# General dynamic recovery for compensating CSP

**Article** *in* Electronic Proceedings in Theoretical Computer Science · March 2014

DOI: 10.4204/EPTCS.143.1



**2 authors**, including:

Abeer Al-Humaimeedy
King Saud University
**13** PUBLICATIONS **195** CITATIONS

# General dynamic recovery for compensating CSP

Abeer S. Al-Humaimeedy
Department of Informatics
King's College London, United Kingdom
Department of Information Technology
King Saud University, Riyadh, Saudi Arabia
Abeer@ksu.edu.sa

Maribel Fernández
Department of Informatics
King's College London, United Kingdom
Maribel.Fernandez@kcl.ac.uk

Compensation is a technique to roll-back a system to a consistent state in case of failure. Recovery mechanisms for compensating calculi specify the order of execution of compensation sequences. Dynamic recovery means that the order of execution is determined at runtime. In this paper, we define an extension of Compensating CSP, called DEcCSP, with general dynamic recovery. We provide a formal, operational semantics for the calculus, and illustrate its expressive power with a case study. In contrast with previous versions of Compensating CSP, DEcCSP provides mechanisms to replace or discard compensations at runtime. Additionally, we bring back to DEcCSP standard CSP operators that are not available in other compensating CSP calculi, and introduce channel communication.

## 1 Introduction

Transactions are units of work comprising a set of interactions between entities to achieve a final output [8]. The notion of a transaction is based on the idea of all-or-nothing, that is, none of the transaction's effects can take place until the whole transaction is committed.

Basically, there are two types of transactions: *Atomic Transaction (AT)* and *Long Running Transaction (LRT)* [8]. ATs prevent entities from updating system resources until the whole transaction is committed. Checkpoints and resources' key-locks are used to maintain systems in a safe state. LRTs relax the previous condition and allow the entities to update resources. However, LRTs use compensations to maintain systems in a safe state. Compensation is a technique to roll-back the system to a consistent state in the case of failure.

LRTs are intensively used in complex systems where entities usually engage in transactions that last for hours, days or even longer while resources cannot be locked for such a long time. As a result, modelling languages for complex systems should be equipped with techniques to implement LRTs.

Considering process calculi as modelling languages, compensation has emerged as a crucial need. The fundamental idea behind compensating process calculi is to adapt the well-developed transaction techniques from database theory to the theory of process calculi, by introducing primitives to model and handle transactions. In essence, the key concepts introduced within process calculi are: scope, fault, termination and compensation. *Scope* defines the transaction boundaries. Issues to be considered in relation with scope are: the relation between transaction's subprocesses and the fate of subtransactions if the parent transaction terminates (if nested transactions are allowed). Subtransactions can be aborted (i.e., terminated), discarded (i.e., deleted), or preserved (i.e., subtransactions are levelled up and continue running). *Fault* represents an exception (internal fault) thrown by a process; fault handlers are procedures that should be evaluated in such a case. *Termination* is the state of a process which is either committed or interrupted by other processes (external fault); termination handlers are procedures that should be evaluated in such a case. Finally, *compensation* is the reverse behaviour of a process, to undo the effects

B. Löwe, G. Winskel (Eds.):
Developments in Computational Models 2012 (DCM 2012)
EPTCS 143, 2014, pp. 3–16, doi:10.4204/EPTCS.143.1

of the normal execution in the case of a failure. Compensations are evaluated when the process needs to roll-back. Issues to be considered when dealing with compensation are the installation of a new compensation in the system, and the recovery mechanism which determines the evaluation order of the compensations to recover the system to a consistent state after a failure.

Basically, compensation can be statically implemented in any process calculus by creating a new process, which captures a fixed compensation scenario (planned in advance). Static compensation is feasible in systems where the evaluation of processes is fixed. However, in complex systems where there are interleaved, parallel and complex patterns of interactions, the compensation scenario heavily depends on the execution order. Therefore, compensating process calculi have been proposed as a suitable solution for modelling such complex systems. The fundamental idea of such process calculi is introducing a new type of processes called *compensable processes*.

A compensable process comprises two behaviours: the *forward behaviour* corresponds to the normal execution of the process, and the *compensation behaviour* corresponds to its reverse execution, which will undo the effects of the normal execution in case of system failure. While the system is running, a sorted compensation scenario (sorted according to the execution order) will be built incrementally from individual reverse behaviours.

Compensation has been introduced in a range of process calculi, including CCS [19], $\pi$-calculus [14, 18, 13], CSP [4] and Sagas [3, 2]. These compensating process calculi are either interaction-based or flow-composition calculi [1]. Interaction-based calculi associate with each transaction explicit compensation sequences, and new compensations are installed in the system as an update to the compensation process by using explicit primitives like ($inst\lfloor\rfloor$) [13]. In flow-composition calculi the compensation sequence is built as a composition of smaller compensable components. Attached to each subprocess is a compensation component; when this subprocess is successfully terminated its compensation is composed (sequentially or in parallel) to the compensation sequence. This is to undo the effects of this subprocess in case of system failure.

We are interested in flow-composition calculi, where when a transaction fails, compensations are activated in the order specified by the recovery mechanisms. We distinguish between *static recovery*, which is the activation of a previously implemented compensation sequence, and *dynamic recovery*, which is the activation of a dynamically generated compensation sequence (i.e., generated while the system is running). In turn, dynamic recovery mechanisms can be classified as: *parallel recovery*, if all compensations are executed in parallel; *backward recovery*, if parallel processes are compensated in parallel and sequential processes are compensated in backward order; and *general dynamic recovery*, which is backward recovery with the option of replacing or discarding compensations at runtime [18, 12, 13].

Specifically, we focus on CSP as a modelling language for complex systems, because of its flexible communication patterns and because it provides simple reasoning mechanisms to verify significant properties of models, such as good/bad traces, deadlock freedom and divergence.

Compensation has been introduced in CSP by Butler, Hoare, and Ferreira who defined compensating CSP (cCSP) as a flow-composition calculus with a backward recovery mechanism [4]. cCSP has been extended by Chen, Liu, and Wang in the Extended compensating CSP (EcCSP), bringing back some significant operators from the original CSP and developing a theory of refinement [6, 7].

In this paper we extend cCSP further by introducing primitives to facilitate general dynamic recovery. We call the new calculus DEcCSP (Dynamic EcCSP). Improving the recovery mechanism from backward recovery to general dynamic recovery allows compensations to be replaced or discarded after they have been recorded. This is useful in many cases, such as: (i) The compensation process is unknown at the start. (ii) The compensation process is subject to change while the process evolves. (iii) The

compensation's logic is complex and spans several processes. We demonstrate some of these cases in Section 5.

