ok};\ \mathit{SKIPP} \\
&\qquad\qquad \Box\ \mathit{NotOk};\ \mathit{THROWW}\ ) \\
\mathit{PackOrder} &= \|\, i \in \mathit{Items} \ \bullet\ (\mathit{PackItem}(i) \div \mathit{UnpackItem}(i))
\end{aligned}
$$

Fig. 2. Order transaction example

stop its execution, with the courier possibly having been booked and possibly some of the items having being packed. In case of failure, the semantics of the transaction block will ensure that the appropriate compensation activities will be invoked for those activities that did take place.

3 Trace semantics for standard processes

We assume a process has an alphabet of actions Σ which does not include any of the special events in $\Omega = \{\checkmark, !, ?\}$. For traces s and t, we write st for their concatenation. Standard processes are defined as non-empty sets of traces each of the form $s\langle\omega\rangle$ where $s \in \Sigma^*$ and $\omega \in \Omega$. Thus all traces of standard processes are of one of the following forms:

- $s\langle\checkmark\rangle$ trace leading to normal termination
- $s\langle !\,\rangle$ trace leading to interrupt throw
- $s\langle ?\,\rangle$ trace leading to interrupt yield

Unlike the traces model for CSP in [8], we include only completed traces in our traces model, not prefixes of traces. This simplifies many definitions since the nature of a trace is indicated by its final symbol.

3.1 Sequential operators

The process that performs a single atomic event and terminates successfully consists of a single complete trace:

Definition 1 (Atomic action). *For* $A \in \Sigma$, $\qquad A \ =\ \{\,\langle A, \checkmark\rangle\,\}$

As in CSP the choice between two process is defined as the union of their traces:

Definition 2 (Choice). $\qquad P \,\Box\, Q \ =\ P \cup Q$

With sequential composition $P;Q$, execution of Q commences when P has completed successfully; thus successful traces of P are extended with traces of Q, while other traces of P remain unchanged. We define a sequential operator on traces and then lift it to processes in the following way:

Definition 3 (Sequential composition).

$$
\begin{aligned}
p\langle\checkmark\rangle\ ;\ q &= pq \\
p\langle\omega\rangle\ ;\ q &= p\langle\omega\rangle, \ \textbf{where}\ \omega \neq \checkmark
\end{aligned}
$$

$$
P\,;\,Q \ =\ \{\,p\,;\,q \mid p \in P \ \wedge\ q \in Q\,\}
$$

The process *SKIP* immediately terminates successfully:

Definition 4 (Skip). $SKIP = \{ \langle \checkmark \rangle \}$

THROW is the process that immediately raises an interrupt. *YIELD* is the process that yields or terminates. These processes are defined as follows:

Definition 5 (Throw and yield).

$$THROW = \{ \langle ! \rangle \} \qquad YIELD = \{ \langle ? \rangle, \langle \checkmark \rangle \}$$

The process $P; YIELD; Q$ may yield to an interrupt from the environment after executing P and before executing Q.

Sequential processes satisfy the following laws:

$$\begin{aligned} P;(Q \,\Box\, R) &= (P;Q) \,\Box\, (P;R) \\ (P \,\Box\, Q);R &= (P;R) \,\Box\, (Q;R) \\ P;(Q;R) &= (P;Q);R \\ P;SKIP &= P \\ SKIP;P &= P \\ THROW;P &= THROW \\ YIELD;YIELD &= YIELD \end{aligned}$$

We look now at defining an operator for handling interrupts. For processes P and Q, $P \rhd Q$ represents a process that behaves as P until an interrupt is raised by P, at which point it behaves as Q. The interrupt handling operator is defined as follows:

Definition 6 (Interrupt handler).

