An Extended CSP for Service-Oriented Architectures

Abeer S. Al-Humaimeedy[1,2] and Maribel Fernández[1]

[1] King's College London, UK
[2] King Saud University, Riyadh, Saudi Arabia
Abeer@ksu.edu.sa, Maribel.Fernandez@kcl.ac.uk

Abstract. The Service Oriented Architecture (SOA) paradigm is based on service composition, that is, loosely coupled autonomous heterogeneous services, which are collectively composed to implement a particular task. This paper presents a new calculus for SOA within the framework of CSP process algebra, showing how services can be defined, invoked, orchestrated and terminated within session hierarchies. Additionally, the new proposal, called SoaCSP, introduces a new method to achieve pi-calculus style mobility within CSP. We provide a formal, operational semantics for the new calculus, and illustrate its expressiveness with examples.

Keywords: SOA Calculi, Compensating CSP, Mobility, Sessions

1 Introduction

Software architecture paradigms define the fundamental features of a software system and govern the subsequent design activities. The Service Oriented Architecture (SOA) paradigm, based on service composition, has been successfully applied to facilitate systems integration. Service composition refers to an aggregate of loosely coupled autonomous heterogeneous services, which are collectively composed to implement a particular task. Services can communicate using messages solely, and can change as the composition evolves.

The dominant modelling methods for SOA systems are a series of internet standards (*WS**-) [31,39,37]. Although widely used, these standards raise two fundamental issues. They lack a formal semantics, and there are many interdependencies, so internet standards fall short when applied independently. To address these issues, many process calculi, mostly inspired by the π-calculus [30] (e.g., [2,22,3]), or its extensions (e.g., [17,24,27]), have been developed as a formal modelling languages for specifying and analysing SOA systems.

Although these calculi in general host similar mechanisms for service definition and invocation, they are based on different mechanisms to encode service interactions. Explicit primitives, such as sessions, streams, orchestrator, and pipelines have been used in [25,2,22,3], to dynamically structure the conversations between services. Other calculi [36,16,26] adapt WS-PBEL[31] correlation mechanism, where conversations are tagged with variables to identify that they

belong to one session. Other proposals [7,24,27] introduce transaction mechanisms into general process calculi to cope with failure behaviours in SOA systems. Critical issues in SOA environments, such as termination and multi party sessions, have been discussed in [3,5].

The calculi mentioned above have significant differences in their expressive power. These differences will be highlighted in Section 7.

Herewith, we propose a new calculus for SOA, focusing on the expressiveness of the communication model, safe termination for services, preservation of service autonomy, and availability of formal reasoning and verification techniques.

Our new calculus is based on the well-known process calculus CSP [18], for which many useful verification techniques are already available, including mechanisms to identify good/bad traces, deadlock freedom and divergence freedom [19]. CSP has many features that make it a good starting point to develop SOA calculi. For instance, it distinguishes between deterministic and non-deterministic behaviour of systems and it is therefore possible to reason directly on these properties; it permits explicit synchronisation of terminal events, which facilitates the specification of termination and interruption of services; finally, its multi-way synchronisation feature can be used to model the mixed and complicated communication patterns of SOA.

To the best of our knowledge, our proposal is the first extension of CSP with SOA primitives. However, although less popular for SOA, CSP has already been used in this context, in particular to verify the correctness of the internet standards [41,21], to model agent communication [28,29] and to model transactions [8].

CSP with transactions primitives [8], Compensating CSP (cCSP), introduces an expressive termination mechanism to the flexible communication mechanism of the original CSP. However, its lack of mobility and sessioning reduces the usage of this calculus as SOA modelling language. Therefore, we build on cCSP and endow it with mobility and services primitives to obtain adequately expressive calculus for SOA. Compensating CSP [8,13] includes transaction-handling primitives inspired by the well-developed transaction techniques from database theory. Compensations are evaluated in case of system failure, when processes need to roll-back.

In previous work [20], we extended the recovery mechanisms in Compensating CSP to facilitate general dynamic recovery, where compensations can be replaced or discarded at runtime. In this paper, we present SoaCSP as a mobile, compensating process calculus for SOA based on CSP. We extend our improved version of Compensating CSP with primitives to model mobility, service definition, service invocation and service communication. We introduce these primitives in two steps: informally by describing the purpose of these primitives, then formally by developing an operational semantics using labelled transitions systems.

We propose a novel algorithm to organise processes in sessions. We discuss the expressiveness of our algorithm to deal with nested sessions, multi-party sessions, communication delegation, and graceful termination. Additionally, we

propose a novel algorithm to introduce π-calculus style mobility in CSP. In both algorithms we insist on service autonomy.

We design our calculus in a cumulative way, that is, adding the new SOA features without compromising the CSP original features. In other words, retain the original syntax and theory of CSP, or in particular cCSP, and extend it with the new primitives, then build the theory in a consistent way with the old theory. Therefore, the resulting calculus has a rich syntax, providing designers with more tools. Much of our work is in introducing new types of events and new dynamic algorithms to implement the new features needed.

Overview of the paper: For the sake of clarity we present SoaCSP in two steps. First, in Section 3 we present SOA primitives only without mobility along with its operational semantics. In Section 4 we introduce our mobility model for SoaCSP along with its syntax and operational semantics. Section 2 recalls briefly the syntax and operational semantics of the base calculus *Compensating CSP*. Section 5 discusses the calculus properties. Related work and future work avenues are exposed in Section 7 and Section 8 respectively. Section 6 illustrates the calculus expressive power with an example.

2 Preliminaries

In this section, we recall the main concepts in Compensating CSP (cCSP) that are necessary to understand the concepts and the examples in this paper, assuming the reader is familiar with CSP [32,34]. Interested readers can refer to [8,13,20] for details of compensating CSP.

Compensating CSP (cCSP) [7] introduced the new type of processes *compensable processes* into the original CSP. Compensable processes are processes which comprise two behaviours: the normal behaviour and the reverse behaviour which undo the effects of the normal execution in the case of a failure.

Herewith we recall the main concepts in Compensating CSP (cCSP) (cCSP) [8] and its extensions [13,14], assuming the reader is familiar with CSP [32,34].

cCSP categorises processes into 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 $STOP$, 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.

EcCSP [13,14] is an extension of cCSP for which a theory of refinement has been developed.

Compensation in cCSP and EcCSP is limited to backward recovery. DEcCSP (Dynamic EcCSP) [20] extends EcCSP further by introducing primitives to facilitate general dynamic recovery, which allows compensations to be replaced or discarded after they have been recorded. This is useful in many cases, for instance, when the compensation process is unknown at the start, the compensation process is subject to change while the process evolves, or the compensation logic

is complex and spans several processes. Additionally, DEcCSP extends EcCSP by including all the standard CSP operators, to facilitate the specification of complex systems.

In this paper, we build on DEcCSP to obtain a mobile, compensating calculus for SOA modelling. We recall the syntax and operational semantics of DEcCSP in Figures 1,2, and 3. The following notations are used. Standard processes are denoted by lower case letters $p, q, \ldots$. Compensable processes are written $p \div q$ and referred to using double letters $pp, qq, \ldots$. The set of events that the standard process p can perform is denoted by αp; αpp is the set of events that the compensable process pp can perform: if $pp = p \div q$, then $\alpha pp = \alpha p \cup \alpha q$. Σ is the universal set that contains all the observable events in a system ($\Sigma = \bigcup_{i=0}^{i=n-1} \alpha p_i$ if the system contains n standard processes); $a, b, c \ldots$ will be used to range over this set. Ω is the set of terminal events $\{!, ?, \surd\}$; ω will be used to range over this set. Σ^τ is the universal set Σ and in addition the silent event τ. $\Sigma^{\tau\Omega}$ is the universal set Σ^τ union Ω. Capital letters $A, B, \ldots$ denote sets of observable events; b, e denote Boolean and integer expressions, respectively; $\ell, x, y, \ldots$ denote integer variables and $X, Y, Z, \ldots$ denote process variables.

(Standard Processes)		(Compensable Processes)	
$p, q ::= \quad a$	(Atomic process)	$pp, qq ::= \quad p \div q$	(Compensation pair (CP))
$\mid \quad p \Box q$	(External choice)	$\mid \quad pp \Box qq$	(External choice)
$\mid \quad p; q$	(Sequential operator)	$\mid \quad pp; qq$	(Sequential operator)
$\mid \quad SKIP$	(Primitive process)	$\mid \quad SKIPP$	(Primitive process)
$\mid \quad THROW$	(Primitive process)	$\mid \quad THROWW$	(Primitive process)
$\mid \quad YIELD$	(Primitive process)	$\mid \quad YIELDD$	(Primitive process)
$\mid \quad p \sqcap q$	(Internal choice)	$\mid \quad pp \sqcap qq$	(Internal choice)
$\mid \quad p \underset{A}{\parallel} q$	(Parallel operator)	$\mid \quad pp \underset{A}{\parallel} qq$	(Parallel operator)
$\mid \quad p \backslash A$	(Hiding operator)	$\mid \quad pp \backslash A$	(Hiding operator)
$\mid \quad p[\![R]\!]$	(Renaming operator)	$\mid \quad pp[\![R]\!]$	(Renaming operator)
$\mid$ If b Then p Else q	(conditional block)	$\mid$ If b Then pp Else qq	(conditional block)
$\mid$ while b do p	(while block)	$\mid$ while b do pp	(while block)
$\mid \quad N$	(Process name)	$\mid \quad NN$	(Process name)
$\mid \quad a \longrightarrow p$	(Prefix operator)	$\mid \quad p \div X$	(Variable CP)
$\mid \quad X := p$	(Variable assignment)	$\mid \quad pp \boxtimes qq$	(Speculative choice)
$\mid \quad p \triangleright q$	(Interrupt handler)		
$\mid \quad STOP$	(Primitive process)		
$\mid \quad [pp]$	(Transaction block)		

Network of n parallel processes:
If all the n processes have the same A: $\underset{A}{\parallel_{i=0}^{n-1}} p_i \qquad \underset{A}{\parallel_{i=0}^{n-1}} pp_i$
If they have different sets: $((((p_0 \underset{A_1}{\parallel} p_1) \underset{A_2}{\parallel} p_2) \ldots) \underset{A_{n-1}}{\parallel} p_{n-1}) \qquad ((((pp_0 \underset{A_1}{\parallel} pp_1) \underset{A_2}{\parallel} pp_2) \ldots) \underset{A_{n-1}}{\parallel} pp_{n-1})$
Process definitions: $(N = p) \mid (NN = pp)$

Fig. 1. DEcCSP syntax

The sequential composition, parallel composition, external (deterministic) choice, internal (non-deterministic) choice, STOP, hiding, and renaming are standard CSP operators. We describe below the new and modified constructs.

The operator [ ] is used to identify transaction boundaries (transactions can be nested, and subtransactions should be aborted if the parent transaction is

terminated). To represent unsuccessful termination of a process, new terminal signals (events) have been added to CSP: (!) represents internal fault; (?) represents yielding to external fault. *THROW* is a primitive process that throws an exception then terminates. *YIELD* is a process that yields to an external exception and terminates. The operator $\triangleright$ has been added to represent fault handlers: in $p \triangleright q$, q is the fault handler of p. Compensable processes can be defined in the calculus as a pair of standard processes composed with the new operator $\div$. In $p \div q$, q is the compensation handler of p.

Speculative choice, where two processes run in parallel without synchronisation, is resolved when one of them commits, then the other should immediately compensate. If both of them fail then the whole choice will fail and the processes should compensate in parallel.

The *Prefix operator* is a sequential operator to link critical events, which should be executed in sequence without interruption.

General dynamic recovery is implemented in the calculus by using a *compensation pair with process variable*, which 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. Process variables can be assigned new values anywhere in the system using assignment (:=).

If-Then-Else and *while-do* are similar to the standard control blocks.

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 recursive definitions (where the process name is used in the process body).

Events in DEcCSP, like in the standard CSP, can be classified as **ordinary**, which are either a single literal name describing an action to be performed, or a compound name built by composing multiple literal names (e.g., $a.b$); and **channels**, which are ordinary events composed with integer expressions to denote communicated data. For example, if a is a channel name, we write $a.3$ to indicate that the number 3 is transmitted through the channel a, without indicating the direction. The composition operator . can be replaced by ! or ? to indicate the direction of communication: $a?\ell$ denotes input on the variable ℓ, and $a!e$ denotes output of the value of the integer expression e. Channels can carry simultaneously multiple values in both directions (e.g., if c is a channel name then $c?x!y!z$ is a valid event).

The operational semantics of DEcCSP's processes without channels (i.e., considering only ordinary events) is given in [20] as a transition relation between configurations containing a global store to keep track of the values of process variables. The global store is denoted by ρ, where ρ is a collection of process locations. Formally, it is a function $\rho[X \mapsto p]$ which associates to each X a value p; $dom(\rho)$ denotes the set of locations where ρ is defined. Configurations are written as (p, ρ) or (pp, ρ). The state of the global store is only changed when a new process variable is declared or if a process variable is assigned a new value; we use ρ', ρ'',... to represent its different states. The b Boolean expressions is evaluated as the standard Boolean semantics.

A formal semantics for channel events will be given in Section 4, where mobility will be discussed. In the rest of the paper, Compensating CSP will mean DEcCSP rather than Butler *et al*'s cCSP.