Additionally, DEcCSP extends EcCSP by including all the standard CSP operators, to facilitate the specification of complex systems. DEcCSP provides a conditional (if-then-else), iteration (while-do), prefixing operator, named processes and channels. Channels have been used informally in extensions of CSP [4, 5, 6, 7, 15]. We have extended the syntax of DEcCSP with primitives for channel communication, and adapted the semantics to allow processes to pass data (the latter is omitted due to lack of space).

The remainder of this paper is organised as follows: Section 2 recalls cCSP. Section 3 provides an operational semantics for EcCSP, which had so far only a denotational semantics. The operational semantics for EcCSP is used as a basis for the design, in section 4, of the syntax and the operational semantics of our calculus, DEcCSP. Section 5 illustrates its expressive power using a case study. Finally, Section 6 concludes the paper and briefly discusses future directions.

## 2 Background: compensating CSP (cCSP)

In this section we recall the main concepts in cCSP [4], assuming the reader is familiar with CSP [16, 17].

The novelty in cCSP is the introduction of transaction processing features within the standard CSP processes. cCSP categorises processes into two types: standard processes, which are a subset of standard CSP processes, and compensable processes, where a standard process is attached to another standard process to undo its effects. When a standard process terminates normally, it evolves to $\varnothing$, which means nothing to do; however, when a compensable process terminates normally then its compensation will be preserved in case the transaction fails and the system needs to roll-back.

The syntax of cCSP is summarised in Figure 1. We describe below the main constructs.

The operator **[ ]** is used to identify transaction's boundaries in cCSP. Transactions can be nested; subtransactions should be aborted if the parent transaction is terminated. According to the semantics of cCSP, transaction blocks cannot be interrupted. cCSP includes operators for handling the key concepts of compensations presented in the introduction. To represent unsuccessful termination of a process, new terminal signals (events) have been added to the calculus: **(**!**)** represents internal fault; **(**?**)** represents yielding to external fault. In addition to these terminal events new primitive processes have been introduced: THROW is a process that throws an exception then terminates; YIELD is a process that yields to an external exception and terminates. A new operator $\triangleright$ has been added to implement fault handler (named interrupt handler). In $p \triangleright q$, $q$ is the fault handler of $p$. Compensable process can be defined in the calculus as a pair of standard processes which are composed with the new operator $\div$. In $p \div q$, $q$ is the compensation handler of $p$. The $\varnothing$ primitive process is added to the syntax of cCSP to represent a process which does nothing. It is equal to STOP in the original CSP. cCSP processes can be composed sequentially or in parallel. Parallel processes can synchronise on terminal events solely. Choice between two processes is resolved by either of them performing an event.

We are now going to describe the operational semantics of these operators and primitive processes; it is based on the semantics presented by Ripon and Butler [5], but we have adapted it to follow the operational semantics of the standard CSP [16, 17]. Throughout what follows, the following notations will be used. Standard processes will be referred to by using lower case letters $p$, $q$. Compensable processes will be referred to by using double letters $pp$, $qq$. The set $\Sigma$ is the universal set that contains all the observable events in a system; $a$, $b$,... will be used to range over this set. The set $\Omega$ consists of the terminal events $\{!, ?, \surd\}$; $\omega$ will be used to range over this set. We define $\Sigma^{\tau} := \Sigma \cup \{\tau\}$ and $\Sigma^{\tau\Omega} := \Sigma^{\tau} \cup \Omega$. Finally, capital letters $A$, $B$,.. will denote sets of observable events.

| | (Standard Processes) | | | (Compensable Processes) | |
|---|---|---|---|---|---|
| $p,q ::=$ | $a$ | (Atomic process) | $pp,qq ::=$ | $p \div q$ | (Compensation pair (CP)) |
| | $\mid p \Box q$ | (Choice operator) | | $\mid pp \Box qq$ | (Choice operator) |
| | $\mid p;q$ | (Sequential composition) | | $\mid pp;qq$ | (Sequential composition) |
| | $\mid p \parallel q$ | (Parallel composition) | | $\mid pp \parallel qq$ | (Parallel composition) |
| | $\mid$SKIP | (Primitive process) | | $\mid$SKIPP | (Primitive process) |
| | $\mid$THROW | (Primitive process) | | $\mid$THROWW | (Primitive process) |
| | $\mid$YIELD | (Primitive process) | | $\mid$YIELDD | (Primitive process) |
| | $\mid p \triangleright q$ | (Interrupt handler) | | | |
| | $\mid \varnothing$ | (Primitive process) | | | |
| | $\mid [pp]$ | (Transaction block) | | | |

Figure 1: cCSP Syntax

The following are *standard processes*: *primitive processes* are $\frac{}{\text{SKIP} \xrightarrow{\surd} \varnothing}$, $\frac{}{\text{THROW} \xrightarrow{!} \varnothing}$, and $\frac{}{\text{YIELD} \xrightarrow{\omega} \varnothing}$ for $\omega \in \{?, \surd\}$. For $a \in \Sigma$, the following is process $a$: $\frac{}{a \xrightarrow{a} \text{SKIP}}$. If $a,b \in \Sigma^{\omega\tau}$, then the following two processes are called *Choice*: $\frac{p \xrightarrow{a} p'}{p \Box q \xrightarrow{a} p'}$ and $\frac{q \xrightarrow{b} q'}{p \Box q \xrightarrow{b} q'}$. If $a \in \Sigma^{\tau}$ and $\omega \in \{!, ?\}$, the following three processes are called *Sequential Composition*: $\frac{p \xrightarrow{a} p'}{p;q \xrightarrow{a} p';q}$, $\frac{p \xrightarrow{\surd} p'}{p;q \xrightarrow{\tau} q}$, and $\frac{p \xrightarrow{\omega} \varnothing}{p;q \xrightarrow{\omega} \varnothing}$. For $a \in \Sigma^{\tau}$ and $\omega \in \{\surd, ?\}$, the following three processes are called *Interrupt Handler*: $\frac{p \xrightarrow{a} p'}{p \triangleright q \xrightarrow{a} p' \triangleright q}$, $\frac{p \xrightarrow{!} p'}{p \triangleright q \xrightarrow{\tau} q}$, and $\frac{p \xrightarrow{\omega} \varnothing}{p \triangleright q \xrightarrow{\omega} \varnothing}$. We define the binary operation $(\omega, \omega') \mapsto \omega \& \omega'$ by the following table:

| | ! | ? | $\surd$ |
|---|---|---|---|
| ! | ! | ! | ! |
| ? | ! | ? | ? |
| $\surd$ | ! | ? | $\surd$ |

Then, for $b,c \in \Sigma^{\tau}$ and $\omega, \omega' \in \Omega$, the following three processes are called *Parallel Composition*: $\frac{p \xrightarrow{b} p'}{p \parallel q \xrightarrow{b} p' \parallel q}$, $\frac{q \xrightarrow{c} q'}{p \parallel q \xrightarrow{c} p \parallel q'}$, and $\frac{p \xrightarrow{\omega} \varnothing \quad q \xrightarrow{\omega'} \varnothing}{p \parallel q \xrightarrow{\omega \& \omega'} \varnothing}$. Finally, the following three processes are called *Transaction block*: $\frac{pp \xrightarrow{a} pp'}{[pp] \xrightarrow{a} [pp']}$, $\frac{pp \xrightarrow{!} p}{[pp] \xrightarrow{!} p}$, and $\frac{pp \xrightarrow{\surd} p}{[pp] \xrightarrow{\surd} \varnothing}$. This finishes the list of *standard processes*.