$$\begin{aligned} p\langle ! \rangle \rhd q &= pq \\ p\langle \omega \rangle \rhd q &= p\langle \omega \rangle, \textbf{ where } \omega \neq ! \end{aligned}$$

$$P \rhd Q = \{ p \rhd q \mid p \in P \wedge q \in Q \}$$

Laws for interrupt handling:

$$\begin{aligned} (P \rhd Q) \rhd R &= P \rhd (Q \rhd R) \\ SKIP \rhd P &= SKIP \\ YIELD \rhd P &= YIELD \\ THROW \rhd P &= P \end{aligned}$$

3.2 Concurrency

In this paper we do not support synchronous execution of observable actions. A parallel block of processes will synchronise only on joint termination or joint interruption. We represent this by defining a synchronisation operator on the special terminal events from the set Ω. If ω and ω' are terminal events of distinct concurrent processes, we denote by $\omega \& \omega'$ the joint terminal event of their concurrent execution. Evaluations of this operator are enumerated in Table 1. The first three rows of the table show that the synchronisation of an interrupt throw with any other terminal event results in an interrupt throw. The next two rows show that the synchronisation of a yield with either a yield or a successful termination result in a yield. The first five rows are motivated by our decision that if a process is willing to

ω	ω'	$\omega \& \omega'$
!	!	!
!	?	!
!	✓	!
?	?	?
?	✓	?
✓	✓	✓

Table 1. Synchronisation of terminal events

terminate (in any of the three ways), then it is willing to yield to an interrupt from its environment. The last row of Table 1 shows that a pair of parallel processes may terminate successfully when both processes are willing to terminate successfully. We also define the synchronisation operator to be commutative; from this and from Table 1 it can be seen that the operator is well-defined for all operands in the set Ω. Case analysis shows the synchronisation operator to be associative.

As usual in process algebra, we model asynchronous execution of actions in separate processes as occurring in an interleaved fashion. Asynchronous actions can lead to different interleavings; for example, $A \parallel B$ can execute A followed by B or B followed by A. For traces p and q, we write $p \mid\mid\mid q$ to denote the set of all interleaving of p and q:

$$\begin{aligned} p \mid\mid\mid \langle\rangle &= \{p\} \\ \langle\rangle \mid\mid\mid q &= \{q\} \\ \langle x\rangle p \mid\mid\mid \langle y\rangle q &= \{\, \langle x\rangle r \mid r \in (p \mid\mid\mid \langle y\rangle q) \,\} \cup \{\, \langle y\rangle r \mid r \in (\langle x\rangle p \mid\mid\mid q) \,\} \end{aligned}$$

We define parallel composition of traces to be the set of interleavings of their observable part followed by the synchronisation of their terminal events. This is then lifted to sets of traces to define parallel composition of processes:

Definition 7 (Parallel composition).

$$p\langle\omega\rangle \parallel q\langle\omega'\rangle = \{\, r\langle\omega \& \omega'\rangle \mid r \in (p \mid\mid\mid q) \,\}$$

$$P \parallel Q = \{\, r \mid r \in (p \parallel q) \wedge p \in P \wedge q \in Q \,\}$$

Parallel composition is commutative and associative:

$$\begin{aligned} P \parallel Q &= Q \parallel P \\ (P \parallel Q) \parallel R &= P \parallel (Q \parallel R) \end{aligned}$$

If P does not contain any yields, then $\mathit{YIELD}; P$ is only willing to yield to an interrupt either before P commences or when P terminates. This is shown in the following law (for P not containing any yields):

$$\mathit{THROW} \parallel (\mathit{YIELD}; P) = \mathit{THROW} \;\Box\; P; \mathit{THROW}$$

This law shows that interrupt does *not* have priority over other events. This is what we would expect in a distributed setting where we cannot expect an entire distributed system to respond immediately to an attempt by one party to raise an exception.

4 Compensable processes

A compensable process contains forward behaviour and compensation behaviour. The intention is that the compensation can be executed to compensate for the forward action, if necessary (e.g., when an error or interrupt occurs later). Compensable behaviour is modelled by pairs of traces of the form $(p\langle\omega\rangle, p'\langle\omega'\rangle)$, where $p\langle\omega\rangle$ represents a forward trace and $p'\langle\omega'\rangle$ represents the corresponding compensation trace. A compensable process is modelled by a non-empty set of such pairs.