(skip) $\frac{}{(SKIP,\rho)\xrightarrow{\checkmark}(STOP,\rho)}$ (throw) $\frac{}{(THROW,\rho)\xrightarrow{!}(STOP,\rho)}$

(yield) $\frac{}{(YIELD,\rho)\xrightarrow{\omega}(STOP,\rho)}$ $(\omega \in \{?,\checkmark\})$

(atomic) $\frac{}{(a,\rho)\xrightarrow{a}(SKIP,\rho)}$ (event-intr-b) $\frac{}{(a,\rho)\xrightarrow{?}(STOP,\rho)}$

(event-intr-a) $\frac{}{(a,\rho)\xrightarrow{a}(STOP,\rho)}$ $(a \in \Sigma)$

(Echc1,2) $\frac{(p,\rho)\xrightarrow{a}(p',\rho)}{(p\,\Box\,q,\rho)\xrightarrow{a}(p',\rho)}$ $\frac{(q,\rho)\xrightarrow{b}(q',\rho)}{(p\,\Box\,q,\rho)\xrightarrow{b}(q',\rho)}$ $(a,b \in \Sigma^{\omega})$

(Echc3,4) $\frac{(p,\rho)\xrightarrow{\tau}(p',\rho)}{(p\,\Box\,q,\rho)\xrightarrow{\tau}(p'\,\Box\,q,\rho)}$ $\frac{(q,\rho)\xrightarrow{\tau}(q',\rho)}{(p\,\Box\,q,\rho)\xrightarrow{\tau}(p\,\Box\,q',\rho)}$

(seq1) $\frac{(p,\rho)\xrightarrow{a}(p',\rho)}{(p\,;q,\rho)\xrightarrow{a}(p'\,;q,\rho)}$ $(a \in \Sigma^{\tau})$ (seq2) $\frac{(p,\rho)\xrightarrow{\checkmark}(p',\rho)}{(p\,;q,\rho)\xrightarrow{\tau}(q,\rho)}$

(seq3) $\frac{(p,\rho)\xrightarrow{\omega}(STOP,\rho)}{(p\,;q,\rho)\xrightarrow{\omega}(STOP,\rho)}$ $(\omega \in \{!,?\})$ (Intr-hdr1) $\frac{(p,\rho)\xrightarrow{a}(p',\rho)}{(p\triangleright q,\rho)\xrightarrow{a}(p'\triangleright q,\rho)}$ $(a \in \Sigma^{\tau})$

(Intr-hdr2) $\frac{(p,\rho)\xrightarrow{!}(p',\rho)}{(p\triangleright q,\rho)\xrightarrow{\tau}(q,\rho)}$ (Intr-hdr3) $\frac{(p,\rho)\xrightarrow{\omega}(STOP,\rho)}{(p\triangleright q,\rho)\xrightarrow{\omega}(STOP,\rho)}$ $(\omega \in \{\checkmark,?\})$

(par1) $\frac{(p,\rho)\xrightarrow{b}(p',\rho)}{(p \underset{A}{\parallel} q,\rho)\xrightarrow{b}(p' \underset{A}{\parallel} q,\rho)}$ (par2) $\frac{(q,\rho)\xrightarrow{c}(q',\rho)}{(p \underset{A}{\parallel} q,\rho)\xrightarrow{c}(p \underset{A}{\parallel} q',\rho)}$ $(b,c \notin A)$

(par3) $\frac{(p,\rho)\xrightarrow{a}(p',\rho) \quad (q,\rho)\xrightarrow{a}(q',\rho)}{(p \underset{A}{\parallel} q,\rho)\xrightarrow{a}(p' \underset{A}{\parallel} q',\rho)}$ $(a \in A)$ (syn-table) $\begin{array}{c|cccccc} \overset{\omega}{\omega'} & ! & ! & ! & ? & ? & \checkmark \\ \hline ! & ? & \checkmark & ? & \checkmark & \checkmark \\ ? & ! & ! & ! & ? & ? & \checkmark \end{array}$ $\omega\&\omega'$

(par4) $\frac{(p,\rho)\xrightarrow{\omega}(STOP,\rho) \quad (q,\rho)\xrightarrow{\omega'}(STOP,\rho)}{(p \underset{A}{\parallel} q,\rho)\xrightarrow{\omega\&\omega'}(STOP,\rho)}$ $(\omega,\omega' \in \Omega \wedge \omega\&\omega' \in (syn-table))$

(trans-blk1) $\frac{(pp,\rho)\xrightarrow{a}(pp',\rho)}{([pp],\rho)\xrightarrow{a}([pp'],\rho)}$ $(a \in \Sigma^{\tau})$ (trans-blk2) $\frac{(pp,\rho)\xrightarrow{!}(p,\rho)}{([pp],\rho)\xrightarrow{!}(p,\rho)}$

(trans-blk3) $\frac{(pp,\rho)\xrightarrow{\checkmark}(p,\rho)}{([pp],\rho)\xrightarrow{\checkmark}(STOP,\rho)}$ (trans-blk4) $\frac{(pp,\rho)\xrightarrow{?}(p,\rho)}{([pp],\rho)\xrightarrow{?}(p,\rho)}$

(Ich1,2) $\frac{}{(p\sqcap q,\rho)\xrightarrow{\tau}(p,\rho)}$ $\frac{}{(p\sqcap q,\rho)\xrightarrow{\tau}(q,\rho)}$

(hd1) $\frac{(p,\rho)\xrightarrow{b}(p',\rho)}{(p\setminus A,\rho)\xrightarrow{b}(p'\setminus A,\rho)}$ $(b \notin A)$ (hd2) $\frac{(p,\rho)\xrightarrow{a}(p',\rho)}{(p\setminus A,\rho)\xrightarrow{\tau}(p'\setminus A,\rho)}$ $(a \in A)$

(hd3) $\frac{(p,\rho)\xrightarrow{\omega}(STOP,\rho)}{(p\setminus A,\rho)\xrightarrow{\omega}(STOP,\rho)}$ $(\omega \in \Omega)$ (rnm1) $\frac{(p,\rho)\xrightarrow{a}(p',\rho)}{(p[\![R]\!],\rho)\xrightarrow{b}(p'[\![R]\!],\rho)}$ $(a\,R\,b)$

(rnm2) $\frac{(p,\rho)\xrightarrow{\tau}(p',\rho)}{(p[\![R]\!],\rho)\xrightarrow{\tau}(p'[\![R]\!],\rho)}$ (rnm3) $\frac{(p,\rho)\xrightarrow{\omega}(STOP,\rho)}{(p[\![R]\!],\rho)\xrightarrow{\omega}(STOP,\rho)}$ $(\omega \in \Omega)$

(var-assign) $\frac{}{(X:=p,\rho)\xrightarrow{\tau}(SKIP,\rho[X\mapsto p])}$

(if1) $\frac{(b,\rho)\xrightarrow{\tau}(b',\rho)}{(If\ b\ Then\ p\ Else\ q,\rho)\xrightarrow{\tau}(If\ b'\ Then\ p\ Else\ q,\rho)}$

(if2) $\frac{}{(If\ True\ Then\ p\ Else\ q,\rho)\xrightarrow{\tau}(p,\rho)}$ $\frac{}{(If\ False\ Then\ p\ Else\ q,\rho)\xrightarrow{\tau}(q,\rho)}$

(while) $\frac{}{(while\ b\ do\ p,\rho)\xrightarrow{\tau}(If\ b\ Then\ (p;while\ b\ do\ p)\ Else\ SKIP,\rho)}$

(name) $\frac{}{(N,\rho)\xrightarrow{\tau}(p,\rho)}$ (if $N = p$) (prefix) $\frac{}{((a\longrightarrow p,\rho)\xrightarrow{a}(p,\rho))}$

Fig. 2. DEcCSP operational semantics: standard processes

(cp1) $\frac{(p,\rho)\xrightarrow{a}(p',\rho)}{(p\div q,\rho)\xrightarrow{a}(p'\div q,\rho)}\ (a\in\Sigma^\tau)$ (cp2) $\frac{(p,\rho)\xrightarrow{\surd}(STOP,\rho)}{(p\div q,\rho)\xrightarrow{\surd}(q,\rho)}$

(cp3) $\frac{(p,\rho)\xrightarrow{\omega}(STOP,\rho)}{(p\div q,\rho)\xrightarrow{\omega}(SKIP,\rho)}\ (\omega\in\{!,?\})$ (var-cp1) $\frac{(p,\rho)\xrightarrow{a}(p',\rho)}{(p\div X,\rho)\xrightarrow{a}(p'\div X,\rho)}\ (a\in\Sigma^\tau)$

(var-cp2) $\frac{(p,\rho)\xrightarrow{\surd}(STOP,\rho)}{(p\div X,\rho)\xrightarrow{\surd}(\rho(X),\rho)}$ (var-cp3) $\frac{(p,\rho)\xrightarrow{\omega}(STOP,\rho)}{(p\div q,\rho)\xrightarrow{\omega}(SKIP,\rho)}\quad(\omega\in\{!,?\})$

(skipp) $\frac{}{(SKIPP,\rho)\xrightarrow{\surd}(SKIP,\rho)}\ where\ SKIPP=SKIP\div SKIP$

(throww) $\frac{}{(THROWW,\rho)\xrightarrow{!}(SKIP,\rho)}\ where\ THROWW=THROW\div SKIP$

(yieldd) $\frac{}{(YIELDD,\rho)\xrightarrow{\omega}(SKIP,\rho)}\ (\omega\in\{?,\surd\})\ where\ YIELDD=YIELD\div SKIP$

(Echc-c1,2) $\frac{(pp,\rho)\xrightarrow{a}(pp',\rho)}{(pp\,\Box\,qq,\rho)\xrightarrow{a}(pp',\rho)}\ \frac{(qq,\rho)\xrightarrow{b}(qq',\rho)}{(pp\,\Box\,qq,\rho)\xrightarrow{b}(qq',\rho)}\quad(a,b\in\Sigma)$

(Echc-c3,4) $\frac{(pp,\rho)\xrightarrow{\omega}(p,\rho)}{(pp\Box qq,\rho)\xrightarrow{\omega}(p,\rho)}\ \frac{(qq,\rho)\xrightarrow{\omega'}(q,\rho)}{(pp\Box qq,\rho)\xrightarrow{\omega'}(q,\rho)}\ (\omega,\omega'\in\Omega)$

(Echc-c5,6) $\frac{(pp,\rho)\xrightarrow{\tau}(pp',\rho)}{(pp\Box qq,\rho)\xrightarrow{\tau}(pp'\Box qq,\rho)}\ \frac{(qq,\rho)\xrightarrow{\tau}(qq',\rho)}{(pp\Box qq,\rho)\xrightarrow{\tau}(pp\Box qq',\rho)}$

(seq1-c) $\frac{(pp,\rho)\xrightarrow{a}(pp',\rho)}{(pp\,;qq,\rho)\xrightarrow{a}(pp'\,;qq,\rho)}\ (a\in\Sigma^\tau)$ (seq2-c) $\frac{(pp,\rho)\xrightarrow{\surd}(p,\rho)}{(pp\,;qq,\rho)\xrightarrow{\tau}(\langle qq,p\rangle,\rho)}$

(seq3-c) $\frac{(pp,\rho)\xrightarrow{\omega}(p,\rho)}{(pp\,;qq,\rho)\xrightarrow{\omega}(p,\rho)}\ (\omega\in\{!,?\})$ (seq-axl1) $\frac{(qq,\rho)\xrightarrow{a}(qq',\rho)}{(\langle qq,p\rangle,\rho)\xrightarrow{a}(\langle qq',p\rangle,\rho)}\ (a\in\Sigma^\tau)$

(seq-axl2) $\frac{(qq,\rho)\xrightarrow{\omega}(q,\rho)}{(\langle qq,p\rangle,\rho)\xrightarrow{\omega}(q\,;p,\rho)}\ (\omega\in\Omega)$ (par-c1,2) $\frac{(pp,\rho)\xrightarrow{b}(pp',\rho)}{(pp\underset{A}{\parallel}qq,\rho)\xrightarrow{b}(pp'\underset{A}{\parallel}qq,\rho)}$

$\frac{(qq,\rho)\xrightarrow{c}(qq',\rho)}{(pp\underset{A}{\parallel}qq,\rho)\xrightarrow{c}(pp\underset{A}{\parallel}qq',\rho)}\quad(b,c\notin A)$ (par-c3) $\frac{(pp,\rho)\xrightarrow{a}(pp',\rho)\quad(qq,\rho)\xrightarrow{a}(qq',\rho)}{(pp\underset{A}{\parallel}qq,\rho)\xrightarrow{a}(pp'\underset{A}{\parallel}qq',\rho)}\quad(a\in A)$

(par-c4) $\frac{(pp,\rho)\xrightarrow{\omega}(p,\rho)\quad(qq,\rho)\xrightarrow{\omega'}(q,\rho)}{(pp\underset{A}{\parallel}qq,\rho)\xrightarrow{\omega\&\omega'}(p\underset{A}{\parallel}q,\rho)}\ (\omega,\omega'\in\Omega)$ (name-c) $\frac{}{(NN,\rho)\xrightarrow{\tau}(pp,\rho)}$ (if $NN=pp$)

(Ichc-c1,2): $\frac{}{(pp\sqcap qq,\rho)\xrightarrow{\tau}(pp,\rho)}\quad\frac{}{(pp\sqcap qq,\rho)\xrightarrow{\tau}(qq,\rho)}$ (Hd-c1) $\frac{(pp,\rho)\xrightarrow{b}(pp',\rho)}{(pp\setminus A,\rho)\xrightarrow{b}(pp'\setminus A,\rho)}\ (b\notin A)$

(Hd-c2) $\frac{(pp,\rho)\xrightarrow{a}(pp',\rho)}{(pp\setminus A,\rho)\xrightarrow{\tau}(pp'\setminus A,\rho)}\ (a\in A)$ (Hd-c3) $\frac{(pp,\rho)\xrightarrow{\omega}(p,\rho)}{(pp\setminus A,\rho)\xrightarrow{\omega}(p\setminus A,\rho)}\ (\omega\in\Omega)$

(rnm-c1) $\frac{(pp,\rho)\xrightarrow{a}(pp',\rho)}{(pp[\![R]\!],\rho)\xrightarrow{b}(pp'[\![R]\!],\rho)}\ (a\,R\,b)$ (rnm-c2) $\frac{(pp,\rho)\xrightarrow{\tau}(pp',\rho)}{(pp[\![R]\!],\rho)\xrightarrow{\tau}(pp'[\![R]\!],\rho)}$

(rnm-c3) $\frac{(pp,\rho)\xrightarrow{\omega}(p,\rho)}{(p[\![R]\!],\rho)\xrightarrow{\omega}(p[\![R]\!],\rho)}\ (\omega\in\Omega)$ (spc1) $\frac{(pp,\rho)\xrightarrow{\surd}(p,\rho)\quad(qq,\rho)\xrightarrow{\surd}(q,\rho)}{(pp\boxtimes qq,\rho)\xrightarrow{\surd}(\langle q,p\rangle\,\Box\,\langle p,q\rangle,\rho)}$

(spc2,3) $\frac{(pp,\rho)\xrightarrow{a}(pp',\rho)}{(pp\boxtimes qq,\rho)\xrightarrow{a}(pp'\boxtimes qq,\rho)}\ \frac{(qq,\rho)\xrightarrow{b}(qq',\rho)}{(pp\boxtimes qq,\rho)\xrightarrow{b}(pp\boxtimes qq',\rho)}\ (a,b\in\Sigma)$

(spc4,5) $\frac{(pp,\rho)\xrightarrow{\surd}(p,\rho)\quad(qq,\rho)\xrightarrow{\omega}(q,\rho)}{(pp\boxtimes qq,\rho)\xrightarrow{\surd}(\langle q,p\rangle,\rho)}\ \frac{(qq,\rho)\xrightarrow{\surd}(q,\rho)\quad(pp,\rho)\xrightarrow{\omega}(p,\rho)}{(pp\boxtimes qq,\rho)\xrightarrow{\surd}(\langle p,q\rangle,\rho)}(\omega\in\{!,?\})$

(spc6) $\frac{(pp,\rho)\xrightarrow{\omega}(p,\rho)\quad(qq,\rho)\xrightarrow{\omega'}(q,\rho)}{(pp\boxtimes qq,\rho)\xrightarrow{\omega\&\omega'}(\langle(\omega\&\omega'),(p\parallel q)\rangle,\rho)}\quad(\omega,\omega'\in\{!,?\})$

(spc-axl1) $\frac{(p,\rho)\xrightarrow{a}(p',\rho)}{(\langle p,q\rangle,\rho)\xrightarrow{a}(\langle p',q\rangle,\rho)}(a\in\Sigma^\tau)$ (spc-axl2) $\frac{(p,\rho)\xrightarrow{\omega}(STOP,\rho)}{(\langle p,q\rangle,\rho)\xrightarrow{\omega}(q,\rho)}(\omega\in\Omega)$

(if-c1) $\frac{(b,\rho)\xrightarrow{\tau}(b',\rho)}{(If\ b\ Then\ pp\ Else\ qq,\rho)\xrightarrow{\tau}(If\ b'\ Then\ pp\ Else\ qq,\rho)}$

(if-c2,3) $\frac{}{(If\ True\ then\ pp\ Else\ qq,\rho)\xrightarrow{\tau}(pp,\rho)}\quad\frac{}{(If\ False\ Then\ pp\ Else\ qq,\rho)\xrightarrow{\tau}(qq,\rho)}$

(while-c) $\frac{}{(while\ b\ do\ pp,\rho)\xrightarrow{\tau}(If\ b\ Then\ (pp;while\ b\ do\ pp)\ Else\ SKIP,\rho)}$

Fig. 3. DEcCSP operational semantics: compensable processes

3 The Mobility-free Fragment of SoaCSP

In this section, we introduce the informal then the formal definitions of service invocation and sessioning, including a discussion on nesting of sessions and terminations; session expansion will be discussed after the mobility model is presented.