We now define the *compensable processes*: For $\omega \in \{!, ?\}$, we call the following three processes *compensation pair*: $\frac{p \xrightarrow{a} p'}{p \div q \xrightarrow{a} p' \div q}$, $\frac{p \xrightarrow{\surd} \varnothing}{p \div q \xrightarrow{\surd} q}$, and $\frac{p \xrightarrow{\omega} \varnothing}{p \div q \xrightarrow{\omega} \text{SKIP}}$. Furthermore, for $\omega \in \{?, \surd\}$, the following are *primitive processes*: SKIPP $=$ SKIP $\div$ SKIP, THROWW $=$ THROW $\div$ SKIP, YIELDD $=$ YIELD $\div$ SKIP, $\frac{}{\text{SKIPP} \xrightarrow{\surd} \text{SKIP}}$, $\frac{}{\text{THROWW} \xrightarrow{!} \text{SKIP}}$, and $\frac{}{\text{YIELDD} \xrightarrow{\omega} \text{SKIP}}$.

| (Standard Processes) | | | (Compensable Processes) | | |
|---|---|---|---|---|---|
| $p,q ::=$ | $\ldots$ | (cCSP syntax) | $pp,qq ::=$ | $\ldots$ | (cCSP syntax) |
| | $\mid p \sqcap q$ | (Internal choice) | | $\mid pp \sqcap qq$ | (Internal choice) |
| | $\mid p \parallel q$ | (Parallel composition) | | $\mid pp \parallel qq$ | (Parallel composition) |
| | $\mid p \underset{A}{\setminus} A$ | (Hiding operator) | | $\mid pp \underset{A}{\setminus} A$ | (Hiding operator) |
| | $\mid p[\![R]\!]$ | (Renaming operator) | | $\mid pp[\![R]\!]$ | (Renaming operator) |
| | $\mid \mu p.f(p)$ | (Recursion) | | $\mid \mu pp.ff(pp)$ | (Recursion) |
| | | | | $\mid pp \boxtimes qq$ | (Speculative choice) |

Figure 2: EcCSP Syntax

If $\omega,\omega' \in \Omega$ and $a,b \in \Sigma^\tau$, we call the following processes *Choice*: $\frac{pp \xrightarrow{a} pp'}{pp \Box qq \xrightarrow{a} pp'}$, $\frac{qq \xrightarrow{b} qq'}{pp \Box qq \xrightarrow{b} qq'}$, $\frac{pp \xrightarrow{\omega} p}{pp \Box qq \xrightarrow{\omega} p}$, and $\frac{qq \xrightarrow{\omega'} q}{pp \Box qq \xrightarrow{\omega'} q}$. If $a \in \Sigma^\tau$ and $\omega \in \{!,?\}$, then the following three processes are called *Sequential Composition*: $\frac{pp \xrightarrow{a} pp'}{pp;qq \xrightarrow{a} pp';qq}$, $\frac{pp \xrightarrow{\surd} p}{pp;qq \xrightarrow{\tau} \langle qq,p\rangle}$, and $\frac{pp \xrightarrow{\omega} p}{pp;qq \xrightarrow{\omega} p}$. For $a \in \Sigma^\tau$ and $\omega \in \Omega$, the following two processes are called *the auxiliary operator:* $\frac{qq \xrightarrow{a} qq'}{\langle qq,p\rangle \xrightarrow{a} \langle qq',p\rangle}$ and $\frac{qq \xrightarrow{\omega} q}{\langle qq,p\rangle \xrightarrow{\omega} q;p}$. Finally, if $b,c \in \Sigma^\tau$ and $\omega,\omega' \in \Omega$, then the following three processes are called *Parallel Composition*: $\frac{pp \xrightarrow{b} pp'}{pp \parallel qq \xrightarrow{b} pp' \parallel qq}$, $\frac{qq \xrightarrow{c} qq'}{pp \parallel qq \xrightarrow{c} pp \parallel qq'}$, and $\frac{pp \xrightarrow{\omega} p \quad qq \xrightarrow{\omega'} q}{pp \parallel qq \xrightarrow{\omega\&\omega'} p \parallel q}$.

## 3 Extended cCSP (EcCSP)

Chen, Liu, and Wang extended cCSP, adapted its trace semantics and developed stable-failure semantics and failure-divergence semantics for cCSP as in the standard CSP [6, 7]. They also brought back to the syntax of cCSP the original CSP operators: hiding, renaming, non-deterministic choice and recursion. In addition, they changed the parallel operator to be synchronous, and introduced speculative choice ($\boxtimes$). A preliminary semantics for speculative choice was presented in [4, 15], however, it was not included in the original cCSP. The EcCSP syntax is summarised in Figure 2.

EcCSP was developed using denotational semantics in [6, 7] and no operational semantics was provided. Therefore, we developed an operational semantics for EcCSP based on the operational semantics of cCSP and CSP. We below describe the new and the adaptive inference rules; cCSP rules which are part of cCSP operational semantics and not listed here are adopted without modification.

The adaptive operators of *standard processes* are as follows: Atomic process semantics has been adjusted to permit interruptions before or after performing its event. Therefore, for $a \in \Sigma$, if a completed process $a$ is $\frac{}{a \xrightarrow{a} \text{SKIP}}$, then an interrupted process $a$ is as follows: *before event a:* $\frac{}{a \xrightarrow{?} \text{STOP}}$, *after event a:* $\frac{}{a \xrightarrow{a} \text{STOP}}$.

Choice, which can be deterministic or non-deterministic as in the standard CSP. Deterministic choice is resolved by either of the processes performing an observable or terminal event as follows. If $a,b \in \Sigma^{\omega}$, then the following four processes are called *External (Deterministic) Choice*: $\frac{p \xrightarrow{a} p'}{p \Box q \xrightarrow{a} p'}$, $\frac{q \xrightarrow{b} q'}{p \Box q \xrightarrow{b} q'}$, $\frac{p \xrightarrow{\tau} p'}{p \Box q \xrightarrow{a} p' \Box q}$, and $\frac{q \xrightarrow{\tau} q'}{p \Box q \xrightarrow{b} p \Box q'}$.