The choice of compensable processes is as for standard processes:

Definition 8 (Compensable choice).

$$PP \,\Box\, QQ \;=\; PP \cup QQ$$

Parallel composition of compensable processes is similar to the standard case:

Definition 9 (Compensable parallel composition).

$$(p, p') \parallel (q, q') \;=\; \{\; (r, r') \mid r \in (p \parallel q) \;\wedge\; r' \in (p' \parallel q') \;\}$$

$$PP \parallel QQ \;=\; \{\, rr \mid rr \in (pp \parallel qq) \wedge pp \in PP \wedge qq \in QQ \,\}$$

We redefine the sequential composition operator so that the compensation behaviour of the first process is made to happen after that of the second process. Behaviours of PP where the forward trace is unsuccessful remain unchanged.

Definition 10 (Compensable sequential composition).

$$\begin{aligned}(p\langle\checkmark\rangle, p') \,;\, (q, q') &= (pq,\; q'; p')\\ (p\langle\omega\rangle, p') \,;\, (q, q') &= (p\langle\omega\rangle, p'), \quad \textbf{where } \omega \neq \checkmark\end{aligned}$$

$$PP \,;\, QQ \;=\; \{\, pp \,;\, qq \mid pp \in PP \wedge qq \in QQ \,\}$$

A compensation pair is a compensable process constructed from two standard processes. In the pair $P \div Q$, successfully terminating forward behaviour of P is augmented by compensation behaviour from Q resulting in a compensable process. If P throws or yields, the compensation is empty. The rationale for our definition is that a compensation is intended to be used to compensate, at a later stage, for a successfully completed forward unit of work and not for an interrupted unit of work. As before we define the pairing operator on compensable behaviours and then lift it to processes. When lifting to processes, we include an extra behaviour which allows the compensation pair to yield immediately with the empty compensation. The operator is defined as follows:

Definition 11 (Compensation pair).

$$\begin{aligned}p\langle\checkmark\rangle \div q &= (p\langle\checkmark\rangle, q)\\ p\langle\omega\rangle \div q &= (p\langle\omega\rangle, \langle\checkmark\rangle), \quad \textbf{where } \omega \neq \checkmark\end{aligned}$$

$$\begin{aligned}P \div Q \;=\; &\{\, (\langle ?\rangle, \langle\checkmark\rangle) \,\} \;\cup\\ &\{\, p \div q \mid p \in P \wedge q \in Q \,\}\end{aligned}$$

The operators on compensable processes are designed to ensure the correct compensation is accummulated even when an interrupt is yielded to. For example, consider the traces of the following process:

$$\begin{aligned} A \div A';\ B \div B' \ =\ \{\, & (\langle ?\rangle, \langle \checkmark \rangle), \\ & (\langle A,\ ?\rangle, \langle A', \checkmark\rangle), \\ & (\langle A, B, \checkmark\rangle, \langle B', A', \checkmark\rangle)\ \ \} \end{aligned}$$

If this process yields immediately, the compensation is empty. If it yields after executing A, the compensation is A'. If it completes successfully, the compensation is B' followed by A'.

Definition 12 (Compensable basic processes).

$$\begin{aligned} SKIPP &= SKIP \div SKIP \\ THROWW &= THROW \div SKIP \\ YIELDD &= YIELD \div SKIP \end{aligned}$$

Laws:

$$\begin{aligned} PP \parallel QQ &= QQ \parallel PP \\ (PP \parallel QQ) \parallel RR &= PP \parallel (QQ \parallel RR) \\ (PP\,;QQ)\,;RR &= PP\,;(QQ\,;RR) \\ PP\,;SKIPP &= PP \\ SKIPP\,;PP &= PP \\ THROWW\,;PP &= THROWW \\ YIELDD\,;(P \div Q) &= P \div Q \end{aligned}$$