As mentioned in the introduction, existing SOA calculi in general host similar mechanisms for service definition and invocation; however they introduce different mechanisms to encode service interaction (sessions).

In general, we can divide the SOA calculi into calculi which are hosting static or dynamic primitives to identify sessions borders and participants, and calculi which are encoding mechanisms like correlation to identify conversations that belong to one session.

Although correlation mechanisms can be encoded using the existing mechanisms in CSP, we suggest introducing explicit primitives to define sessions for the following reasons: (i) having primitives to identify sessions facilitates direct analysis of this feature, whereas correlating algorithms complicate the analysis of sessions; (ii) sessions provide a scoping hierarchy for process activity, especially the termination processes.

In our version of CSP for SOA, we adopt Hoare Labelling operator[18] to specify sessions. Labels are used to attach session names to processes. Hoare Labelling operator is defined as follows:

Definition 1 (*Hoare's Labelling*). If p is a process then $l : p$ is the same process with label l. If a process is labelled then all its events will be labelled as well. Thus, if p is ready to engage in event a then $l : p$ will be ready to engage in event $l.a$.

Consequently, a system which has different instances of p participating in sessions $(l_1, l_2, \cdots)$ behaves as $(l_1 : p \parallel l_2 : p \parallel l_3 : p \cdots)$.

Recall that in CSP, $\|_\Sigma$, presents totally synchronous communication; $\|_\phi$ represents totally asynchronous communication; and $\|_A$ represents mixed communication where processes synchronise on A's events solely.

In standard CSP, $\|$ is equivalent to $\|_\Sigma$ due to the fact that synchronous communications are the dominant communications in concurrent systems. However, in SoaCSP, $\|$ is equivalent to $\|_\phi$, due to that fact that asynchronous communications are the dominant communications in SOA systems.

Service Definition, Invocation, and Sessioning. In SoaCSP, if a service is defined or invoked then the service will be labelled with a fresh name. The labelled service is considered as a new instance of that service, and the label serves as a session name.

To define an invocable service we write $(N \Rightarrow p)$, where N is the name of the service and p is its body. This definition allows a service to be invoked only once, then it disappears from the system.

To define a persistent service (replicated one) we add star to service names in the form $(*N)$. It is worth to mention that ordinary processes still can be defined

using the old form ($ServiceName = ServiceBody$)(see Figure 1); however these processes are not labelled.

Service invocations will be denoted by $\Leftarrow$, thus $N \Leftarrow \{p\}$, is a service with name N which is invoked with a protocol p. In other words, the invocation operator is a composition operator between the invoked service and the protocol service. The actual sessioning will not be activated until the invocation is placed in parallel with the service definition due the fact that calling a service by a client will not take place until this service is available online on the server side.

Although service definitions and invocations can be encoded in CSP as a simple definition and calling for processes, the new operators ($\Rightarrow, \Leftarrow$) are needed to trigger the sessioning algorithm. Sessions are not only important to avoid interference between instances of the service, it is also important to abstract the design model from having distinct definition for each instance.

To clarify these ideas consider the following examples:

Example 1(asynchronous communication): A warehouse system contains service $orderTrans$ which receives orders from customers and replies with an acknowledgement letter *Received* informing the customer that the order has been received. Let $C1$ and $C2$ be customer services. Let services $C1$ and $C2$ invoke service $orderTrans$ to process two different orders. A system which has these services communicating asynchronously is defined as follows:

```
orderTrans ⇒ order?x?y; Received.x.y
C1 = orderTrans ⇐ {order!3!2; Received.x.y}
C2 = orderTrans ⇐ {order!6!44; Received.x.y}
system = (C1 ‖ ⋆orderTrans) ‖ C2
```

The system (*system*) will be reduced to:

```
...
→ l1:(order!3!2; Received.x.y) ‖ l1:(order?x?y; Received.x.y) ‖ ⋆orderTrans ‖
(orderTrans ⇐ {order!6!44; Received.x.y})

→ l1:(order!3!2; Received.x.y) ‖ l1:(order?x?y; Received.x.y) ‖ l2:(order?x?y;
Received.x.y) ‖ ⋆orderTrans ‖ l2:(order!6!44; Received.x.y)

= (l1.order!3!2; l1.Received.x.y) ‖ (l1.order?x?y; l1.Received.x.y) ‖
(l2.order?x?y; l2.Received.x.y) ‖ ⋆orderTrans ‖ (l2.order!6!44; l2.Received.x.y)
```

In this system, we have two different instances of service $orderTrans$, with two different customers in two different sessions (l_1 and l_2). Notice that $l_1.order?x?y$ will match only $l_1.order!3!2$ where the channel name *order* in session l_1 has been renamed to be $l_1.order$.

Example 2(synchronous communication): considering *Example 1*, let the service $orderTrans$ and the customers synchronise on the *order*.

```
system = (C1 ‖{order} ⋆orderTrans) ‖{order} C2
```

The new system will be reduced to:

```
    ...
→ l1:(order!3!2; Received.x.y) ‖{l1.order} l1:(order?x?y; Received.x.y) ‖
l2:(order?x?y; Received.x.y) ‖ ⋆orderTrans ‖{l2.order} l2:(order!6!44; Received.x.y)
    ...
```

Notice that in this system all events inside the session should be labelled even the synchronisation set's events.

Nested Sessions. Basically, a business transaction is a hierarchy of activities whose execution needs to be orchestrated. Therefore, sessions with nested sessions are the standard behaviour of SOA systems.

In case of a service invocation inside an active session, a nested session will be created. Session names are always freshly generated, and services will be always labelled with the innermost session name, as the example below illustrates.
Example 3(sessions in two levels): considering *Example 1*, assume the policy in $C2$'s company requires that, when the customer receives the acknowledgement from the warehouse, a messenger agent is invoked with the order, so the approved order can be delivered to the needed departments.

```
C2= orderTrans ⇐ {order!6!44; Received.x.y; MSNGer⇐ {order.x.y} }
```

```
system = (C1 ‖{order} ⋆orderTrans) ‖{order} (C2‖ MSNGer)
```

The system will be reduced to:

```
   ...
→ l1:(order!3!2; Received.x.y) ‖{l1.order} l1:(order?x?y; Received.x.y) ‖
l2:(order?x?y; Received.x.y) ‖ ⋆orderTrans ‖{l2.order} l2:(order!6!44; Received.x.y;
MSNGer⇐ {order.x.y}) ‖ MSNGer

→ (l1.order!3!2; l1.Received.x.y) ‖{l1.order} (l1.order?x?y; l1.Received.x.y) ‖
(l2.order?x?y; l2.Received.x.y) ‖ ⋆orderTrans ‖{l2.order}
(l2.order!6!44; l2.Received.x.y; l2:(r:(order.x.y)) ) ‖ r:MSNGer
   ...
```

One of the significant features of the CSP is the multi-way handshaken communication. In SoaCSP, multi-way handshaken is preserved within the same session, but is not held across multi-sessions.

Multi-party sessions. Due to the multi-way communication of CSP one session can have more than two participants.
Example 4: considering *Example 3*, assume the customer $C1$ invokes $orderTrans$ with a protocol consisting of two parallel processes.

```
C1 = orderTrans ⇐ { Purchasing ‖ Accounting }
```

In this example a session will be created which has $orderTrans$, $Purchasing$ and $Accounting$ in one session.

Nested invocations in client side or server side will create nested sessions as shown above. In SoaCSP, we allow nested invocations to join the current session instead of creating nested sessions when the designer explicitly indicates that by using ($\Leftarrow^+$ instead of $\Leftarrow$ in invocations. In this case the session will take the parent session's label.
Example 5, considering *Example 3*, let $C2$ be defined as follows:

```
C2= orderTrans ⇐ {order!6!44; Received.x.y; MSNGer⇐+ {order.x.y} }
```

Then the system will be reduced to:

```
   ...
→ (l1.order!3!2; l1.Received.x.y) ‖{l1.order} (l1.order?x?y; l1.Received.x.y) ‖
(l2.order?x?y; l2.Received.x.y) ‖ ⋆orderTrans ‖{l2.order}
(l2.order!6!44; l2.Received.x.y; l2:(order.x.y)) ) ‖ l2:MSNGer
   ...
```

Extra-Sessions Communication. Nested sessions can throw a result out of their scope or get an input from the environment by tagging a channel name with ($\updownarrow$), i.e. $a \updownarrow ?x$ and $a \updownarrow !v$. In this case the channel will have no labels. A tagged action only matches another tagged action, e.g. $a \updownarrow ?x$ with $a \updownarrow !v$. Normal channels ($a?$,$a!$) can be seen as private channels within a session, whereas tagged channels ($a \updownarrow ?$, $a \updownarrow !$) can be seen as global channels on a system.
Example 6 (out of session communication): consider an asynchronous version of *Example 3*. Assume the server side, where $orderTrans$ resides, has an *Archive* service where all the orders from all customers are kept in the company archive database.

```
orderTrans ⇒ order?x?y; Received.x.y; Archive⇐ {store↕!x!y}

Archive= store↕?ARCx?ARCy; database↕.ARCx.ARCy)

system = (C1 ‖ (⋆orderTrans‖ ⋆Archive) ) ‖ (C2‖ MSNGer)
```

This system will be reduced to:

```
   ...
→ (l1.order!3!2; l1.Received.x.y) ‖ (l1.order?x?y; l1.Received.x.y; l1:(m:
(store↕!x!y)) ) ‖ (l2.order?x?y; l2.Received.x.y; l2:(n:(store↕!x!y))) ‖ ⋆orderTrans‖
m:(store↕?ARCx?ARCy; database↕.ARCx.ARCy) ‖ n:(store↕?ARCx?ARCy;
database↕.ARCx.ARCy) ‖ ⋆Archive ‖ (l2.order!6!44; l2.Received.x.y;
l2:(r.order.x.y) ‖ r:MSNGer

→ (l1.order!3!2; l1.Received.x.y) ‖ (l1.order?x?y; l1.Received.x.y; store!x!y) ‖
(l2.order?x?y; l2.Received.x.y; store!x!y) ‖ ⋆orderTrans‖ (store?ARCx?ARCy;
database.ARCx.ARCy) ‖ (store?ARCx?ARCy; database.ARCx.ARCy) ‖ ⋆Archive ‖
(l2.order!6!44; l2.Received.x.y; l2:(r.order.x.y) ‖ r:MSNGer
   ...
```

As a special case of global communication, sessions can throw a result out of their scope directly to the parent scope or get an input from the parent scope. This can be achieved by attaching ($\uparrow$) to channel names, i.e. $a \uparrow ?x$ and $a \uparrow !v$. In this case the channel will have the parent's session label not its session label.

Parent sessions can feed or accept results directly to and from their nested sessions by attaching ($\downarrow$) to channels names, i.e. $a \downarrow ?x$ and $a \downarrow !v$. In this case the channel will have a set of labels containing the nested sessions labels. It can be thought of this type of channels as shared channels between all sub-sessions but are used by only one at a time.

Session Termination. In SoaCSP, as in CSP and cCSP, terminal events always synchronise even if the communication is asynchronous. However, in SoaCSP, we need to introduce the right scope for the different activities of a process.

In cCSP, if parent transaction blocks terminate unsuccessfully then the nested blocks are forced to terminate immediately, but if parents blocks terminate successfully then nested blocks determine internally their termination. The same concept will be adopted for sessions, where the joint terminal events determine the final fate of sessions. However, session termination should be programmed carefully to ensure graceful termination. Therefore, when one side of a session terminates it should inform the other side, and the other side should be able to respond and close or continue its execution.

To avoid interference between sessioning and compensation mechanisms we implement the following policy: (i) Unsuccessful terminal events should prop-

agate through the entire transaction and force processes to shutdown even if they work under different session hierarchies as long as they are included in the same transaction block. (ii) Successful terminations should propagate gradually through the current sessions hierarchy to ensure safe termination.

Recorded compensations should not be tied to sessions, because after their execution their sessions should have been closed. Therefore, recorded events and processes for compensations should be unlabelled. In compensations, the order is important, however scoping is not, which corresponds to our design choice. For instance, if two processes updated a shared database within two different sessions, and later the system failed to terminate successfully, these updates should be compensated in the right order despite their original sessions. The main example in Section 6 explains this idea further.

To implement graceful termination in SoaCSP, we introduce new terminal events to represent session termination (\$), and session interruption (†). Additionally, following the CSP and cCSP assumption, where terminal signals cannot be used directly in processes, we introduce the primitive processes $CLOSE$ and $LISTEN$ to represent \$ and † signals respectively.

When one side of a session terminates then it evolves to the suitable primitive process followed by a labelled $CLOSE$, i.e session name:$CLOSE$, as a sign that this side of the session has terminated.

When one of the session participants terminates then its siblings can interrupt their execution if they have $LISTEN$. It can be be thought of $LISTEN$ as a special $YIELD$ which responds to $CLOSE$ if there is any, or lets the process continue its execution.

Optionally, the terminal event \$ can trigger the execution of a predefined terminal handler by using the new operator ▶. In other words, if p and q are processes then $p \blacktriangleright q$ will behave as p until a \$ is evaluated then the execution is passed to q.

Additionally, a sibling to a terminated service continues its execution as long as it does not execute synchronous event. If it executes a synchronous event then it should be forced to close to avoid dangling processes because of synchronisation.

Similar to the distributed termination in the original CSP [34], in SoaCSP we make a parallel composition terminate when all of the combined processes terminate. However, in SoaCSP processes might work in different layers due to sessioning. Therefore, symmetrically we introduce a distributed sessions closing: a session terminates when all of its participants terminate including sub sessions. When a session terminates it should inform its parent by levelling up $CLOSE$ when there is no parent $CLOSE$ turns to $SKIP$.