On the other hand, non-deterministic choice is resolved by either of the processes performing the silent event "$\tau$" as follows: $\frac{}{p \sqcap q \xrightarrow{a} p}$, and $\frac{}{p \sqcap q \xrightarrow{\tau} q}$.

The parallel operator, which has been parameterised with a set of events. The purpose of this set is to govern the synchronisation between participants. Thereby, every event in this set should be performed simultaneously, other events can interleave in any order. If $b,c \in \Sigma^{\tau}$, and $\omega,\omega' \in \Omega$, then the following processes are defining the new *Parallel Composition*: For $b,c \notin A$, $\frac{p \xrightarrow{b} p'}{p \underset{A}{\parallel} q \xrightarrow{b} p' \underset{A}{\parallel} q}$ and $\frac{q \xrightarrow{c} q'}{p \underset{A}{\parallel} q \xrightarrow{c} p \underset{A}{\parallel} q'}$. For $a \in A$, $\frac{p \xrightarrow{a} p' \quad q \xrightarrow{a} q'}{p \underset{A}{\parallel} q \xrightarrow{b} p' \underset{A}{\parallel} q}$. If the binary operation $(\omega,\omega') \mapsto \omega \& \omega'$ as defined in Section 2, then $\frac{p \xrightarrow{\omega} \mathrm{STOP} \quad q \xrightarrow{\omega'} \mathrm{STOP}}{p \parallel q \xrightarrow{\omega \& \omega'} \mathrm{STOP}}$. Transaction block semantics has been adjusted to permit interruptions by adding the following rule: $\frac{pp \xrightarrow{?} p}{[pp] \xrightarrow{?} p}$.

The new operators of *standard processes* are as follows: *Hiding*, where a predefined set of events turns to "$\tau$" in the targeted process. If $b \notin (A \subseteq \Sigma)$, then $\frac{p \xrightarrow{b} p'}{p \backslash A \xrightarrow{b} p' \backslash A}$, if $a \in (A \subseteq \Sigma)$, then $\frac{p \xrightarrow{a} p'}{p \backslash A \xrightarrow{\tau} p' \backslash A}$, and finally if $\omega \in \Omega$, then $\frac{p \xrightarrow{\omega} \mathrm{STOP}}{p \backslash A \xrightarrow{\omega} \mathrm{STOP}}$. *Renaming*, where the events of a process (or a subset of them) are renamed according to a renaming relation (in other words, if $R \subseteq \Sigma \times \Sigma$ is a defined renaming relation which maps events in set $A$ to the events in set $B$, then renaming process $p$ with $R$ is mapping its events from $A$ to $B$). The following processes define the *Renaming operator*. If $(aRb)$, then $\frac{p \xrightarrow{a} p'}{p[\![R]\!] \xrightarrow{b} p'[\![R]\!]}$, $\frac{p \xrightarrow{\tau} p'}{p[\![R]\!] \xrightarrow{\tau} p'[\![R]\!]}$, and if $\omega \in \Omega$, then $\frac{p \xrightarrow{\omega} \mathrm{STOP}}{p[\![R]\!] \xrightarrow{\omega} \mathrm{STOP}}$. *Recursion*, using a fixed point operator as follows: $\frac{}{\mu p.f(p) \xrightarrow{\tau} f[\mu p.f(p)/p]}$. Symmetrically, we update the choice and parallel composition operators for compensable processes as we did for standard processes as following:

If $\omega,\omega' \in \Omega$ and $a,b \in \Sigma$, the following six processes are called *External (Deterministic) Choice*: $\frac{pp \xrightarrow{a} pp'}{pp \Box qq \xrightarrow{a} pp'}$, $\frac{qq \xrightarrow{b} qq'}{pp \Box qq \xrightarrow{b} qq'}$, $\frac{pp \xrightarrow{\omega} p}{pp \Box qq \xrightarrow{\omega} p}$, $\frac{qq \xrightarrow{\omega'} q}{pp \Box qq \xrightarrow{\omega'} q}$, $\frac{pp \xrightarrow{\omega} p}{pp \Box qq \xrightarrow{\omega} p}$, and $\frac{qq \xrightarrow{\omega'} q}{pp \Box qq \xrightarrow{\omega'} q}$; furthermore, the following two processes are called *Internal (Non-deterministic) Choice*: $\frac{pp \xrightarrow{\tau} pp'}{pp \Box qq \xrightarrow{\tau} pp' \Box qq}$ and $\frac{qq \xrightarrow{\tau} qq'}{pp \Box qq \xrightarrow{\tau} pp \Box qq'}$.

If $b,c \in \Sigma^{\tau}$ and $\omega,\omega' \in \Omega$, then the following three processes are called *Parallel Composition*. For

$b,c \notin A$, $\dfrac{pp \xrightarrow{b} pp'}{pp \underset{A}{\parallel} qq \xrightarrow{b} pp' \underset{A}{\parallel} qq}$, $\dfrac{qq \xrightarrow{c} qq'}{pp \underset{A}{\parallel} qq \xrightarrow{c} pp \underset{A}{\parallel} qq'}$; for $a \in A$, $\dfrac{pp \xrightarrow{a} pp' \qquad qq \xrightarrow{a} qq'}{pp \underset{A}{\parallel} qq \xrightarrow{a} pp' \underset{A}{\parallel} qq'}$; and finally, for $\omega,\omega' \in \Omega$, $\dfrac{pp \xrightarrow{\omega} p \qquad qq \xrightarrow{\omega'} q}{pp \underset{A}{\parallel} qq \xrightarrow{\omega\&\omega'} p \underset{A}{\parallel} q}$.

Additionally, we define a new processes to implement the new operators *(Recursion, Hiding, and Renaming)* for compensable processes as we did for standard processes as following: *Recursion* is the process $\dfrac{}{\mu pp.ff(pp) \xrightarrow{\tau} ff[\mu pp.ff(pp)/pp]}$. *Hiding* are the following processes: If $b \notin (A \subseteq \Sigma)$, then $\dfrac{pp \xrightarrow{b} pp'}{pp\backslash A \xrightarrow{b} pp'\backslash A}$, and if $a \in (A \subseteq \Sigma)$, then $\dfrac{pp \xrightarrow{a} pp'}{pp\backslash A \xrightarrow{\tau} pp'\backslash A}$, and finally if $\omega \in \Omega$, then $\dfrac{pp \xrightarrow{\omega} p}{pp\backslash A \xrightarrow{\omega} p\backslash A}$. Finally, *Renaming* are the following processes: If $(aRb)$, then $\dfrac{pp \xrightarrow{a} pp'}{pp[\![R]\!] \xrightarrow{b} pp'[\![R]\!]}$, $\dfrac{pp \xrightarrow{\tau} pp'}{pp[\![R]\!] \xrightarrow{\tau} pp'[\![R]\!]}$, and if $\omega \in \Omega$, then $\dfrac{pp \xrightarrow{\omega} p}{p[\![R]\!] \xrightarrow{\omega} p[\![R]\!]}$.