A transaction block involves running the compensation part of interrupted forward traces, discarding the compensation parts of terminating forward traces and completely removing traces whose forward parts are yielding. A transaction block converts a compensable process into a standard process:

Definition 13 (Transaction block).

$$\begin{aligned} [PP] \ =\ & \{\, pp' \mid (p\langle !\rangle, p') \in PP \,\} \ \cup \\ & \{\, p\langle \checkmark\rangle \mid (p\langle \checkmark\rangle, p') \in PP \,\} \end{aligned}$$

Note that non-emptiness of PP is not sufficient to ensure non-emptiness of $[PP]$. If PP only contained yielding behaviours, then $[PP]$ would be empty. The following healthiness conditions, declaring that all processes P and PP consist of some terminating or interrupting behaviour, will ensure that $[PP]$ is non-empty:

- $p\langle \checkmark\rangle \in P$ or $p\langle !\rangle \in P$, for some p
- $(p\langle \checkmark\rangle, p') \in PP$ or $(p\langle !\rangle, p') \in PP$, for some p, p'

These conditions are true of the basic processes and are preserved by all the operators.

The transaction block masks interrupts and yields in forward behaviour:

$$\begin{aligned} [THROWW] \ &=\ SKIP \\ [YIELDD] \ &=\ SKIP \end{aligned}$$

Assume P is non-yielding. The following laws show that installed compensation is run in the case of an interrupt and discarded in the case of successful termination:

$$\begin{aligned} [\ P \div P'\ ;\ THROWW\ ] &= P;P' \\ [\ P \div P'\ ] &= P \end{aligned}$$

Assume P, P', Q, Q' terminate successfully, neither raising nor yielding to interrupts. The following laws show the effect of the parallel and sequential composition operators on the order of compensations:

$$\begin{aligned} [\ P \div P'\ ;\ Q \div Q'\ ;\ THROWW\ ] &= P;Q;Q';P' \\ [\ (P \div P' \parallel Q \div Q')\ ;\ THROWW\ ] &= (P \parallel Q)\ ;\ (P' \parallel Q') \\ [\ (P \div P'\ ;\ Q \div Q') \parallel THROWW\ ] &= SKIP \ \Box\ (P;P') \ \Box\ (P;Q;Q';P') \end{aligned}$$

$$\begin{aligned} &[\ P \div P' \parallel Q \div Q' \parallel THROWW\ ] = \\ &\quad SKIP \ \Box\ (P;P') \ \Box\ (Q;Q') \ \Box\ (P \parallel Q);(P' \parallel Q') \end{aligned}$$

5 Cancellation semantics for transactions

So far we have said very little about the relationship between forward actions and their compensations other than the relative order in which they may occur. In this section we develop a theory of cancellation for compensable processes in which the effect of forward actions is cancelled by compensation actions. We take a very abstract view of cancellation in which we can declare that an atomic action, say A, is compensated by A° and that the behaviour exhibited by A followed by A° is the same as *SKIP*. We will introduce a cancellation function that removes cancelling forward and compensation actions from process traces. We will introduce a correctness criteria on compensable processes which says they should be *self-cancelling*. We will introduce a rule which says that when the cancellation function is applied to a self-cancelling transaction, then the overall effect is either to perform the normal forward behaviour of the transaction or to do nothing (*SKIP*). We will show under what conditions the self-cancellation property is preserved by the operators of our language.

Assume F is a set of forward actions and C is a set of compensation actions with F and C being disjoint. We assume that *cancel* is a relation between F and C so that $cancel(A, A^\circ)$ means that A° cancels the effect of A. We can also declare that certain actions are independent so that they can occur in either order. This would typically be the case for compensations of parallel processes. We write $independent(A, B)$ to indicate that A and B may be transposed in a trace as they do not interfere with each other. We assume that *independent* is symmetric (unlike *cancel*).