To clarify these ideas consider the following examples:

Example 7,(simple session termination): Reverting to *Example 1* the dynamic reduction of the system should be as follows:

$$
\begin{aligned}
&\ldots \\
&\rightarrow ((l_1{:}CLOSE) \parallel (l_1{:}CLOSE)) \parallel ((l_2{:}CLOSE) \parallel (l_2{:}CLOSE)) \\
&\rightarrow (l_1{:}CLOSE) \parallel (l_2{:}CLOSE) \quad \rightarrow SKIP \parallel SKIP \quad \rightarrow SKIP
\end{aligned}
$$

Example 8,(nested session termination): Reverting to *Example 3* the dynamic reduction of the system should be as follows:

...

$\rightarrow$((l$_1$:$CLOSE$) $\parallel$ (l$_1$:$CLOSE$)) $\parallel$ ((l$_2$:(r:$CLOSE$ $\parallel$ r:$CLOSE$))) $\parallel$ (l$_2$:$CLOSE$))

$\rightarrow$ (l$_1$:$CLOSE$) $\parallel$ ((l$_2$:(r:$CLOSE$))) $\parallel$ (l$_2$:$CLOSE$))

$\rightarrow$ (l$_1$:$CLOSE$) $\parallel$ ((l$_2$:$CLOSE$)) $\parallel$ (l$_2$:$CLOSE$))

$\rightarrow$ (l$_1$:$CLOSE$) $\parallel$ (l$_2$:$CLOSE$) $\rightarrow SKIP \parallel SKIP$ $\rightarrow SKIP$

In these two examples, we omitted the $SKIP$ processes following the CSP rule $SKIP; p = p$ [18].

This scenario for session terminations assumes successful termination. However, $CLOSE$ always turns at last to $SKIP$ even in cases of unsuccessful termination (see Section 5) for justification.

The termination mechanism of SoaCSP ensure safe termination and simultaneously clean the system from labels generated during service invocations.

3.1 SoaCSP Syntax

The syntax of SoaCSP is given in Figure 4. SoaCSP extends DEcCSP's syntax with operators to facilitate service invocation and sessioning.

Service invocations can be represented in SoaCSP using the invocation operator ($\Leftarrow$). In ($N \Leftarrow [p]$) the service N is called with the protocol p.

The *Labelling* operator is used to identify session boundaries in SoaCSP. Sessions can be nested. In the normal cases sub-sessions determine their termination internally. However, isomorphic to cCSP semantics, in the cases of abnormal terminations sub-sessions should be aborted if the parent session is terminated.

In SoaCSP, to represent session termination, new terminal signals (events) have been added to the calculus: (\$) represents session closing; (†) represents listening to sassion closing. In addition to these terminal events new primitive processes have been introduced: $CLOSE$ is a process that closes a session then terminates; $LISTEN$ is a process that listens to session closing and terminates. A new operator $\blacktriangleright$ has been added to implement terminal handler. In $p \blacktriangleright q$, q is the terminal handler of p.

Structural congruence. The structural congruence ($\equiv$) laws in Figure 5 are defining the least congruence relation on SoaCSP process terms. These rules comprise the extensions and the reductions of the Labelling operator. $f_p(l)$ is defined as a function which returns the parent session for the session l, if there is no parent it returns nothing. $f_c(l)$ is defined as a function which returns one of the session l's children, if there are no children then returns nothing.

3.2 Operational Semantics of SoaCSP

We present the operational semantics of the mobility-free fragment of SoaCSP in Figure 3.2; the mobility semantics is presented in Section 4.2. SoaCSP operational semantics is based on the operational semantics that we developed for DEcCSP [20] (see Section 2). Configurations now contain two stores. The first

(Standard Processes)		(Compensable Processes)	
$p, q ::= \quad \ldots$	(DEcCSP syntax)	$pp, qq ::= \quad \ldots$	(DEcCSP syntax)
$\mid N \Rightarrow p$	(Service definition)	$\mid NN \Rightarrow pp$	(Service definition)
$\mid *N$	(Persistent Service)	$\mid *NN$	(Persistent Service)
$\mid N \Leftarrow \{p\}$	(Service Invocations)	$\mid NN \Leftarrow \{pp\}$	(Service Invocations)
$\mid N \Leftarrow^{+} \{p\}$	(Join Invocations)	$\mid NN \Leftarrow^{+} \{pp\}$	(Join Invocations)
$\mid l : p$	(Labelling)	$\mid l : pp$	(Labelling)
$\mid CLOSE$	(Primitive Process)	$\mid CLOSEE$	(Primitive Process)
$\mid LISTEN$	(Primitive Process)	$\mid LISTENN$	(Primitive Process)
$\mid p \blacktriangleright q$	(Termination Handler)		

Fig. 4. SoaCSP Syntax

$$
\begin{array}{ccc}
l : (a; p) \equiv l.a; l : p & l : (a!v; p) \equiv l.a!v; l : p & l : (a?x; p) \equiv l.a?x; l : p \\
l : (r : (a; p)) \equiv l : (r.a; r : p) & l : (r : (a!v; p)) \equiv l : (r.a!v; r : p) & l : (r : (a?x; p)) \equiv l : (r.a?x; r : p) \\
l : (a \updownarrow; p) \equiv a; l : p & l : (a \updownarrow !v; p) \equiv a!v; l : p & l : (a \updownarrow ?x; p) \equiv a?x; l : p \\
l : (a \uparrow; p) \equiv f_p(l).a; l : p & l : (a \uparrow !v; p) \equiv f_p(l).a!v; l : p & l : (a \uparrow ?x; p) \equiv f_p(l).a?x; l : p \\
l : (a \downarrow; p) \equiv f_c(l).a; l : p & l : (a \downarrow !v; p) \equiv f_c(l).a!v; l : p & l : (a \downarrow ?x; p) \equiv f_c(l).a?x; l : p
\end{array}
$$

Fig. 5. Structural Congruence

one is a collection of integer and channel-name locations, and is called the *local* store. We use σ, σ',... to represent its different states. The second one is a collection of process locations, and is called the *global* store. We use ρ, ρ',... to represent its different states. Configurations are written as $((p, \sigma), \rho)$, or $((pp, \sigma), \rho)$.

The local store keeps track of the values of data variables and channel names in the scope of the associated process. The global store keeps track of the values of process variables in the full space of configurations. The state of the global store is only changed when a new process variable has been declared or a process variable is assigned a new value. We assume that no type errors occur (the development of a type system for our calculus is left for future work).

Below we present the transition system that defines the semantics of the new extensions in SoaCSP, the rest of SoaCSP is similar to DEcCSP. In the semantics, e and b are evaluated according to the standard integer and Boolean semantics.

In the sequel, we define $\mathbb{N}$ and $\mathbb{L}$ to be the sets of service names and label names respectively. We define λ to be the set of service definitions or invocation events. $\lambda = \{N\top \mid N \in \mathbb{N}\} \cup \{N\top l \mid N \in \mathbb{N} \wedge l \in \mathbb{L}\} \cup \{N\bot l \mid N \in \mathbb{N} \wedge l \in \mathbb{L}\} \cup \mathbb{N}$

Hence, observable events will be defined as the set of processes actions and services actions including invocations and definitions. We write Σ^{λ} to denote the set of process actions and service actions. $\Sigma^{\lambda\tau}$ denotes the set of observable events with the silent event. $\Sigma^{\lambda\Omega}$ denotes the set of observable events with the terminal events. $\Sigma^{\lambda\Omega\tau}$ denotes the set of observable events with the silent event and terminal events.

$$\text{(label)}\ \frac{p \xrightarrow{a} p'}{l{:}p \xrightarrow{l.a} l{:}p'} \qquad \text{(new-label)}\ \frac{p \xrightarrow{a} p'}{(new\ l)\ p \xrightarrow{a} (new\ l)\ p'}$$

Where $a \in \Sigma^{\lambda \Omega \tau}\ \wedge\ l \in \mathbb{L}$ in (label) and (new-label)

$$\text{(srv-inv)}\ \frac{}{((N \Leftarrow \{q\},\sigma),\rho) \xrightarrow{N \perp l} ((l{:}q,\sigma),\rho)}\ \ l \notin \alpha q$$

$$\text{(srv-def1)}\ \frac{}{((N,\sigma),\rho) \xrightarrow{N \top} ((N \Rightarrow p,\sigma),\rho)} \qquad \text{(srv-def2)}\ \frac{}{((N \Rightarrow p,\sigma),\rho) \xrightarrow{N \top l} ((l{:}p,\sigma),\rho)}\ l \notin \alpha p$$

$$\text{(Seq1)}\ \frac{((p,\sigma),\rho) \xrightarrow{a} ((p',\sigma'),\rho)}{((p\,;q,\sigma),\rho) \xrightarrow{a} ((p',\sigma')\,;(q,\sigma),\rho)}\ (a \in \Sigma^{\lambda' \tau}) \qquad \text{(Seq2)}\ \frac{((p,\sigma),\rho) \xrightarrow{\surd} ((p',\sigma),\rho)}{((p\,;q,\sigma),\rho) \xrightarrow{\tau} ((q,\sigma),\rho)}$$

$$\text{(Seq3)}\ \frac{((p,\sigma),\rho) \xrightarrow{\omega} ((STOP,\sigma),\rho)}{((p\,;q,\sigma),\rho) \xrightarrow{\omega} ((STOP,\sigma),\rho)}\ (\omega \in \{!,?\}) \qquad \text{(Seq4)}\ \frac{((p,\sigma),\rho) \xrightarrow{N \top} ((p',\sigma),\rho)}{((p\,;q,\sigma),\rho) \xrightarrow{a} ((p',\sigma')\parallel(q,\sigma),\rho)}$$

$$\text{(ps-def1)}\ \frac{}{((*N,\sigma),\rho) \xrightarrow{N \top} ((*N \Rightarrow p,\sigma),\rho)} \qquad \text{(ps-def2)}\ \frac{}{((*N \Rightarrow p,\sigma),\rho) \xrightarrow{N \top l} ((l{:}p,\sigma) \parallel (*N \Rightarrow p,\sigma),\rho)}\ l \notin \alpha p$$

$$\text{(sessions)}\ \frac{((p,\sigma),\rho) \xrightarrow{N \top l} ((p',\sigma),\rho) \qquad ((q,\sigma),\rho) \xrightarrow{N \perp l} ((q',\sigma),\rho)}{((p \underset{A}{\parallel} q,\sigma),\rho) \xrightarrow{N} (new\ l)((p',\sigma) \underset{A}{\parallel} (q',\sigma),\rho)}\ \ l \notin \Sigma$$

$$\text{(join-ses)}\ \frac{}{((N \Leftarrow^{+} \{q\},\sigma),\rho) \xrightarrow{N \perp l} ((f_p(l){:}q,\sigma),\rho)}\ \ l \notin \alpha q$$

$$\text{(close)}\ \frac{}{((CLOSE,\sigma),\rho) \xrightarrow{\$} ((SKIP,\sigma),\rho)} \qquad \text{(listen)}\ \frac{}{((LISTEN,\sigma),\rho) \xrightarrow{\omega} ((STOP,\sigma),\rho)}\ \omega \in \{\surd, \dagger\}$$

$$\text{(term1)}\ \frac{}{((l{:}p,\sigma),\rho) \xrightarrow{l.\omega} ((p';l{:}CLOSE,\sigma),\rho)}\ \omega \in \Omega \qquad \text{(term3)}\ \frac{}{((l{:}CLOSE,\sigma),\rho) \xrightarrow{\tau} ((f_p(l){:}ClOSE,\sigma),\rho)}$$

$$\text{(term2)}\ \frac{}{((l{:}CLOSE \parallel l{:}CLOSE,\sigma),\rho) \xrightarrow{\tau} ((l{:}CLOSE,\sigma),\rho)}$$

$$\text{(term-hdr1)}\ \frac{((p,\sigma),\rho) \xrightarrow{a} ((p',\sigma),\rho)}{((p \blacktriangleright q,\sigma),\rho) \xrightarrow{a} ((p' \blacktriangleright q,\sigma),\rho)}\ (a \in \Sigma^{\tau}) \qquad \text{(term-hdr2)}\ \frac{((p,\sigma),\rho) \xrightarrow{\$} ((p',\sigma),\rho)}{((p \blacktriangleright q,\sigma),\rho) \xrightarrow{\tau} ((q,\sigma),\rho)}$$

$$\text{(term-hdr3)}\ \frac{((p,\sigma),\rho) \xrightarrow{\omega} ((p',\sigma),\rho)}{((p \blacktriangleright q,\sigma),\rho) \xrightarrow{\omega} ((p',\sigma),\rho)}\ (\omega \in \Omega - \{\$\})$$

Fig. 6. SoaCSP Operational Semantics

Hoare's labelling operator. According to Definition 1, the labelling operator can be written as (label). In (new-label) we add the keyword *new* to refresh labels to a new name.

Service Definitions, Invocations and sessioning.

Service invocation: For service invocation we introduce rule (srv-inv) which says that the process q is ready to establish a new session l.

Service Definition: In SoaCSP, processes can be defined as $(N = p)$. If we want to define an invocable service then we should write it as $(N \Rightarrow p)$. Thus, N will not be reduced to p, but it will be reduced to labelled p. However the reduction will be done in two steps due to the following reason:

If service names appear in a sequential composition, e.g. $a; b; c$ where b is defined as $b \Rightarrow p$. This implies that a will be evaluated firstly, then service definition will be evaluated secondly $l : p$. If the service definition synchronises with a service invocation then c can start. However, if p is a persistent service then c can not be reached.

Basically, invocable services are meant to be standalone services working in parallel with other services that reside in a server. Therefore, if a service definition $(N \Rightarrow p)$ or a service name (N) where N is defined as $(N \Rightarrow p)$ are mentioned in a code, this should be considered as publishing of the service on the server. Publishing means making the service work in parallel with other services.

Thus, in $a; b; c$ where b is defined as $b \Rightarrow p$: first, a will be evaluated; second service definition should be published $b \Rightarrow p; c$ then evaluated $l : p \| c$.

Therefore, the reduction is done in two steps: (i) (srv-def1) replaces the name with the definition of a service; (i) (srv-def2) reduces the definition to a labelled process.

To do the publishing we should adjust sequential composition semantics to be as follows in rules (seq1 - seq4).

Here, we define λ' to be: $\{N \top l \mid N \in \mathbb{N} \wedge l \in \mathbb{L}\} \cup \{N \bot l \mid N \in \mathbb{N} \wedge l \in \mathbb{L}\} \cup \mathbb{N}$

Definition of persistent services: A persistent service can be defined in SoaCSP as $*N$ where N is defined as $N \Rightarrow p$. $*N$ behaves as $l : (p \,\|\, l_1 : (p \,\|\, l_2 : (p \,\|\, l_3 : (p \,\|\, \cdots))))$. To reflect this behaviour we update the rules (srv-def1) and (srv-def2) to (ps-def1) and (ps-def2).