We also define the following processes to implement the *Speculative choice*. *Speculative choice* can be defined as if two processes run in parallel without synchronisation, then the choice is resolved when one of them commits, and the other should immediately compensate. If both of them fail then the whole choice will fail and the processes should compensate in parallel.

If $a,b \in \Sigma$ and $\omega \in \{!,?\}$, then the following six processes are called *Speculative Choice*: $\dfrac{pp \xrightarrow{a} pp'}{pp \boxtimes qq \xrightarrow{a} pp' \boxtimes qq}$, $\dfrac{qq \xrightarrow{b} qq'}{pp \boxtimes qq \xrightarrow{b} pp \boxtimes qq'}$, $\dfrac{pp \xrightarrow{\omega} p \qquad qq \xrightarrow{\omega'} q}{pp \boxtimes qq \xrightarrow{\omega\&\omega'} \langle(\omega\&\omega'),(p \parallel q)\rangle}$, $\dfrac{pp \xrightarrow{\surd} p \qquad qq \xrightarrow{\omega} q}{pp \boxtimes qq \xrightarrow{\surd} \langle q,p\rangle}$, $\dfrac{qq \xrightarrow{\surd} q \qquad pp \xrightarrow{\omega} p}{pp \boxtimes qq \xrightarrow{\surd} \langle p,q\rangle}$, and $\dfrac{pp \xrightarrow{\surd} p \qquad qq \xrightarrow{\surd} q}{pp \boxtimes qq \xrightarrow{\surd} \langle q,p\rangle \Box \langle p,q\rangle}$. For $a \in \Sigma^{\tau}$ and $\omega \in \Omega$, the following two processes are called *the auxiliary operator:* $\dfrac{p \xrightarrow{a} p'}{\langle p,q\rangle \xrightarrow{a} \langle p',q\rangle}$ and $\dfrac{p \xrightarrow{\omega} \mathrm{STOP}}{\langle p,q\rangle \xrightarrow{\omega} q}$.

## 4 Dynamic Extended cCSP (DEcCSP)

Improving the compensation recovery mechanism from backward recovery to general dynamic recovery allows compensations not only to be recorded in the right order dynamically, but to be discarded or replaced dynamically too. In this section we extend EcCSP to include primitives that facilitate general dynamic recovery. The main idea is to use a free process variable instead of the reverse behaviour process in compensation pairs. The variable will work as a place holder within the recovery sequence, where the real content can be retrieved later at the start of the execution. This will give the designer the ability to replace variable values whenever needed or discard them if they are no longer needed as long as the compensation has not been activated yet. Compensation can be discarded by assigning SKIP to the variable, where the SKIP process in fact equals to an empty compensation.

The use of a process variable to update compensations is inspired by the work of Guidi, Lanese, Montesi, and Zavattaro to model fault handling in SOCK [10] (a service-oriented process calculus) [9]. This idea has also been applied to the $\pi$-calculus [12, 13]. However, these calculi are interaction-based, whereas DEcCSP is a flow-composition calculus. Because of this difference, we have followed a different

| (Standard Processes) | | | (Compensable Processes) | | |
|---|---|---|---|---|---|
| $p, q ::=$ | $\ldots$ | (EcCSP syntax) | $pp, qq ::=$ | $\ldots$ | (EcCSP syntax) |
| | $\mid$ If $b$ Then $p$ Else $q$ | (condition block) | | $\mid$ If $b$ Then $pp$ Else $qq$ | (condition block) |
| | $\mid$ while $b$ do $p$ | (Iteration block) | | $\mid$ while $b$ do $pp$ | (Iteration block) |
| | $\mid N$ | (Process name) | | $\mid NN$ | (Process name) |
| | $\mid a \longrightarrow p$ | (Prefix operator) | | $\mid p \div X$ | (Variable CP) |
| | $\mid X := p$ | (Variable assignment) | | | |

| (Events) | | | | | |
|---|---|---|---|---|---|
| $a ::=$ | Names | $\mid$ $a?\ell$ | $\mid$ $a!e$ | $\mid$ $a.e$ | |
| $Names ::=$ | name | $\mid$ name.Names | | | |

Figure 3: DEcCSP Syntax. Here, $e$ is a standard integer expression, $b$ is a standard Boolean expression, and $\ell$ is an integer variable.

strategy to update and discard compensations.

In addition to improving the recovery mechanism, we bring back the remainder of the CSP standard operators. These include: conditional (if-then-else) and iteration (while-do) control blocks, prefixing operator, and named processes. Although control blocks can be simulated in CSP using the primitive operators as Hoare shows [11], Hoare also argues in [11] that having a reasonably wide range of operators is needed in practice.

We also introduce channels passing data: we extend the syntax with channel communication primitives, and adapt the semantics to allow processes to pass data (due to lack of space we present only the syntax of channels carrying data, and omit the operational semantics). Butler, Ripon, Chen, Liu, and Wang do not include data in their calculi [4, 5, 6, 7]. Channels have been used informally in Ripon's case study [15].

The rest of this section will be devoted to presenting DEcCSP's syntax and operational semantics in sections 4.1 and 4.2 respectively.

### 4.1 DEcCSP syntax

The syntax of DEcCSP is given in Figure 3. DEcCSP extends EcCSP's syntax with operators to facilitate general dynamic recovery, as explained above. These operators include: (i) Assignment, to assign values to process variables. (ii)Variable compensation pair, where a process variable will take the place of the compensation in the usual compensation pair.

DEcCSP also includes the standard CSP operators: *if-then-else*, *while-do*, prefixing operator and named processes $N$. Process names can be used to specify recursive processes. We also extend the EcCSP syntax with the standard primitives of channel communications in CSP. We assume the communicated data are integers or ordinary names.

Events in CSP can be classified into *ordinary names* (a single constant name describing an action to be performed); *compound names*, which are built by composing ordinary names with other ordinary names (e.g, $a.b$) or with an integer expression (e.g, $a.4$) to denote data that is passed without indicating the direction; and *channels*, where the composition operator (.) is replaced by one of (!,?) to indicate the direction of communication via the channel: if $a$ is a channel, then $a?\ell$ denotes input on the variable $\ell$, which will be recorded in the local store, and $a!e$ denotes output of the value of the integer expression $e$.