We now define our cancellation function ($\mathcal{C}$) on traces. If a trace t is of the form $p\langle A\rangle q\langle A^\circ\rangle r$ and if $cancel(A, A^\circ)$ and $\forall\, B \in q \cdot independent(A^\circ, B)$, then:

$$\mathcal{C}(\ p\langle A\rangle q\langle A^\circ\rangle r\ ) = \mathcal{C}(\ pqr\ )$$

If trace t does not satisfy the above conditions then no further cancellation can be applied:

$$\mathcal{C}(t) = t, \ \textbf{otherwise}$$

For example, assuming A°, B° and C° cancel A, B and C respectively and A° and B° are independent:

$$\mathcal{C}(\ \langle A, B, C, C^\circ, A^\circ, B^\circ\rangle\ ) = \mathcal{C}(\ \langle A, B, A^\circ, B^\circ\rangle\ )$$

$$
\begin{aligned}
&= \mathcal{C}(\,\langle A, A^\circ\rangle\,), \quad \text{since } \mathit{independent}(A^\circ, B^\circ)\\
&= \mathcal{C}(\,\langle\rangle\,)\\
&= \langle\rangle
\end{aligned}
$$

Cancellation is lifted to processes by mapping the cancellation function to each trace. We refer to a transaction block to which cancellation has being applied, $\mathcal{C}[PP]$, as being *closed*.

A compensation behaviour $(p\langle\omega\rangle, p'\langle\omega'\rangle)$ is self-cancelling if the forward and compensation parts together are equivalent to the empty trace and the compensation terminates sucessfully:

$$
\mathit{self\_cancelling}(p\langle\omega\rangle, p'\langle\omega'\rangle) \quad = \quad \mathcal{C}(pp') = \langle\rangle \ \wedge \ \omega' = \checkmark
$$

A compensable process PP is self-cancelling, $\mathit{self\_cancelling}(PP)$, when all its behaviours are self cancelling. Self-cancelling transactions enjoy some important properties. If we force an interrupt, then the closed transaction behaves simply as $SKIP$:

$$
\frac{\mathit{self\_cancelling}(PP)}{\mathcal{C}[\,PP; THROWW\,] \ = \ SKIP} \tag{1}
$$

The closure of a self-cancelling transaction either completes a forward trace successfully or, if an exception occurs, terminates immediately with no observable effect:

$$
\frac{\mathit{self\_cancelling}(PP)}{\mathcal{C}[PP] \ \subseteq \ PP_\checkmark \,\Box\, SKIP} \tag{2}
$$

Here, $PP_\checkmark$ represents successfully completing executions of PP:

$$
PP_\checkmark \ = \{\, t\langle\checkmark\rangle \mid (t\langle\checkmark\rangle, t') \in PP \,\}
$$

Inequality arises in rule (2) because PP might not have any successful behaviours or might not have interrupted behaviours. This rule is quite powerful as it allows us to reason separately about the normal behaviour and the compensation behaviour of a closed transaction block. The abstract specification of a transaction block might be to achieve a certain goal or to do nothing. We verify this by verifying that $PP_\checkmark$ achieves that goal and by verifying that PP is self-cancelling.

The following rules allow $PP_\checkmark$ to be derived through simple structural calculation:

$$
\begin{aligned}
(A \div A^\circ)_\checkmark &= A\\
(PP \,\Box\, QQ)_\checkmark &= PP_\checkmark \,\Box\, QQ_\checkmark\\
(PP \parallel QQ)_\checkmark &= PP_\checkmark \parallel QQ_\checkmark\\
(PP\,;\ QQ)_\checkmark &= PP_\checkmark \ ; \ QQ_\checkmark\\
THROWW_\checkmark &= NULL
\end{aligned}
$$