Session: A definition and an invocation in a parallel compensation create a session and the two processes agree in a fresh new name for the session as shown in rule (session). Symmetrically, we develop a sessioning reduction relation to emphasize that the two rules of invocation and definition are happening at the same time. $\frac{}{((N \underset{A}{\|} N \Leftarrow [q], \sigma, \rho)) \longrightarrow (new\, l)((p, \sigma) \underset{A}{\|} (q, \sigma), \rho)} \quad l \notin \Sigma$

Multi-party Sessions. As mentioned before, $\Leftarrow^+$ in rule (join-ses) is used to invite nested sessions to join the parent session.

Sessions and Nested Sessions Termination

The graceful termination algorithm which we suggested before is formally defined by rules (term1 - term3).

In rule (term1), if a process executes a terminal event it will evolve to the primitive process which is suitable with that terminal event, followed by the special primitive process for closing sessions $CLOSE$ (close).

Isomorphic to the distributed termination of the CSP, the distributed sessions closing means that a session terminates when all of its participants terminate as shown in rule (term2).

Rule (term3) says that when the session terminates then the primitive process $CLOSE$ will be levelled up.

Optionally, processes can have the special primitive process $LISTEN$ (listen) to interrupt their execution if their sibling has terminated.

We add to (syn-table) (see Figure 1) the following lines:

	$\$$	!	$\$$	?	$\surd$	†
ω						
ω'	$\$$	†	†	†	†	†
$\omega \& \omega'$	$\$$	!	$\$$	?	$\surd$	†

The new *Termination handler* operator (term-hdr1 - term-hdr3) can be used to start a termination handler in case of a session closure.

4 The Full Calculus

One of the significant features of SOA systems is their ability to dynamically reconfigure themselves. Therefore, any process calculus proposed for modelling SOA systems should be equipped with primitives facilitating dynamic reconfiguration, more specifically, primitives to model delegation of the communication

capabilities from one service to another. Passing channel names is a way of transferring communication capabilities, usually called *mobility*.

We believe that the main reason CSP has not been used intensively in SOA modelling so far is its lack of support for mobility. Introducing mobility into CSP as in the π-calculus is not straightforward for the following reason: the parallel composition in CSP is parametrised with an interface set which governs the synchronisation between participants. Events in this set should be simultaneously performed by all participants whereas events outside this set (even if they are shared) are not. Although CSP parallel composition improves communication flexibility, it lets processes alphabets play a significant role in the communication. Therefore, the silent growth of alphabets as in π-calculus is not enough. Processes alphabets should be grown explicitly because of its relation with the parallel operator.

To overcome this problem several solutions has been proposed: in [40], occam-π is presented as a mobile version of occam (a concurrent programming language founded on CSP). Mobility is obtained in occam-π by declaring communicated channels as a bundle of a fixed number of sub-channels with flexible ends. Mobile communications are employed by processes connecting and disconnecting from these ends.

Replacing channels in a process alphabet with a bundle of channels is not strictly increasing the communication capability at runtime, instead it offers a range of possible communication that can be adjusted at runtime. Using this model to introduce mobility into the original CSP will complicate the theory and will not add all the expressivity of the π-calculus mobility.

Another mobility model has been proposed in [33], and has been adopted in (CSP $\|$ B) in [35]. In this model, mobility is obtained by sending the right to use a channel instead of sending the channel name to avoid changing the actual processes alphabets. This mobility model has been employed in Hoare's CSP [18].

Introducing rights beside channels complicates the calculus and does not strictly add mobility: the processes alphabets are still fixed and the rights to use channels are circulated. Additionally, using the classical CSP communication compromises the new capabilities introduced with the generalised version of the parallel composition [34].

A promising mobility model is presented in [34], where the actual channels are circulated and the processes alphabets are dynamically changed. In this model, channels are sent with marks $(-,+)$ to guide changes to processes alphabets. It introduces a mobile version of the Hoare alphabetised parallel operator to ensure that participant alphabet changes are included in communications every time. However, this restricts CSP communication patterns. Additionally, in the semantics of this model a restriction has been made that only one channel can be communicated through another channel which comprises the multi-simultaneous communication of CSP.

Each of these models have some drawbacks, therefore, in this section, we propose a new mobility model to accommodate mobility in SoaCSP; the same model

can be used to introduce mobility into standard CSP. Our mobility model will use the marks, as the last model, to guide alphabet changes; however we consider also the no-change case. The novelty in our model is the introduction of a new dynamic algorithm to update the synchronisation set of the generalised parallel operator. Additionally, we relax the restriction of one communicated channel per channel and allow channels to carry muti-channels in one communication.

We argue that our mobility model is easy to use and more expressive than the previously mentioned models. It accommodates the well-known name-passing mobility model of the π-calculus and it does not comprise the original capabilities of CSP communications.

Example 7: the π-calculus communication $(\bar{x}y|x(w).\bar{w}c)$ can be rewritten in SoaCSP as $(x!(y)^+ \| x?(w)^+; w!c)$, where the parentheses () are used to identify channels names and the $(+)$ marks are used to augment alphabets with the new name and at the same time enforce synchronisation over this name.

Before giving the formal syntax and semantics of the full calculus, we discuss in Section 4.1 the mobility ideas informally.

4.1 A mobility model for SoaCSP

To achieve mobility maintaining the expressivity of the calculus, we will allow channel names to be passed as part of the data sent on channels. Channel names will be distinguished from the other components by enclosing them in brackets. For instance, $c?x!y!z?(m)$ will denote a communication through the channel c where the process accepts an input for the variable x, outputs the values in variables y and z, and in addition accepts the new connection m. There is no restriction on the number of components sent through channels. For instance, $a!(c)?(m)$ is a communication through the channel a where the process sends the connection c and receives the new connection m.

It is worth to mentioned that restricting communication to names as the π-calculus, is insufficient in the CSP. The restriction will compromise the CSP typed multi-way communications (see [19,34]).

the important distinction between π-calculus and CSP, that CSP parallel composition relies on a synchronisation set which should be explicitly updated to maintain synchronisation as the system evolves, if mobility is permitted.

Therefore, to deal with synchronisation, we associate to each process a synchronisation set (SS), which is initially empty ($SS = \emptyset$). This set will be updated as the system evolves, where at any point in the system if p is a process, then $(SS_p \subseteq \alpha p \subseteq \Sigma)$. In the model, we allow communications to be marked with $(-,+)$ to guide changes to this set.

The standard communication model of CSP will be extended to include the following kinds of communicated events:

- **Output channels:** Outputting channel names will be written as $c!(cm)$, where c is the current channel and cm is the communicated name. This form of output event can be optionally decorated with marks $(-)$, as follows:

  - **Output with no marks:** $c!(cm)$ means that the process sends the connection (cm) and at the same time keeps it in its synchronisation set. Therefore, no updates are needed in the process synchronisation set SS.
  - **Output with $-$:** $c!(cm)^-$ means that the process sends the connection (cm) and at the same time releases it. Therefore, the communicated name (cm) should be removed from the process synchronisation set (SS becomes $SS - \{cm\}$).
- **Input channels:** Inputting channel names will be written as $c?(cm)$, where c is the current channel and cm is the communicated name. This form of input event can be optionally decorated with marking $(+)$, as follows:
  - **Input with $+$:** $c?(cm)^+$ means that the process receives the connection (cm) and at the same time adds it to its synchronisation set. Therefore, the process synchronisation set will be augmented with this name (SS becomes $SS \cup \{cm\}$).
  - **Input with no marks:** $c?(cm)$ means that the process accepts the connection (cm) as asynchronous connection. Therefore, no changes are made in the process synchronisation set SS.

Processes Alphabets In our mobility model, handshaken communication is achieved by one form of input channel event synchronising with one form of output channel event. SoaCSP's parallel composition is a generalised parallel composition, parameterised with two sets: the fixed synchronisation set which designers provide, as in DEcCSP, and another set which will be updated dynamically. The second set is the intersection of the individual (SS)s for each process participating in the parallel composition. An outputting process alphabet (α) will not be affected. However, inputting processes' alphabets (α) will be updated with the new channel name if it is not already there. In this way the set SS for a process p will always be included in p's alphabet.

Open-World Communication So far, we made the assumption that all processes run in a closed world. In other words, they are running in the same system under one Σ, where channels which are already in Σ are being passed around. However, SOA systems can be expanded with new processes from outside the system. In this case, the new process alphabet will join Σ and if the new names are already in Σ, then they will be α-converted before they join.

Mobile Sessions Mobile channels facilitate sessions expansions and migration, because sessions are related in our calculus to channels, and channels can be sent from one process to another process to delegate communications. In case a channel has been sent then the associate session will have one of two options: delegate the session to the new process or expand the session to have the new process. As aforementioned, channels names inside sessions will be renamed by the session label even if they are mentioned inside $()$,$()^+$, or $()^-$ constructs. On the other hand, channels variables are not labelled. To further illustrate this issue consider the following example:

Example 8,

See sheet 1.

The rest of this section will be devoted to presenting the full SoaCSP's syntax and operational semantics in Sections 4.2 and 4.3 respectively.

4.2 Full SoaCSP syntax

The syntax of the full SoaCSP is given in Figure 7. The full version extends SoaCSP with primitives to facilitate channel communications.

As mentioned in Section 2, events in CSP can be classified into *ordinary events*, which can be single or compound names (e.g., a or $a.b$); and *channels*, which are ordinary events composed with integer expressions to denote communicated data. Although standard CSP allows data of different types to be communicated, in our semantics, for simplicity, we assume that the communicated data are integers or literal names.

The novelty here is that channel communication can also include channel names. This is represented using brackets () and its associated marking $(+, -)$, thus channel names can be passed through existing channels.

To implement mobility, we have modified the parallel composition operator: it is parameterised with two sets instead of one. The first is the *fixed synchronisation set*, which the designer provides, as in CSP. The second is the *dynamic set*, which is updated according to the communication of mobile channels.

In addition, we extend the SoaCSP syntax to allow named processes to have parameters: integer expressions or named events.

4.3 Operational Semantics of the full SoaCSP

The operational semantics of the full SoaCSP is based on the operational semantics that we developed for the mobility-free SoaCSP (see Section 3.2).

Below we present the transition system that defines the semantics of the new mobility extensions in SoaCSP the rest of SoaCSP is similar to the one defined in Section 3.2.

In the semantics, Configurations and stores definitions are similar to Section 3.2. e and b are evaluated according to the standard integer and Boolean semantics.

Literal names can be any name, however in our semantics, for the sake of clarity, we use c to denote channel names (constant names), and we use m to denote mobile channel names (variable names)

Parameterisation

Parameters for processes are written in SoaCSP as functional arguments, as in the standard CSP. Process names can have any finite number of parameters, which can only be integer expressions or named events. Parameters are evaluated before executing processes, as the transition rules indicate. We use $\widetilde{\ell}$ to denote a list of variables $\ell_1, \ell_2, \ldots, \ell_n$; $\widetilde{e}$ to denote a list of expressions $e_1, e_2, \ldots, e_n$; $\widetilde{Z}$ to denote a list of integer values $Z_1, Z_2, \ldots, Z_n$; $\widetilde{a}$ to denote

(Standard Processes)		(Compensable Processes)	
$p, q ::= \ \ldots$	(SoaCSP syntax)	$pp, qq ::= \ \ldots$	(SoaCSP syntax)
$\mid N(Par_1 \ldots Par_n)$	(Process name)	$\mid NN(Par_1 \ldots Par_n)$	(Process name)
$\mid p \underset{A,M}{\parallel} q$	(Parallel operator)	$\mid pp \underset{A,M}{\parallel} qq$	(Parallel operator)

$Par ::= \quad e \quad \mid \quad Names$
$Names ::= \quad name \quad \mid \quad name.Names$

(events)
$a ::= \quad name?INPC \quad$ (Input) $\quad \mid \quad name!OUTC \quad$ (Output)
$\mid \quad name.OUTC \quad$ (undirected) $\quad \mid \quad name \quad$ (Literal name)

(Input components)
$INPC ::= \quad INP \quad \mid \quad INP?INPC \quad \mid \quad INP!OUTC \quad \mid \quad INP.OUTC$
(Input elements)
$INP ::= \quad name \quad$ (Literal name) $\quad \mid \quad \ell \quad$ (Integer variable)
$\quad \mid \quad (name)^+ \quad$ (Input channel name then add it to SS)
$\quad \mid \quad (name) \quad$ (Input channel name without adding it to SS)

(Output components)
$OUTC ::= \quad OUT \quad \mid \quad OUT?INPC \quad \mid \quad OUT!OUTC \quad \mid \quad OUT.OUTC$
(Output elements)
$OUT ::= \quad name \quad$ (Literal name) $\quad \mid \quad e \quad$ (Integer expression)
$\quad \mid \quad (name)^- \quad$ (Output channel name then remove it from SS)
$\quad \mid \quad (name) \quad$ (Output channel name without removing it from SS)

Fig. 7. The Full syntax of SoaCSP

a list of events $a_1, a_2, \ldots, a_n$; and $\widetilde{Names}$ to denote a list of events names $Names_1, Names_2, \ldots, Names_n$.
Integer parameters in standard processes:

$$\frac{((\widetilde{e},\sigma),\rho) \xrightarrow{\tau} ((\widetilde{e}',\sigma'),\rho)}{((N(\widetilde{e}),\sigma),\rho) \xrightarrow{\tau} ((N(\widetilde{e}'),\sigma'),\rho)} \quad \frac{((p\{\widetilde{\ell} \mapsto \widetilde{Z}\},\sigma),\rho) \xrightarrow{\tau} ((p',\sigma),\rho)}{((N(\widetilde{Z}),\sigma),\rho) \xrightarrow{\tau} (p',\sigma),\rho)} \quad where\ N(\widetilde{\ell}) = p$$

Integer parameters in compensable processes:

$$\frac{((\widetilde{e},\sigma),\rho) \xrightarrow{\tau} ((\widetilde{e}',\sigma'),\rho)}{((NN(\widetilde{e}),\sigma),\rho) \xrightarrow{\tau} ((NN(\widetilde{e}'),\sigma'),\rho)} \quad \frac{((pp\{\widetilde{\ell} \mapsto \widetilde{Z}\},\sigma),\rho) \xrightarrow{\tau} ((pp',\sigma),\rho)}{((NN(\widetilde{Z}),\sigma),\rho) \xrightarrow{\tau} (pp',\sigma),\rho)}$$

$where\ NN(\widetilde{\ell}) = pp$ Event parameters in standard processes:

$$\frac{((p\{\widetilde{a} \mapsto \widetilde{Names}\},\sigma),\rho) \xrightarrow{\tau} ((p',\sigma),\rho)}{((N(\widetilde{Names}),\sigma),\rho) \xrightarrow{\tau} (p',\sigma),\rho)} \quad where\ N(\widetilde{a}) = p$$

Event parameters in compensable processes:

$$\frac{((pp\{\widetilde{a} \mapsto \widetilde{Names}\},\sigma),\rho) \xrightarrow{\tau} ((pp',\sigma),\rho)}{((NN(\widetilde{Names}),\sigma),\rho) \xrightarrow{\tau} (pp',\sigma),\rho)} \quad where\ NN(\widetilde{a}) = pp$$

Channels

According to SoaCSP's syntax (see Figure 1), channel events can only occur in standard processes, more precisely in the prefix operator or in atomic processes. Events, as mentioned in Section 4.2, can be literal names (single or compound), literal names carrying integers as input or output data, or literal names carrying channel names. Channels carrying data represent a class of events in Σ rather than one single value, e.g., $c.Z$ represents the channel c which can communicate any integer value ($c.Z$ means $\{c.x | x \in Z\} \subseteq \Sigma$). Inputting processes are able to receive any element of type Z, whereas outputting processes are only able to communicate one value [34]. Therefore, if $c.Z$ is an input event then it is actually an external choice over all the values of type integer, which is represented elegantly as $c?x$. If $c.Z$ is an output event then it is a single integer value, $c.x$ where $x \in Z$. Output events are represented as $c!x$. In our semantics, $c.x$ always equals to $c!x$, except when x is not in the scope of the process which indicates that this event is a declaration for a new input variable. Channels in McCSP, as in standard CSP, can carry simultaneously multiple values in both directions. For instance, if c is a channel name then the event $(c?x!y!z)$ is a valid event which accepts an integer and sends two integers. We give below transition rules for the cases of prefix operator and atomic process.