The traditional input/output notations $(?, !)$ coincide with the notations for terminal events $(?, !)$ introduced by Butler, Hoare, and Ferreira [4], however it will always be clear from the context which one is intended.

### 4.2 Operational semantics of DEcCSP

We present below the operational semantics of DEcCSP, based on the operational semantics that we developed for EcCSP in Section 3. To deal with variables, we introduce stores to keep track of the different values of these variables. Stores are denoted by $S$, where $S$ is a collection of typed locations. Variables will have a location in this store to hold its current value. The store $S$ is represented as a function $S[\ell \mapsto v]$ which associates to each $\ell$ a value $v$. We denote the set of locations where $S$ is defined by $dom(S)$.

In our semantics, configurations contain two stores. The first one is a collection of integer locations, and is called the *local* store. We use $\sigma$, $\sigma'$,... to represent its different states. The second one is a collection of process locations, and is called the *global* store. We use $\rho$, $\rho'$,... to represent its different states. We write $\rho(X)$ to denote the value of the process variable $X$ in $\rho$. Configurations are written $((p,\sigma),\rho)$, or $((pp,\sigma),\rho)$.

The local store keeps track of the values of data variables in the scope of the associated process. The global store keeps track of the values of process variables in the full space of configurations. Therefore, the state of the global store is only changed when a new process variable has been declared or if a process variable is assigned a new value.

Below we present the semantics of the new extensions in DEcCSP, the rest of DEcCSP is similar to EcCSP.

***General Dynamic recovery*** can be implemented in the calculus by using a *compensation pair with process variable*. A compensation pair with process variable consists of a standard process as a forward behaviour and a process variable as its compensation partner. The variable works as a place holder within the recovery sequence, where the real content can be retrieved later.

A compensation pair with process variable will be denoted by *px*, to distinguish it from the standard one denoted by *pp*, that is, $px = p \div X$, where $p$ is a standard process representing *px*'s forward behaviour, and $X$ is a process variable which works as place holder for *px*'s compensation behaviour. The variable $X$ should be fresh, i.e., not in the domain of the global store.

During the execution of the forward behaviour $p$, $X$'s value can be changed anywhere in the system. If $p$ terminates normally then the variable $X$ will be recorded, and $X$ still can be changed anywhere in the system as long as the compensation sequence has not been activated. If $p$ terminates abnormally then so does the compensation pair, resulting in an empty compensation. This is formalised by the following rules: $\dfrac{((p,\sigma),\rho) \xrightarrow{a} ((p',\sigma'),\rho)}{((p \div X,\sigma),\rho) \xrightarrow{a} ((p' \div X,\sigma'),\rho)}$, $\dfrac{((p,\sigma),\rho) \xrightarrow{\surd} ((\text{STOP},\sigma),\rho)}{((p \div X,\sigma),\rho) \xrightarrow{\surd} ((X,\sigma),\rho)}$, and $\dfrac{((p,\sigma),\rho) \xrightarrow{\omega} ((\text{STOP},\sigma),\rho)}{((p \div q,\sigma),\rho) \xrightarrow{\omega} ((\text{SKIP},\sigma),\rho)}$ for $\omega \in \{!,?\}$. The value of $X$ can be replaced by assigning a new value to it; to discard the compensation, we assign SKIP: $\dfrac{}{((X := p,\sigma),\rho) \xrightarrow{\tau} ((\text{SKIP},\sigma),\rho[X \mapsto p])}$.

The stored value of the process variable $X$ will be retrieved if the associated transaction throws an exception. If a transaction $[((pp,\sigma),\rho)]$ throws an exception !, then the transaction block will be ended, and the corresponding compensation $p$ will be activated. Therefore, the values of every process variable should be retrieved by replacing it with its value in the global store. If $\rho(X) = p$ then

$\dfrac{}{((X,\sigma),\rho) \xrightarrow{\tau} ((p,\sigma),\rho)}$

***Control Blocks.*** *If-then-else* and *while-do* are the same as the standard control blocks, where $b$ in the two control blocks is a Boolean expression which is evaluated according to the standard Boolean

semantics. The following three processes are defining the *Condition Block* for standard processes: $\frac{((b,\sigma),\rho) \xrightarrow{\tau} ((b',\sigma'),\rho)}{((\text{If } b \text{ Then } p \text{ Else } q,\sigma),\rho) \xrightarrow{\tau} ((\text{If } b' \text{ Then } p \text{ Else } q,\sigma'),\rho)}$, $\frac{}{((\text{If True Then } p \text{ Else } q,\sigma),\rho) \xrightarrow{\tau} ((p,\sigma),\rho)}$, and $\frac{}{((\text{If False Then } p \text{ Else } q,\sigma),\rho) \xrightarrow{\tau} ((q,\sigma),\rho)}$. The following three processes are defining the *Condition Block* for compensable processes: $\frac{((b,\sigma),\rho) \xrightarrow{\tau} ((b',\sigma'),\rho)}{((\text{If } b \text{ Then } pp \text{ Else } qq,\sigma),\rho) \xrightarrow{\tau} ((\text{If } b' \text{ Then } pp \text{ Else } qq,\sigma'),\rho)}$, $\frac{}{((\text{If True Then } pp \text{ Else } qq,\sigma),\rho) \xrightarrow{\tau} ((pp,\sigma),\rho)}$, and $\frac{}{((\text{If False Then } pp \text{ Else } qq,\sigma),\rho) \xrightarrow{\tau} ((qq,\sigma),\rho)}$.

Finally, *Iteration Block* in standard and compensable processes can be defined as follows:

$$\frac{}{((\text{While } b \text{ Do } p,\sigma),\rho) \xrightarrow{\tau} ((\text{If } b \text{ Then } (p;\text{While } b \text{ Do } p) \text{ Else SKIP},\sigma),\rho)} \text{ and}$$

$$\frac{}{((\text{While } b \text{ Do } pp,\sigma),\rho) \xrightarrow{\tau} ((\text{If } b \text{ Then } (pp;\text{While } b \text{ Do } pp) \text{ Else SKIP},\sigma),\rho)}.$$

***Named Processes for Definitions and Recursion.*** We write $(N = p)$ if $N$ is the name of the standard process $p$, and $(NN = pp)$ if $NN$ is the name of the compensable process $pp$. Process names can be used in the more common style of recursion where the process name is used in the process body. This can be defined as following: If $N = p$ then $\frac{}{((N,\sigma),\rho) \xrightarrow{\tau} ((p,\sigma),\rho)}$, and if $NN = pp$ then $\frac{}{((NN,\sigma),\rho) \xrightarrow{\tau} ((pp,\sigma),\rho)}$.