Here $NULL$ stands for the empty set of traces. $NULL$ does not correspond to a valid process but is a useful calculational artefact. $NULL$ satisfies the following laws:

$$
\begin{aligned}
NULL\,;\ PP &= NULL\\
PP\,;\ NULL &= NULL\\
NULL \parallel PP &= NULL\\
NULL \,\Box\, PP &= PP
\end{aligned}
$$

$$ProcessOrder_{\checkmark} = AcceptOrder \,;\, FulfillOrder_{\checkmark}$$

$$\begin{aligned} FulfillOrder_{\checkmark} = {} & BookCourier \;\; \| \\ & PackOrder_{\checkmark} \;\; \| \\ & CreditCheck \;;\; Ok \end{aligned}$$

$$PackOrder_{\checkmark} = \|\, i \in Items \;\bullet\; PackItem(i)$$

Fig. 3. Forward behaviour for order transaction example

The final law above shows that $NULL$ is absorbed by choice. This means that the result of applying cancellation to a self-cancelling transaction block (rule (2) above) is a well defined process even if $PP_{\checkmark} = NULL$. Figure 3 shows the result of calculating the forward behaviour of the order process example of Figure 2.

We look now at how self cancellation relates to the operators of our language.

$$cancel(A, A^{\circ}) \;\Rightarrow\; self\_cancelling(\, A \div A^{\circ}\,)$$

$SKIPP$, $THROWW$ and $YIELDD$ are all self-cancelling. Self-cancellation is preserved by sequential composition and choice:

$$\frac{self\_cancelling(PP) \quad self\_cancelling(QQ)}{self\_cancelling(PP; QQ)} \qquad \frac{self\_cancelling(PP) \quad self\_cancelling(QQ)}{self\_cancelling(PP \,\Box\, QQ)}$$

Parallel composition preserves self-cancellation provided the compensations from parallel processes are independent:

$$\frac{\begin{array}{c} self\_cancelling(PP) \\ self\_cancelling(QQ) \\ \forall\, A \in comp(PP),\, B \in comp(QQ) \cdot independent(A, B) \end{array}}{self\_cancelling(PP \parallel QQ)}$$

Here, $comp(PP)$ represents the set of compensation actions of PP.

From the above rules, we see the result that, if the programmer of a transaction ensures

- an action A is directly paired with its compensation A° and
- every compensation is independent of compensations in parallel processes,

then the transaction will be self-cancelling under our theory.

6 Speculative choice

When the goal of a transaction can be achieved in different ways, responsiveness may be improved by attempting these different means in parallel. When one attempt succeeds, the other attempts may be abandoned. Compensation can be used to cancel the effect so far of the abandoned attempts. In this section, we define a form of speculative choice which can be shown to be equivalent to standard choice under the right conditions.

We write $PP \boxtimes QQ$ for the speculative choice of PP and QQ. The effect of $PP \boxtimes QQ$ is to run the forward behaviour of PP and QQ in parallel until one of

them terminates successfully. If PP terminates successfully, then the compensation accumulated for QQ is run while the compensation for PP is preserved:

$$(p\langle\checkmark\rangle, p') \boxtimes (q\langle\omega\rangle, q') = \{ (rq', p') \mid r \in (p \mid\mid\mid q) \}$$

Here and below we assume $\omega, \omega' \neq \checkmark$. Trace r above represents any interleaving of the forward trace p with the forward trace q. The compensation q' is run immediately, i.e., appended to r, while the compensation trace p' is preserved. The case where QQ terminates successfully is similar:

$$(p\langle\omega\rangle, p') \boxtimes (q\langle\checkmark\rangle, q') = \{ (rp', q') \mid r \in (p \mid\mid\mid q) \}$$

Behaviours in which both processes terminate successfully result in a choice between one or the other succeeding:

$$\begin{aligned}(p\langle\checkmark\rangle, p') \boxtimes (q\langle\checkmark\rangle, q') = {} & \{ (rq', p') \mid r \in (p \mid\mid\mid q) \} \cup \\ & \{ (rp', q') \mid r \in (p \mid\mid\mid q) \}\end{aligned}$$