Atomic process: Let a be an atomic event in Σ. The process a is the process which performs this atomic event a and then terminates normally. We discuss below each case for events.

If the event is a literal name, that is, the atomic process a is a single or compound name:

$$\frac{}{((a,\sigma),\rho) \xrightarrow{a} ((SKIP,\sigma),\rho)}$$

If the event is an output, that is, the atomic process a is of the form $a!e$ or $a.e$:

$$\frac{((e,\sigma),\rho) \xrightarrow{\tau} ((e',\sigma'),\rho)}{((a!e,\sigma),\rho) \xrightarrow{\tau} ((a!e',\sigma'),\rho)} \quad \frac{}{((a!\ell,\sigma),\rho) \xrightarrow{\tau} ((a!\sigma(\ell),\sigma),\rho)}$$

$$\frac{((e,\sigma),\rho) \xrightarrow{\tau} ((e',\sigma'),\rho)}{((a.e,\sigma),\rho) \xrightarrow{\tau} ((a.e',\sigma'),\rho)} \quad \frac{}{((a.\ell,\sigma),\rho) \xrightarrow{\tau} ((a.\sigma(\ell),\sigma),\rho)}\ \ell \in \sigma$$

$$\frac{}{((a.n,\sigma),\rho) \xrightarrow{a.n} ((SKIP,\sigma),\rho)}\ n \in Z \quad \frac{}{((a!n,\sigma),\rho) \xrightarrow{a.n} ((SKIP,\sigma),\rho)}\ n \in Z$$

If the event is an input, that is, the atomic process a is of the form $a?\ell$ or $(a.\ell\ \wedge\ \ell \notin \sigma)$:

$$\frac{}{((a?\ell,\sigma),\rho) \xrightarrow{a.Z} ((SKIP,\sigma[\ell \mapsto Z]),\rho)} \quad \frac{}{((a.\ell,\sigma),\rho) \xrightarrow{a.Z} ((SKIP,\sigma[\ell \mapsto Z]),\rho)}\ \ell \notin \sigma$$

If the event is outputting a mobile channel, that is, the atomic process a is of the form $a!(m)$ or $a!(m)^-$:

$$\frac{}{((a!(m),\sigma),\rho) \xrightarrow{\tau} ((a!(\sigma(m)),\sigma),\rho)} \quad \frac{}{((a!(m)^-,\sigma),\rho) \xrightarrow{\tau} ((a!(\sigma(m))^-,\sigma),\rho)}$$

$$\frac{}{((a!(c),\sigma),\rho) \xrightarrow{a!(c)} ((SKIP,\sigma),\rho)} \quad \frac{}{((a!(c)^-,\sigma),\rho) \xrightarrow{a!(c)^-} ((SKIP,\sigma),\rho)} \; where\ c \in \Sigma$$

If the event is inputting a mobile channel, that is, the atomic process a is of the form $a?(m)$ or $a?(m)^+$:

$$\frac{}{((a?(m),\sigma),\rho) \xrightarrow{a?(c)} ((SKIP,\sigma[m \mapsto c]),\rho)} \; where\ c \in \Sigma$$

$$\frac{}{((a?(m)^+,\sigma),\rho) \xrightarrow{a?(c)^+} ((SKIP,\sigma[m \mapsto c]),\rho)} \; where\ c \in \Sigma$$

If the event is a mobile channel, that is, the atomic process a is of the form $m?INPC$, $m!OUTC$, or $m.OUTC$:

$$\frac{}{((m?INPC,\sigma),\rho) \xrightarrow{\tau} ((\sigma(m)?INPC,\sigma),\rho)} \quad \frac{}{((m!OUTC,\sigma),\rho) \xrightarrow{\tau} ((\sigma(m)!OUTC,\sigma),\rho)}$$

$$\frac{}{((m.OUTC,\sigma),\rho) \xrightarrow{\tau} ((\sigma(m).OUTC,\sigma),\rho)}$$

The event a is considered as an atomic unit of work that cannot be interrupted. Therefore, the process a can be interrupted before or after performing the event a. We emphasise that if an input event has happened, then the store σ will be updated even if the atomic process interrupted after that and did not terminate normally.

$$\frac{}{((a,\sigma),\rho) \xrightarrow{?} ((STOP,\sigma),\rho)} \quad \frac{}{((a,\sigma),\rho) \xrightarrow{a} ((STOP,\sigma),\rho)}$$

Prefixing: Let a be an event in Σ and let p be a process. Prefixing represents the process that is ready to engage in event a and then behave as p. We discuss below each case for events.

If the event is a literal name, that is, the event a is a single or compound name:

$$\frac{}{(((a \to p,\sigma),\rho) \xrightarrow{a} ((p,\sigma),\rho))}$$

If the event is an output, that is, the event a is of the form $a!e$ or $a.e$:

$$\frac{((e,\sigma),\rho) \xrightarrow{\tau} ((e',\sigma'),\rho)}{((a!e \to p,\sigma),\rho) \xrightarrow{\tau} ((a!e' \to p,\sigma'),\rho)} \quad \frac{}{((a!\ell \to p,\sigma),\rho) \xrightarrow{\tau} ((a!\sigma(\ell) \to p,\sigma),\rho)}$$

$$\frac{((e,\sigma),\rho) \xrightarrow{\tau} ((e',\sigma'),\rho)}{((a.e \to p,\sigma),\rho) \xrightarrow{\tau} ((a.e' \to p,\sigma'),\rho)} \quad \frac{}{((a.\ell \to p,\sigma),\rho) \xrightarrow{\tau} ((a.\sigma(\ell) \to p,\sigma),\rho)}\ \ell \in \sigma$$

$$\frac{}{((a!n \to p,\sigma),\rho) \xrightarrow{a.n} ((p,\sigma),\rho)}\ n \in Z \quad \frac{}{((a.n \to p,\sigma),\rho) \xrightarrow{a.n} ((p,\sigma),\rho)}\ n \in Z$$

If the event is an input, that is, the event a is of the form $a?\ell$ or $(a.\ell \ \wedge \ \ell \notin \sigma)$:

$$\frac{}{((a?\ell \to p,\sigma),\rho) \xrightarrow{a.Z} ((p,\sigma[\ell \mapsto Z]),\rho)} \quad \frac{}{((a.\ell \to p,\sigma),\rho) \xrightarrow{a.Z} ((p,\sigma[\ell \mapsto Z]),\rho)}\ \ell \notin \sigma$$

If the event is outputting a mobile channel, that is, the event a is of the form $a!(a)$ or $a!(a)^-$:

$$\frac{}{((a!(m) \to p,\sigma),\rho) \xrightarrow{\tau} ((a!(\sigma(m)) \to p,\sigma),\rho)}$$

$$\frac{}{((a!(m)^- \to p,\sigma),\rho) \xrightarrow{\tau} ((a!(\sigma(m))^- \to p,\sigma),\rho)}$$

$$\frac{}{((a!(c) \to p, \sigma), \rho) \xrightarrow{a!(c)} ((p, \sigma), \rho)} \quad \frac{}{((a!(c)^- \to p, \sigma), \rho) \xrightarrow{a!(c)^-} ((p, \sigma), \rho)} \quad \textit{where } c \in \Sigma$$

If the event is inputting a mobile channel, that is, the event a is of the form $a?(m)$ or $a?(m)^+$:

$$\frac{}{((a?(m) \to p, \sigma), \rho) \xrightarrow{a?(c)} ((p, \sigma[m \mapsto c]), \rho)} \quad \textit{where } c \in \Sigma$$

$$\frac{}{((a?(m)^+ \to p, \sigma), \rho) \xrightarrow{a?(c)^+} ((p, \sigma[m \mapsto c]), \rho)} \quad \textit{where } c \in \Sigma$$

If the event is a mobile channel, that is, the event a is of the form $m?INPC$, $m!OUTC$, or $m.OUTC$:

$$\frac{}{((m?INPC \to p, \sigma), \rho) \xrightarrow{\tau} ((\sigma(m)?INPC \to p, \sigma), \rho)}$$

$$\frac{}{((m!OUTC \to p, \sigma), \rho) \xrightarrow{\tau} ((\sigma(m)!OUTC \to p, \sigma), \rho)}$$

$$\frac{}{((m.OUTC \to p, \sigma), \rho) \xrightarrow{\tau} ((\sigma(m).OUTC \to p, \sigma), \rho)}$$

Communication

Parallel Communication: Processes can communicate only if they are composed in parallel. McCSP, as the standard CSP, enforces handshasken communication [34], where a communication can only happen if all its participants are prepared to execute this communication. In McCSP, communication is guided by two synchronisation sets which contain the events (or a class of events if a channel's name is included) that should be performed simultaneously. The first one (x) is the fixed set provided by the designer, whereas the second (M) is a dynamic set which is updated if mobile channels are communicated, ($M = SS_p \cap SS_q$, if p and q are parallel processes), as explained in Section 4.1. At the beginning, M will be empty as all the synchronisation sets on the system are empty too. The modified parallel composition operator is defined as follows:

$$\frac{((p, \sigma), \rho) \xrightarrow{b} ((p', \sigma'), \rho)}{((p \underset{x,M}{\parallel} q, \sigma), \rho) \xrightarrow{b} ((p', \sigma'), \underset{x,M}{\parallel} (q, \sigma), \rho)} \quad \frac{((q, \sigma), \rho) \xrightarrow{c} ((q', \sigma'), \rho)}{((p \underset{x,M}{\parallel} q, \sigma), \rho) \xrightarrow{c} ((p, \sigma), \underset{x,M}{\parallel} (q', \sigma'), \rho)}$$

$$\textit{where } \; b, c \notin x \cup M$$

$$\frac{((p, \sigma), \rho) \xrightarrow{a} ((p', \sigma'), \rho) \quad ((q, \sigma), \rho) \xrightarrow{a} ((q', \sigma''), \rho)}{((p \underset{x,M}{\parallel} q, \sigma), \rho) \xrightarrow{a} ((p', \sigma'), \underset{x,M}{\parallel} (q', \sigma''), \rho)} \quad \textit{where } a \in x \cup M$$

M is only updated if there are communications of mobile channels. For the sake of clarity, if there are no communications for mobile channels, we will omit M from the parallel composition. In other words, if M will always remain empty then parallel composition ($\underset{x,M}{\parallel}$) can be written using the old notation ($\underset{x}{\parallel}$).

For a network of n processes, the parallel composition is written as either:

$$(\underset{x,M}{\parallel}{}^{n-1}_{i=0}(p_i, \sigma_i), \rho) \quad \textit{where } M = \bigcap_{i=0}^{n-1} SS_i$$

Therefore, the n processes are synchronised with respect to the set $x \subseteq \Sigma$ and the intersection set of all the SS), or

$$((((((p_0, \sigma_0) \underset{x,M_1}{\parallel} (p_1, \sigma_1)) \underset{y,M_2}{\parallel} (p_2, \sigma_2)) \ldots) \underset{z,M_n}{\parallel} (p_{n-1}, \sigma_{n-1})), \rho)$$

where every pair of processes has different synchronisation sets $(x, y, \ldots z)$. These sets will be associated with other dynamic sets which are calculated in the same way as Hoare's alphabetised parallel composition [18]. Therefore, $M_1 = SS_0 \cap SS_1$, $M_2 = (SS_0 \cup SS_1) \cap SS_2$, ..., and $M_n = (\bigcup_{i=0}^{n-2} SS_i) \cap SS_{n-1}$.

According to the previous parallel rule, any two (or more) equal events can synchronise, even if they are two output events or two input events. This confirms CSP handshaken communication: a communication can only happen if all its participants are prepared to execute this communication. However, if communications involve mobile channels then, according to our mobility model, handshaken communication should be achieved by one form of input channel event synchronising with one form of output channel event. Therefore, we provide special rules for mobile communication events as follows:

$$\frac{((p,\sigma),\rho) \xrightarrow{a!(c)} ((p',\sigma),\rho) \quad ((q,\sigma),\rho) \xrightarrow{a?(c)^+} ((q',\sigma'),\rho)}{((p \underset{x,M}{\parallel} q,\sigma),\rho) \xrightarrow{a.(c)} ((p',\sigma) \underset{x,M'}{\parallel} (q',\sigma'),\rho)}$$

where $SS_{p'} = SS_p \ \wedge \ SS_{q'} = SS_q \cup \{c\} \ \wedge \ M' = SS_{p'} \cap SS_{q'}$

$$\frac{((p,\sigma),\rho) \xrightarrow{a!(c)} ((p',\sigma),\rho) \quad ((q,\sigma),\rho) \xrightarrow{a?(c)} ((q',\sigma'),\rho)}{((p \underset{x,M}{\parallel} q,\sigma),\rho) \xrightarrow{a.(c)} ((p',\sigma) \underset{x,M'}{\parallel} (q',\sigma'),\rho)}$$

where $SS_{p'} = SS_p \ \wedge \ SS_{q'} = SS_q \ \wedge \ M' = SS_{p'} \cap SS_{q'}$

$$\frac{((p,\sigma),\rho) \xrightarrow{a!(c)^-} ((p',\sigma),\rho) \quad ((q,\sigma),\rho) \xrightarrow{a?(c)^+} ((q',\sigma'),\rho)}{((p \underset{x,M}{\parallel} q,\sigma),\rho) \xrightarrow{a.(c)} ((p',\sigma) \underset{x,M'}{\parallel} (q',\sigma'),\rho)}$$

where $SS_{p'} = SS_p - \{c\} \ \wedge \ SS_{q'} = SS_q \cup \{c\} \ \wedge \ M' = SS_{p'} \cap SS_{q'}$

$$\frac{((p,\sigma),\rho) \xrightarrow{a!(c)^-} ((p',\sigma),\rho) \quad ((q,\sigma),\rho) \xrightarrow{a?(c)} ((q',\sigma'),\rho)}{((p \underset{x,M}{\parallel} q,\sigma),\rho) \xrightarrow{a.(c)} ((p',\sigma) \underset{x,M'}{\parallel} (q',\sigma'),\rho)}$$

where $SS_{p'} = SS_p - \{c\} \ \wedge \ SS_{q'} = SS_q \ \wedge \ M' = SS_{p'} \cap SS_{q'}$

The operational rules developed in this section cover only standard processes. Compensable processes have similar rules, because parallel composition rules of compensable processes work in the same way as standard processes.