***Prefixing.*** Let $a$ be an event $\in \Sigma \cap \alpha p$ ($\alpha P$ is the set of events that the process can perform), and let $p$ be a process. Prefixing represents a standard process which is ready to engage in event $a$ and then behave as $p$, $\frac{}{(((a \longrightarrow p,\sigma),\rho) \xrightarrow{a} ((p,\sigma),\rho))}$.

The prefix operator can be used to link critical events, which should be executed in sequence without interruption. Prefixing differs from sequencing, as the following example shows.

According to DEcCSP's syntax ($P = a \longrightarrow b \longrightarrow \text{SKIP}$) and ($Q = a;b;\text{SKIP}$) are valid processes. However, $Q$ can be interrupted whereas $P$ can not. To be more clear the two process will be composed in parallel with THROW as follows: $P \underset{\phi}{\|} Q \underset{\phi}{\|} \text{THROW}$

If THROW has been performed before $P$ or $Q$, then the two processes can be interrupted. If THROW has happened after, e.g., the first event ($a$), then $Q$ will be prohibited from continuing execution while $P$ will not. This is due to the successful terminal event after $a$ in $Q$. This terminal event will synchronise with the terminal event ! in THROW resulting in ! which terminates the parallel operator. Such terminal event does not exist in $P$ therefore the process will continue its execution.

## 5 Case study

To illustrate the new features of DEcCSP, in this section we develop a case study based on the one described in [5]. We first demonstrate the basic extensions to cCSP, namely channels passing integers and control blocks. Then, the case study is extended to emphasise the general dynamic recovery.

Customers can order products from a warehouse. A customer should provide an item number, the quantity and his membership number (0 will be sent if the customer is not a member). If the order is accepted then proceed to fulfill this order and subtract the quantity from the inventory. If this action completed but later the transaction fails then the deducted quantity should be returned to the inventory. The `FulfillOrder` process starts by booking a courier which should be compensated if the transaction fails by cancelling the courier. If the customer is a member of this warehouse then no fee is charged, otherwise fees should be paid as part of the compensation process for booking the courier. `BookCourier` runs in parallel with packing the order, where each item ordered is packed, if there is an error then the item should be unpacked. At the same time, the customer's credit card is charged with the amount needed. This is done at the same time because it usually succeeds. However, if the credit process fails then the whole transaction fails and the system should be compensated.

This ordering system runs in parallel with the `Customer` process which issues an order or applies for a membership, and a `Bank` process which validates the credit card. In the following, we use $\|$ with no parameters to mean that the participant processes synchronise on the terminal events solely, and $\|_{i=0}^{i=y}$ is an indexed version of the parallel operator between several processes.

```
System = ((OrderTransaction ||B Bank ) ||A (Customer ||C CustomerService))
Where A={Order.x.y.ns }, B={CreditCheck.N, Ok, NotOk} and
C={RequestMembership, MembershipNumber.ns }
OrderTransaction = [ProcessOrder]
ProcessOrder = ((Order?x?y?ns; deduct.x.y) ÷ Restock.x.y) ;
FulfillOrder
FulfillOrder = BookCourier ÷ (If ns=0 Then cancelcourier1 Else
cancelcourier) || (||i=0i=y (Pack.x ÷ Unpack.x)) || ((CreditCheck!N ;
(Ok □ (NotOk ; THROW)))÷ SKIP)
cancelcourier1 = cancelcourier ; penalty
Customer= (Order!x!y!ns ; Customer) ⊓ ((RequestMembership; payfee;
MembershipNumber?ns) ; Customer)
CustomerService=RequestMembership;createprofile;MembershipNumber!ns;
CustomerService
Bank = CreditCheck?N ; (Ok ; Bank) ⊓ (NotOk ; Bank)
```

The system will be started by the customer process either executing `Order!x!y!ns`, where *x*, *y* and *ns* are integers, or `RequestMembership`. `Order!x!y!ns` starts a new transaction to process the order, that is, deduct the quantity from the inventory then proceed to `FulfillOrder`, which consists of three parallel subprocesses. Due to the parallel operator in this process its execution may have different possibilities. Assume the following scenario: a courier has been booked then three items of the product have been packed before the system checks the credit card. While the system is waiting for the bank to return the answer, a fourth item has been packed.

This scenario will lead us to the choice: `(Ok □ (NotOk ; THROW))`. If the bank answered `Ok` then the transaction continues, however, if the bank answered `NotOk` the current transaction terminates abnormally causing the associated compensations to start. The current compensations are:

```
Unpack.x || Unpack.x || Unpack.x || Unpack.x || (If ns=0 Then
cancelcourier1 Else cancelcourier); restock.x.y.
```

Alternatively, applying for a membership will be started by `RequestMembership` which is followed by a series of actions to process the application and is finished by the event `MembershipNumber.ns` which will send the membership number for the customer who issued `RequestMembership`.

The If statement provides the designer with the ability to delay resolving the choice of which compensation to start until compensation evaluation. However, the If statement is not sufficient for all cases, e.g, consider the following case.

This system assumes that items are always available in the warehouse. If we extend this system by removing this assumption, then items may not be available in the inventory. In this case, the warehouse should order the unavailable items from its two branches starting by the first one because it is nearer. If both branches fail to satisfy the order then the whole transaction fails.

To design this we add to the system a `PrepareOrder` process which checks if the item is available or not, if not then it orders the item from the two branches in sequence.

We now replace the compensation in `ProcessOrder` with a variable $X$, because the compensation is not known at the start; it depends on the `PrepareOrder` process.

```
OrderTransaction = [ProcessOrder ||_D (PrepareOrder ||_F (Branch1 ||_∅
Branch2))]
Where
D={Inventory.x.y, InvOK} and F={okbranch1,okbranch2,nobranch1,nobranch2}
ProcessOrder = ((Order?x?y?ns; X:= SKIP; Inventory!x!y; InvOK;
deduct.x.y) ÷ X); FulfillOrder
PrepareOrder= (Inventory?x?y; ((Available; X:=restock.x.y; InvOK) □
(NotAvailable; branch1!x!y; ((okbranch1;X:=(restock.x.y;Cancelbranch1);
InvOK) □ (nobranch1; branch2!x!y; ((okbranch2; X:=(restock.x.y;
Cancelbranch2);InvOK) □ (nobranch2; THROW))))) ÷ SKIP
Branch1= (branch1?x?y; (okbranch1 ⊓ nobranch1); Branch1) ÷ SKIP
Branch2= (branch2?x?y; (okbranch2 ⊓ nobranch2); Branch2) ÷ SKIP
```

The compensation sequence of the above process is:

```
Unpack.x ‖ Unpack.x ‖ Unpack.x ‖ Unpack.x ‖ (If ns=0 Then
cancelcourier1
Else cancelcourier); X.
```

At the start, $X$ is initialised with SKIP. The event `InvOK` is used to ensure that the deduction will not happen unless the order has been fulfilled.