Behaviours in which neither terminate successfully are also, in which case the compensations are run in parallel:

$$(p\langle\omega\rangle, p') \boxtimes (q\langle\omega'\rangle, q') = \{ (rr', \langle\checkmark\rangle) \mid r \in (p \mid\mid\mid q) \wedge r' \in (p' \parallel q') \}$$

The operator on compensable behaviours is lifted to compensable processes:

Definition 14 (Speculative choice).

$$PP \boxtimes QQ = \{ pp \boxtimes qq \mid pp \in PP \wedge qq \in QQ \}$$

To illustrate the effect of the operator, consider the following example transaction block containing speculative choice:

$$[\, A \div A' \boxtimes B \div B' \,] = A \;\Box\; B \;\Box\; ((A \parallel B) ; (A' \Box B'))$$

Here, either A succeeds (because $B \div B'$ yields immediately) or B succeeds or both succeed with one of A or B being compensated.

If PP and QQ are self-cancelling and their compensations are independent, then their speculative choice is self-cancelling:

$$\frac{\begin{array}{c} \mathit{self\_cancelling}(PP) \\ \mathit{self\_cancelling}(QQ) \\ \forall\, A \in \mathit{comp}(PP),\ B \in \mathit{comp}(QQ) \cdot \mathit{independent}(A, B) \end{array}}{\mathit{self\_cancelling}(PP \boxtimes QQ)}$$

Under the same conditions, a transaction block consisting of $PP \boxtimes QQ$ is the same as one consisting of $PP \;\Box\; QQ$:

$$\frac{\begin{array}{c} \mathit{self\_cancelling}(PP) \\ \mathit{self\_cancelling}(QQ) \\ \forall\, A \in \mathit{comp}(PP),\ B \in \mathit{comp}(QQ) \cdot \mathit{independent}(A, B) \end{array}}{\mathcal{C}[\; PP \boxtimes QQ \;] = \mathcal{C}[\; PP \;\Box\; QQ \;]}$$

Unlike our other operators, speculative choice is not associative. For example consider the process $(A \div A' \boxtimes B \div B') \boxtimes C \div C'$ and the case where B succeeds overall. This case results in the compensations for the non-succeeding branches being run in the order A' then C'. On the other hand, if B succeeds overall in the process $A \div A' \boxtimes (B \div B' \boxtimes C \div C')$, then the compensations for the non-succeeding branches will be run in the order C' then A'. We could get around this problem by defining an n-ary version of the operator which would select one succeeding branch, if possible, and run the compensations for the other branches in parallel.

7 Related Work

Korth *et al.* [9] define compensating transactions as a way to overcome the limitations of atomicity when dealing with long-running transactions. The authors propose the use of compensating transactions to allow access to uncommitted data and to undo committed transactions. In their work compensation is formalized in terms of the properties it has to guarantee. Consider a transaction T, its compensating transaction CT, and a set of dependent transactions on T (dependent transactions of T are those transactions that read data values written by T). The authors say that a compensation is sound when "compensation does not disturb the outcome of dependent transactions", i.e., the compensation has to:

- reverse the effects of execution of T, and
- assure the outcome of the dependent transactions after the execution of the CT must be the same as if the transaction T did not occur.

As the definition of compensation soundness can be too restrictive the authors present a definition for weaker forms of soundness. Clearly, there are similarities between [9] and our cancellation semantics. One main difference is that [9] does not provide a rich language as the work presented here does. Transaction's operations are limited to reading or writing a set of data, as the focus is on transactional databases.

Two of the authors (Butler and Ferreira) developed the StAC (Structured Activity Compensation) language [2, 3] for modelling long-running business transactions which includes compensation constructs. An important difference between StAC and the work presented here is that instead of the execution of compensations being part of the definition of a transaction block, StAC has explicit primitives for running or discarding installed compensations (*reverse* and *accept* respectively). StAC gives a precise interpretation to the mechanics of compensation, including the combination of compensation with parallel execution, hierarchy and exceptions. However, the design of the language does not lend itself to reasoning about the intended effect of a transaction in a compositional way. In particular the separation of the *accept* and *reverse* operators from compensation scoping prevents the definition of a compositional semantics: the semantics of the reverse operator cannot be defined on its own as its behaviour depends on the context in which it is called. These shortcomings were addressed in the work presented here.