Finally, we remark that channels are usually used to represent reliable and synchronous communications. Although communication becomes asynchronous when the communication events are not included in the synchronisation set, it is unusual to have asynchronous communication via channels. To represent unreliable media or asynchronous media, processes are used instead.

Sequential Communication: Processes cannot communicate if they are composed sequentially. If two processes are composed sequentially, the second process cannot start until the first one terminates normally. The sequential composition operator disconnects the scopes of its subprocesses, thus any variables before (;) are forgotten after it.

Events are composed sequentially to define processes. Events can be connected using a prefix operator (e.g. $p = c!3 \rightarrow d?x \rightarrow SKIP$) which makes the process uninterruptible, or using the sequential composition (e.g. $p = c!3; d?x;\ SKIP$) which makes the process interruptible (for more information on the distinction between prefix operator and sequential composition see [20]). If sequential composition is used to define the behaviour of a process then events should be turned to atomic processes because events cannot be used in their event form in sequential composition. In this case, the process alphabet will be the union of the events that each sub-atomic process can perform.

In our semantics, each process has a scope represented by the store σ. This is valid if processes are defined using prefix operators, which preserve the same scope. However, processes which are defined using sequential composition are not. Therefore, we adapt the semantics of the sequential composition to allow the composition to preserve the same scope if it has atomic processes.

To allow variables to be remembered after the (;) the final state of the first process's store should be transferred to the second process's store. Hence, the two processes' stores can be merged. At this point, we should remark that according to Goldsmith *et al* [15], final states can be identified in CSP by following traces that end with terminal events. Only traces with the terminal event ($\surd$) lead to a final state of a process terminating successfully. If more than one final state is possible after some trace then the choice of which one is passed on is nondeterministic. If the number of possible states were infinite, then the *hiding* of termination by sequential composition could yield unbounded nondeterminism. Roscoe [32] explains this case further, pointing out that although p terminates in $(p; q)$, the overall process has not yet terminated. This means that it is treated in the same way as an event in A, where A is a set of events, hidden in $p \backslash A$. In CSP if this A is infinite, then the hiding operator is capable of introducing infinite branching (on τ's), or unbounded nondeterminism as it is often called (for more information on unbounded nondeterminism caused by hiding refer to [32]). Therefore, σ and σ' cannot be merged, because there is no guarantee that the processes p and q will have finite states to terminate on. However, note that an atomic process is a process that has a single termination state. Therefore, the concerns raised by Goldsmith *et al* [15] no longer hold and the two stores can be safely joined.

Therefore, sequential composition treats atomic processes (a) in a special way:

$$\frac{((a,\sigma),\rho) \xrightarrow{a} ((SKIP,\sigma'),\rho) \xrightarrow{\surd} ((STOP,\sigma'),\rho)}{((a\,;\,q,\sigma),\rho) \xrightarrow{a} ((SKIP,\sigma')\,;\,(q,\sigma),\rho) \xrightarrow{\tau} ((q,\sigma \oplus \sigma'),\rho)}$$

5 SoaCSP Properties

Herewith, we discuss two important features of SOA models: (i) how we avoid deadlocks; (ii) and how sessions terminate gracefully.

5.1 Deadlock avoidance

Deadlocks are representations of systems that cannot evolve further. In CSP and its extensions (including SoaCSP), deadlocks can be explicitly represented by using $STOP$ (nothing to do) as a verification point in systems. However, deadlocks can appear during the dynamic reduction of systems in cases such as: $a \rightarrow q \parallel_x a' \rightarrow s$ where $a, a' \in x \wedge a \neq a'$; $a \rightarrow q \parallel_x s$ where $a \in x \wedge s$ is a termination process; $a \parallel_x a'$ where $a, a' \in x \wedge a \neq a'$; or $a \parallel_x s$ where $a \in x \wedge s$ is a termination process.

Detection of workflow deadlocks can be done using CSP Stable-Failure semantics [18], where processes should provide information on actions that they refuse to evaluate along with actions that they can perform. Although the Stable-Failure semantics is proposed as future work in Section 8, here we prove that our new primitives do not introduce design-deadlocks and discuss the possibility of developing Stable-Failure semantics for our calculus.

In the sequel, we assume $Proc$ is the full space of SoaCSP processes.

Definition 2 (Termination process). if $p \in Proc$ we say p is a termination process if it can only evaluate terminal events $\in \Omega$ and then evolve to do nothing.

$Termination(p) \equiv ((p, \sigma), \rho) \xrightarrow{\omega} ((STOP, \sigma'), \rho)$ where $\omega \in \Omega$

Lemma 1 (SoaCSP Terminations). $\forall p \in Proc$, only ($SKIP$, $THROW$, $YIELD$, $CLOSE$ and $LISTEN$) and their symmetric compensable processes are considered terminations.

Proof. By examination of the calculus operational semantics, only the primitive processes ($SKIP$, $THROW$, $YIELD$, $CLOSE$ and $LISTEN$) can evaluate terminal events only then evolve to $STOP$ therefore only these processes are considered as terminations.

Theorem 1 (Progress). If $p \in Proc$ then $((p, \sigma), \rho)$ is either a termination process or there exist p', σ' and a such that $((p, \sigma), \rho) \xrightarrow{a} ((p', \sigma'), \rho)$.

$Progress(p, \sigma, \rho) \equiv \forall p, \sigma \Longrightarrow Termination(p) \vee (\exists p', \sigma', a.((p, \sigma), \rho) \xrightarrow{a} ((p', \sigma'), \rho))$

Proof. The proof of *Theorem 1* involves an induction on the inference rules of SoaCSP (presented in this paper and in [**?**]). We prove that the set of rules for each process shows that the process is either a termination or has a further step for each type of event it can evaluate.

- **Case**($SKIP, THROW, YIELD, CLOSE$, **and** $LISTEN$) according to *Lemma 1* these primitives processes are termination processes.
- **Case**(a **and** $a \rightarrow p$) where a is an atomic event (unit of work that can not fail). In the semantics, these processes are the only processes where event a can be written explicitly. τ (internal action that can not be controlled or blocked by the environment) and terminal events ($\in \Omega$) can not be used explicitly in the calculus therefore $a \in \Sigma \wedge a \notin \Sigma^{\tau\,\Omega}$.
  $a \notin \Omega$ therefore these processes are not terminations. For that, we should prove $\forall a \in \Sigma\ (\exists p', \sigma'.((a, \sigma), \rho) \xrightarrow{a} ((SKIP, \sigma'), \rho)) \wedge (\exists p', \sigma'.((a \rightarrow p, \sigma), \rho) \xrightarrow{a} ((p, \sigma'), \rho))$
  - **Case** If a is an ordinary event then this condition is hold by SoaCSP semantics axioms.
  - **Case** If a is a channel $(a\, dir\, MSG \wedge (MSG = n \vee (c)) \wedge (dir = . \vee !))$ then this condition is hold by SoaCSP semantics axioms.

  - **Case** If a is a channel $(a\,dir\,MSG \wedge (MSG = e) \wedge (dir = .\vee!))$ then the condition is hold if $e = n \vee \exists e', \sigma'.((e,\sigma),\rho) \xrightarrow{\tau} ((e',\sigma'),\rho))$ which is true according to the standard integer semantics.
  - **Case** If a is a channel $(a\,dir\,MSG \wedge (MSG = x \vee m) \wedge (dir = .\vee?))$ (where x represent integer variable and m represents channel variable) then this condition is hold by SoaCSP semantics axioms.
  - **Case** If a is a mobile channel $(m\,dir\,MSG \wedge (MSG = INPC \vee OUTC) \wedge (dir = .\vee?\vee!))$ (where m represents channel variable) then this condition is hold by SoaCSP semantics axioms.

- **Case** ($p \sqcap q$, $N = p$ and and their symmetric operators for compensable processes) by definition these process are not terminations therefore we should prove that $(\exists p', \sigma', a.((p,\sigma),\rho) \xrightarrow{a} ((p',\sigma'),\rho))$ which is hold by by SoaCSP semantics axioms.
- **Case** (If b Then p Else q, while b do p, and their symmetric operators for compensable processes) the progress condition is hold if $b = TRUE \vee b = FALSE$, if not the we should prove $\exists b', \sigma'.((b,\sigma),\rho) \xrightarrow{\tau} ((b',\sigma'),\rho))$ which is true according to the standard boolean semantics.
- **Case** ($N \Leftarrow p$ and $N \Rightarrow \{q\}$) are available as axioms in the semantics.
- **Case** ($p \,\Box\, q$ and and their symmetric operators for compensable processes) by definition this process is not a termination therefore we should prove that $(\exists p', \sigma', a.((p,\sigma),\rho) \xrightarrow{a} ((p',\sigma'),\rho))$. To prove that it is enough to prove that either p or q able to evaluate an action. If p or q able to evaluate terminal events then they are terminations. If they evaluate $(a \in \Sigma)$ then they of the form (a or $a \rightarrow p$) which has been proved that they always has further step by axioms. Finally, p or q can evaluate (τ) which can be initiated by the process any time and the environment can not block it. If $(a \in \{N\top, N\bot\})$ then they are invocation or definitions which has been proved that they always has further step by axioms.
- **Case** ($p \triangleright q$ and $p \blacktriangleright q$) by definition these processes are not terminations therefore we should prove that $(\exists p', \sigma', a.((p,\sigma),\rho) \xrightarrow{a} ((p',\sigma'),\rho))$. To prove that it is enough to prove that p is able to evaluate an action which can be proved in similar way as the previous case.
- **Case** ($p \div q$) by definition this process is not a termination therefore we should prove that $(\exists pp', \sigma', a.((pp,\sigma),\rho) \xrightarrow{a} ((pp',\sigma'),\rho))$. To prove that it is enough to prove that p is able to evaluate an action which can be proved in similar way as the previous case.
- **Case** ($p; q$, $p \backslash A$, $p[\![R]\!]$, $l : p$, $p \parallel_A q$, and their symmetric operators for compensable processes) can be proved in similar way as previous cases.
- **Case** ($[pp]$ and $pp \boxtimes qq$) can be proved by showing that pp is able to evaluate an action which can be proved from previous cases.

Our sessioning and termination algorithms introduce a way to classify events which allow more events to be available on the calculus, but it does not affect the calculus operators. Therefore, these algorithms do not affect the stable-failure semantics of CSP, and such semantics could be developed for the mobility-free fragment of SoaCSP. However, mobility has major effects on the refusal set of any process because it changes its alphabet. Capturing the effects of mobility on distinct synchronisation sets simplify refusal constructions (by using the same rules to update the dynamic synchronisation set of the parallel operator and to update refusals as well). However, this semantics will need more investigations.

5.2 Graceful Termination

In this section, we investigate the relationship between the session algorithm and the termination algorithm and how the termination algorithm will graceful terminate any open sessions if it will eventually terminate. Therefore, sessions that include services which are defined recursively or explicit $STOP$ which will never terminate will not be considered here.

cCSP adopts the mechanisms for successful terminations in CSP and extends it with mechanisms for unsuccessful terminations. In SoaCSP, we adopt cCSP termination mechanisms and extend cCSP processes to work under sessions.

Our aim is ensure that as we add labels to classify events we do not affect the evaluation of the original termination algorithms of cCSP. We ensure that on termination, session labels installed on creation are removed in termination with the respect to the session hierarchy.

Definition 2 (Terminating processes). We say p is a terminating process if it satisfies two conditions:

(i) If it is a termination or it is eventually reduced to a termination. suppose $p \xrightarrow{a} p'$ we say p is a terminating process if $a \in \Omega$ or $a \in \Sigma^{\tau}$ and p' is a terminating process. Terminating(p)= if $p \xrightarrow{a} p'$ then $a \in \Omega \vee (a \in \Sigma^{\tau} \wedge$ Terminating$(p'))$.

(ii) If it is a composition of terminating processes.

Lemma 2. If p is not recursively defined or $STOP$ then it is a terminating process.
$\forall p \in Proc . if p \neq STOP \wedge Terminating(p)$.

Proof. The proof of *lemma 2* involves a case analysis of event types and how they affect the different operators in SoaCSP. The proof shows that there are specific forms for basic processes which have been proved to be terminating, and all other processes are compositions of these basic processes.

According to CSP, processes are sequences of atomic actions. This definition of processes are adopted by all its extensions including SoaCSP.

In SoaCSP, atomic actions can be of four types:

1. processes normal actions which are represented by the universal set Σ.
2. the silent event τ.
3. terminal events which are represented by the universal set Ω.
4. Service definitions and invocations events which are represented by the universal set λ.

Normal actions Σ can be used in the calculus in two operators only:

1. Atomic processes which always evaluate the atomic action and then evolve to $SKIP$ by definition. Therefore, each atomic action has implicit termination $SKIP$ in its definition so any process definition ends with an atomic process then it evolves implicitly to the termination $SKIP$.
2. Atomic actions can also occurs in prefix operator, where the sequence of actions should end with a process. If this process is one of the primitive processes then it reduced to termination because all of them are terminations except $STOP$ and we exclude $STOP$ from our definition of terminated processes. If it is not a primitive process then it defined as sequence of action again end with process.

τ event can not appear in the definition of processes. It is used in the semantics to represents some of the reduction steps.

Terminal events $\in \Omega$ can appear in processes definitions by using their equivalents primitive processes which by definitions are terminations.

Any other form of processes definition should be a composition of the previously defined basic processes using the calculus different process, and by definition compositions of terminated processes are terminated processes.

Theorem 2 (Graceful termination). Let p and q be terminating processes then whenever a session label is installed by the session algorithm, the label will be removed in termination with the respect to the session hierarchy.

Proof. In session $(l : p \parallel l : q)$. Suppose $Terminating(p), Terminating(q)$ this implies $(l : Terminating(p) \parallel l : Terminating(q))$.

According to *lemma 2* terminating processes eventually reduce to terminations. This implies $(l : Termination(p') \parallel l : Termination(q'))$.