## 6 Conclusions and future work

General dynamic recovery allows compensations to be replaced or discarded in the compensation sequence. This is useful in cases where compensations are not known from the beginning or if they are subject to change while the system is running. DEcCSP is a compensating calculus developed as an extension of EcCSP, improving the recovery mechanism (from backward recovery to general dynamic recovery) and including all of the CSP standard operators.

We have developed an operational semantics for DEcCSP. A denotational semantics for the three behavioural models of the standard CSP, as well as a theory of refinement, is left for future work. We also plan to extend the functionality of DEcCSP by introducing mobility.

## References


[1] Roberto Bruni, Michael J. Butler, Carla Ferreira, C. A. R. Hoare, Hernán C. Melgratti & Ugo Montanari (2005): *Comparing Two Approaches to Compensable Flow Composition*. In Martín Abadi & Luca de Alfaro, editors: *CONCUR 2005, Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, *Lecture Notes in Computer Science* 3653, Springer, pp. 383–397, doi:10.1007/11539452_30.

[2] Roberto Bruni, Anne Kersten, Ivan Lanese & Giorgio Spagnolo (2012): *A New Strategy for Distributed Compensations with Interruption in Long-Running Transactions*. In Till Mossakowski & Hans-Jörg Kreowski, editors: *Recent Trends in Algebraic Development Techniques, 20th International Workshop, WADT 2010, Etelsen, Germany, July 1-4, 2010, Revised Selected Papers*, *Lecture Notes in Computer Science* 7137, Springer, pp. 42–60, doi:10.1007/978-3-642-28412-0_5.

[3] Roberto Bruni, Hernán C. Melgratti & Ugo Montanari (2005): *Theoretical foundations for compensations in flow composition languages*. In Jens Palsberg & Martín Abadi, editors: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, ACM, pp. 209–220, doi:10.1145/1040305.1040323.

[4] Michael J. Butler, C. A. R. Hoare & Carla Ferreira (2005): *A Trace Semantics for Long-Running Transactions*. In Ali E. Abdallah, Cliff B. Jones & Jeff W. Sanders, editors: *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, *Lecture Notes in Computer Science* 3525, Springer, pp. 133–150, doi:10.1007/11423348_8.

[5] Michael J. Butler & Shamim Ripon (2005): *Executable Semantics for Compensating CSP*. In Mario Bravetti, Leïla Kloul & Gianluigi Zavattaro, editors: *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings*, *Lecture Notes in Computer Science* 3670, Springer, pp. 243–256, doi:10.1007/11549970_18.

[6] Zhenbang Chen & Zhiming Liu (2010): *An Extended cCSP with Stable Failures Semantics*. In Ana Cavalcanti, David Déharbe, Marie-Claude Gaudel & Jim Woodcock, editors: *Theoretical Aspects of Computing, ICTAC 2010, 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010. Proceedings*, *Lecture Notes in Computer Science* 6255, Springer, pp. 121–136, doi:10.1007/978-3-642-14808-8_9.

[7] Zhenbang Chen, Zhiming Liu & Ji Wang (2011): *Failure-Divergence Refinement of Compensating Communicating Processes*. In Michael Butler & Wolfram Schulte, editors: *FM 2011: Formal Methods, 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, *Lecture Notes in Computer Science* 6664, Springer, pp. 262–277, doi:10.1007/978-3-642-21437-0_21.

[8] Jim Gray & Andreas Reuter (1993): *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[9] Claudio Guidi, Ivan Lanese, Fabrizio Montesi & Gianluigi Zavattaro (2008): *On the interplay between fault handling and request-response service invocations*. In Jonathan Billington, Zhenhua Duan & Maciej Koutny, editors: *8th International Conference on Application of Concurrency to System Design (ACSD 2008), Xi'an, China, June 23-27, 2008*, IEEE, pp. 190–198, doi:10.1109/ACSD.2008.4574611.

[10] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi & Gianluigi Zavattaro (2006): *SOCK: A Calculus for Service Oriented Computing*. In Asit Dan & Winfried Lamersdorf, editors: *Service-Oriented Computing, ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, *Lecture Notes in Computer Science* 4294, Springer, pp. 327–338, doi:10.1007/11948148_27.

[11] C. A. R. Hoare (1985): *Communicating Sequential Processes*. Prentice-Hall.

[12] Ivan Lanese, Cátia Vaz & Carla Ferreira (2010): *On the Expressive Power of Primitives for Compensation Handling*. In Andrew D. Gordon, editor: *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice*

*of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, *Lecture Notes in Computer Science* 6012, Springer, pp. 366–386, doi:10.1007/978-3-642-11957-6_20.

[13] Ivan Lanese, Cátia Vaz & Carla Ferreira (2011): *On the Expressive Power of Primitives for Compensation Handling*. Journal version of [12], submitted.

[14] Cosimo Laneve & Gianluigi Zavattaro (2005): *Foundations of Web Transactions*. In Vladimiro Sassone, editor: *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005*, *Lecture Notes in Computer Science* 3441, Springer, pp. 282–298, doi:10.1007/978-3-540-31982-5_18.

[15] Shamim Ripon (2008): *Extending and Relating Semantic Models of Compensating CSP*. Ph.D. thesis, University of Southampton. Available at http://eprints.soton.ac.uk/266584/.

[16] A. W. Roscoe (1998): *The theory and practice of concurrency*. Prentice Hall. Available at http://www.cs.ox.ac.uk/people/bill.roscoe/publications/68b.pdf.

[17] A.W. Roscoe (2010): *Understanding Concurrent Systems*. Texts in Computer Science, Springer, doi:10.1007/978-1-84882-258-0.

[18] Cátia Vaz, Carla Ferreira & António Ravara (2009): *Dynamic Recovering of Long Running Transactions*. In Christos Kaklamanis & Flemming Nielson, editors: *Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, *Lecture Notes in Computer Science* 5474, Springer, pp. 201–215, doi:10.1007/978-3-642-00945-7_13.

[19] Edsko de Vries, Vasileios Koutavas & Matthew Hennessy (2010): *Communicating Transactions – (Extended Abstract)*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010, Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, *Lecture Notes in Computer Science* 6269, Springer, pp. 569–583, doi:10.1007/978-3-642-15375-4_39.