Recently Bruni et al [1] have developed an operational semantics for a language with similar operators to ours, including compensation pairs and transaction blocks (or sagas as they call them). Like our work, and unlike StAC, the execution of compensation is part of the definition of a saga which leads to a neater operational semantics. They provide a richer form of exception than us whereby whether or not compensations were run in a saga is visible outside the saga. They also define a form of speculative choice similar to ours.

8 Conclusions

The operators of our language are quite powerful in the way they take care of orchestration of compensation and interrupt handling in a nested way. By working with a trace semantics we have developed a language that supports compensation in the desired way and has a compositional semantics supporting modular reasoning about long-running transactions. Our cancellation semantics is somewhat purist but we believe it points towards what should be achievable with a language for long-running transactions that is designed with correctness in mind. In particular, the way in which the cancellation semantics allows reasoning about normal behaviour

and compensation behaviour to be separated is very powerful. The design of our proposed structures has been through many iterations, in which we have sought simpler and simpler formal definitions. We have also tried to make definitions of each feature logically independent of every other feature, so as to reduce the risk of complex interaction effects.

Compensating CSP can be regarded as a design pattern for a tightly-disciplined form of error handling for transactions. The advantage of a special orchestration language is that the implementation is responsible for avoiding the deadlocks and race conditions that almost universally accompany a programmer's attempt to implement the necessary error recovery protocols.

For this paper we have chosen to use a simple trace semantics making strong use of the special terminal events. This trace semantics allowed us to develop simple elegant definitions of the operators which facilitated the proof of the various laws. However we have avoided modelling several important and well understood features of process algebras for concurrent and distributed systems. In particular we have avoided synchronous communication, event hiding and the distinction between internal and external choice. These will require a richer semantic model and now that we have achieved a better grasp of compensation through the trace model, we are in a better position to tackle these other features in combination with compensation. In our self-cancellation rule for compensation pairs, we have only allowed for pairs of atomic actions. To deal with the more general case, our current belief is that we need a semantic model that admits a notion of event refinement where an atomic event at a course level of granularity is replaced by a whole process at a finer-grained level.

Acknowledgements

Thanks to Peter Welch, Marc Shapiro, Roberto Bruni, Hernan Melgratti, Peter Henderson, Mandy Chessell, David Vines and Catherine Griffin for valuable discussion on compensation and exceptions. Thanks to the anonymous referee for suggesting improvements in the presentation and thanks to Bertrand Meyer for pointing out that 'compensable' was preferable to 'compensatable'.

References

1. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL 2005*, 2005.
2. M. Butler and C. Ferreira. A process compensation language. In *Integrated Formal Methods(IFM'2000)*, volume 1945 of *LNCS*, pages 61 – 76. Springer-Verlag, 2000.
3. M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Coordination 2004*, volume 2949 of *LNCS*. Springer-Verlag, 2004.
4. M. Chessell, D. Vines, C. Griffin, V. Green, and K. Warr. Business process beans: System design and architecture document. Technical report, Transaction Processing Design and New Technology Development Group, IBM UK Laboratories, January 2001.
5. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services, version 1.1. `http://www-106.ibm.com/developerworks/library/ws-bpel/`, 2003.
6. H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM SIGMOD*, pages 249–259, 1987.
7. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.
8. C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

9. H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *16th VLDB Conference*, Brisbane, Australia, 1990.
10. F. Leymann. Web services flow language, version 1.0. `http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`, 2001. IBM.
11. B. Metha, M. Levy, G. Meredith, T. Andrews, B. Beckman, J. Klein, and A. Mital. BizTalk Server 2000 Business Process Orchestration. *IEEE Data Engineering Bulletin*, 24(1):35–39, 2001.