According to (term1) rule (Figure 3.2) $(l : Termination(p') \parallel l : Termination(q'))$ evolves to $(Termination(p'); l : CLOSE \parallel Termination(q'); l : CLOSE)$.

If terminations is $THROW$ then the session hierarchy under the transaction block where $THROW$ happens will be closed. According to the cancellation semantics of cCSP[8] the whole transaction after the compensation is equivalent to $SKIP$, which implies that after $THROW$ the system will evolve to $SKIP; l : CLOSE$.

If the termination is $YIELD$ then by (yield) definition it either has the effect of $THROW$ or $SKIP$ which means it will evolve to $SKIP; l : CLOSE$.

If the termination is $LISTEN$ then by (listen) definition it either has the affect of $l : CLOSE$ or $SKIP$ which means it will evolve to $SKIP; l : CLOSE$ or $l : CLOSE$.

According to CSP axioms [18] $SKIP; p$ reduces to p. Therefore $SKIP; l : CLOSE$ reduces to $l : CLOSE$.

According to (term2) rule, $l : CLOSE$ synchronised with other $l : CLOSE$ in the session. If no synchronisation available then level up $f_p(l) : CLOSE$ as (term3) stated. Note that $CLOSE$ only synchronises with other $CLOSE$ in the system as shown in the updated (syn-table).

If the $CLOSE$ reaches the high level then by definition f_p returns null which will be used by our (close) to evolve $CLOSE$ to $SKIP$.

6 SoaCSP at Work

To give a sense of how SoaCSP can be used, we present in this section a simple but non-trivial example. Herewith, we illustrate a case of a variable compensation, a mobile communication and invocations.

*Problem.*Customers can order products from a warehouse. A customer should identify the item number and the quantity (denoted nx and ny below). If the order is accepted then the order should be fulfilled and the quantity subtracted from the inventory. If the item is unavailable in the inventory, then this item should be ordered from the suppliers.

If this action completed but later the transaction failed then the deducted quantity should be returned to the inventory or the ordered item should be cancelled from the suppliers. The `FulfillOrder` process starts by booking a courier which should be compensated if the transaction failed by cancelling the courier. Booking a courier 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 (identified as ccn below) is charged with the amount needed. This is done at the same time with the other activities in the system, because it usually succeeds. However, if the credit process fails then the whole transaction fails and the system should be compensated.

System Model. We implement this example as an order transaction system residing in a server; it runs in parallel with different $\texttt{Customer}_i$ client services which issue different orders, and a `Bank` service which validates customers' credit cards.

System = ((*OrderTrans $\underset{A}{\|}$ Customer) $\underset{B}{\|}$ *Bank)
where A={Order} and B={CreditCheck, Ok, NotOk}

We define a customer as an ordinary service which invokes `orderTrans` service to order an item. Different customers can be available in the system at the same time. Each client customer will initiate a session to handle its order with `OrderTrans` system in the server.

Customer$_i$= (OrderTrans $\Leftarrow$ {Order!nx!ny})

The server order system is defined as a persistent service which accepts customers orders and fulfills them. For each order the system open a new transaction (using the transaction block operator).

OrderTrans $\Rightarrow$ [ProcessOrder]

Each order should be checked for stock availability in the inventory. If the stock is insufficient then order from suppliers. In this system, we assume the number of suppliers are 20. All the suppliers are the same, so the fastest one will fulfill the order. We use here mobile communication to establish a connection with one of the suppliers rather than invocations, because we do not know which supplier should be invoked. This decision is taken at run time. Alternatively, we can implement a supplier interface service which can be invoked by the inventory every time and the interface service will establish the mobile connections according to some predefined criteria.

ProcessOrder = (Order?x?y; (Inventory(x,y) $\underset{C}{\|}$ Supplier_i)) $\div$Z)
; FulfillOrder(x,y) where C={UnAvailable, ship, finish}

The process variable `Z` is used in the place of the compensation, because we do not know in prior if the order is available in the inventory or we should order it from suppliers.

Inventory(x,y)= (Available; deduct.x.y; Z:=Restock.x.y; SKIP)
$\Box$ (UnAvailable?(sup)$^+$; sup!x!y; ship.x.y; finish!(sup)$^-$; SKIP)

A mobile connection is established via the `UnAvailable` channel and released via the `finish` channel.

Supplier$_i$= (UnAvailable!(SID_i); SID_i?x?y; ship.x.y; Z:=CancelSup_i
; finish?(SID_i); SKIP)

Therefore, we replace the compensation with a variable at the design time, where the real process will be assigned during the run time.)

The order then should be packed and shipped.

FulfillOrder(x,y)= (BookCour$\div$CancelCour) $\|$ PackOrder(x,y) $\|$
((Bank $\Leftarrow$ {CreditCheck!ccn; ((Ok; SKIP)$\Box$(NotOk; THROW))})$\div$SKIP)
PackOrder(x,y) = $\|_{i=0}^{i=y}$ (Pack.x $\div$ Unpack.x)

$\|_{i=0}^{i=y}$ is an indexed version of the parallel operator between y processes.

In parallel, the current customer card should be charged by the bank if the bank approves the card then proceed with the system and terminate successfully; however if the bank rejects the card then terminate the system unsuccessfully by throwing an exception.

We define the Bank as a persistent service which can be invoked by other service to validate customers credit cards.

Bank ⇒ `CreditCheck?n ; (Ok ; Bank) ⊓ (NotOk ; Bank)`

Possible Execution Scenario. Due to the parallel operator in this system the execution may have different possibilities. Assume the following scenario: The item has been found in the inventory therefore the item quantity has been deducted from the inventory. Following that, 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 answer, a fourth item has been packed.

This scenario will lead us to the choice: `(Ok; SKIPP)` □ `(NotOk ; THROWW)` if the bank answered `Ok` then the transaction continues, however, if the bank answered `NotOk` then the current transaction terminates abnormally causing the following (unlabelled) compensations to start.

`(Unpack.x‖Unpack.x‖Unpack.x‖Unpack.x‖CancelCour); Restock.x.y`

7 Related Work

SOA calculi are sophisticated process calculi for specifying and analysing SOA systems. SOA calculi usually host explicit primitives for handling service definitions, invocations and bidirectional sessioning. Several proposals have been put forward, in the last decade, aiming at providing process calculi to support specification and analysis of services. Although these calculi in general host similar mechanisms for service definitions and invocations, these calculi are based on different mechanisms to encode service interactions.

In our version of CSP for SOA (SoaCSP), we adopt Hoare Labelling operator to work in place of sessions. Labels are used as fresh names which are attached dynamically to processes to construct sessions. This approach is similar to the name-scoping mechanism *à la* π-calculus used in calculi like [25,2,22,3], to dynamically structure the conversations between services into a hierarchy of sessions. Explicit session primitives have the advantage of facilitating direct analysis of sessions over correlations mechanisms used in calculi like [36,16,26], which make the analysis of sessions difficult.

cCSP failure and compensation mechanisms comparing to the other transactional process calculi for SOA, like [24,27], are very expressive, specially after our extension with dynamic recovery [20]. General dynamic recovery is encoded in SOCK [16] as well. However, apart from the complexity of the three layered design procedure of SOCK, SOCK designers have strictly followed the standards which raises the same issue of the interdependence between the three sub-calculus. The cCSP failure and compensation algorithm have been adopted by the Conversation Calculus [36,9] however the result was not compositional as stated in [23].

One of the significant features of CSP and cCSP is their explicit primitives to represents successful and unsuccessful terminations. This feature reveals the termination state of a process to its environment, which is well suited to services. Therefore, in SoaCSP, we retain this feature and design our sessioning algorithm to introduce the right scope for the different activities of a process. For instance, unsuccessful terminal events should propagate through the entire transaction and force processes to shutdown

even if they work under different sessions hierarchies as long as they are included in the same transaction block. However, the successful termination should propagate gradually through the current sessions hierarchy to ensure safe termination. The termination mechanism of SoaCSP also cleans the system from labels generated in invocations.

We design our termination algorithm to dynamically employ this policy without burdening designers with the overhead of maintaining session terminations. Moreover, we introduce a new primitive process which optionally allows processes to interrupt their execution if their session has been closed by their siblings.

SoaCSP disciplined termination of session hierarchies has the advantage of relieving the explicit handling of sessions closing, over the other expressive termination proposals like the one in [3].

SoaCSP inherits the multi-way communication feature from CSP. Hence, multi-party sessions are naturally permitted in the calculus, which has advantages over two-party sessions, like the SCC calculi family [2,22,3]. The new feature in SoaCSP is the dynamic expansion and delegation of sessions, achieved by labels sent along with the channel names in mobile communications. Although SoaCSP mobile sessions have the advantages of simplicity and familiarity, session expansions and merges in μse [5] are more disciplined. In μse explicit primitives are introduced for dynamic expansions and merges of sessions, which can be smoothly adopted in SoaCSP; however we made the decision to keep it simple as long as it offers the feature.

In SoaCSP, we use channel-oriented communications with transparent session names for scoping. Thus, channels should be used explicitly to manage communications. This might appear as an overhead designing issue comparing to the transparent channels and sessions names in calculi like the SCC family, where sessions names replace channels names in communications. However, we preferred to retain the basic communication model as defined in CSP.

Moreover, SoaCSP extra-session communication model is comparable to the expressive communication models of [36,5], and has the advantage over one direction out of session communication adopted by SCC calculi family [2,22,3].

Service compositions can be constructed using two models: *orchestration* where orchestrator services are used to control the workflow within the composition; *choreography* where protocols are distributed within the composition to control the collaboration between services to achieve the desired workflow.

While most of the calculi are proposed to deal with orchestrations in service compositions, the Global Calculus [10] captures the global view of the system in general then projects it to the individual services following the choreography approach of WS-CDL [38].

SoaCSP is mainly an orchestration calculus. However, the borders between the two design models are not firm, SoaCSP as other orchestration calculi can project service choreographies as explained and formalised in [6].

The Global calculus has the feature of implementing multi-party sessions however session participants are determined at the projection phase and are not allowed to dynamically expand as the approach of SoaCSP and μse.

Contracts calculi [4,11,12] have focused more in designing the systems with the emphasis on protocols compliance in the discovery phase. Protocol compliance is not considered in SoaCSP, however protocol compliance can be checked using the trace refinement of CSP. A trace semantics for SoaCSP is left for future work.

8 Conclusion and Future Work

Mobility, sessioning and service invocation are crucial features in any process calculus proposed for modelling SOA systems.

In this paper, we developed a version of CSP for SOA (SoaCSP), adopting Hoare Labelling operator to work in place of sessions; we introduced a new primitive for service invocation and we discussed sessions nesting, termination, and migration.

Additionally, we developed a notion of mobility for SoaCSP. The same algorithm can of course be used to introduce mobility in the standard CSP. The mobile version of SoaCSP, extends SoaCSP with channels and mobile channels. In addition, parallel composition has been modified to permit communication of channel names. SoaCSP also extends DEcCSP by allowing named processes to have parameters of type integer or named events.

We have developed an operational semantics for SoaCSP. 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 plan to introduce explicit primitives to identify priorities and establish connections under conditions. The development of a type system is also left for future work. Allowing communicated expressions to be from other types beside integer would be preferable too.

References

1. CSP: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers, ser. LNCS, vol. 3525. Springer (2005)
2. Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., Zavattaro, G.: Scc: A service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM. Lecture Notes in Computer Science, vol. 4184, pp. 38–57. Springer (2006)
3. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS. Lecture Notes in Computer Science, vol. 5051, pp. 19–38. Springer (2008)
4. Bravetti, M., Zavattaro, G.: A theory for strong service compliance. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION. Lecture Notes in Computer Science, vol. 4467, pp. 96–112. Springer (2007)
5. Bruni, R., Lanese, I., Melgratti, H.C., Tuosto, E.: Multiparty sessions in soc. In: Lea, D., Zavattaro, G. (eds.) COORDINATION. Lecture Notes in Computer Science, vol. 5052, pp. 67–82. Springer (2008)
6. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: In COORDINATION, volume 4038 of LNCS. pp. 63–81. Springer (2006)
7. Butler, M.J., Hoare, C.A.R., Ferreira, C.: A trace semantics for long-running transactions. In: 25 Years Communicating Sequential Processes [1], pp. 133–150
8. Butler, M.J., Hoare, C.A.R., Ferreira, C.: A trace semantics for long-running transactions. In: 25 Years CSP [1], pp. 133–150
9. Caires, L., Ferreira, C., Vieira, H.T.: A process calculus analysis of compensations. In: Kaklamanis, C., Nielson, F. (eds.) TGC. Lecture Notes in Computer Science, vol. 5474, pp. 87–103. Springer (2008)
10. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. 34(2), 8 (2012)

11. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for web services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM. Lecture Notes in Computer Science, vol. 4184, pp. 148–162. Springer (2006)
12. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. In: Necula, G.C., Wadler, P. (eds.) POPL. pp. 261–272. ACM (2008)
13. Chen, Z., Liu, Z.: An extended ccsp with stable failures semantics. In ICTAC 2010 pp. 121–136 (05/2010 2010)
14. Chen, Z., Liu, Z., Wang, J.: A theory of failure-divergence refinement for long running transactions. in FM 2011 pp. 262–277 (01/2011 2011)
15. Goldsmith, M., Roscoe, A., Scott, B.: Denotational semantics for occam 2. Tech. rep., Oxford University (1993)
16. Guidi, C., Lucchi, R., Gorrieri, R.: Sock: A calculus for service oriented computing. in Proc. ICSOC06, ser. LNCS, no. 4294. Springer p. 327338 (2006)
17. Hennessy, M.: A. Distributed Π-Calculus. Cambridge University Press (2007)
18. Hoare, C.: Communicating Sequential Processes. Prentice Hall (1985)
19. Hoare, T.: Why ever csp? Electr. Notes Theor. Comput. Sci. 162, 209–215 (2006)
20. Al Humaimeedy, A., Fernández, M.: General dynamic recovery for compensating csp. In Proc. of DCM2012 (2012)
21. Khaxar, M., Jalili, S., Khakpour M. S. Jokhio, N.: Monitoring safety properties of composite web services at runtime using csp. in proc EDOCW 2009 (2009)
22. Lanese, I., Vasconcelos, V., Martins, F., Ravara., A.: Disciplining Orchestration and Conversation in Service-Oriented Computing. in Proc. SEFM07. IEEE p. 305 314 (2007)
23. Lanese, I., Vaz, C., Ferreira, C.: On the expressive power of primitives for compensation handling. In: Proc. of ESOP 2010. Number 6012 in LNCS, Springer p. 366386 (2010)
24. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Proc. of FoSSaCS05. No 3441 in LNCS, Springer p. 282298 (2005)
25. Laneve, C., Padovani, L.: Smooth orchestrators. In: Aceto, L., Ingólfsdóttir, A. (eds.) FoSSaCS. Lecture Notes in Computer Science, vol. 3921, pp. 32–46. Springer (2006)
26. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. in Proc. ESOP07, ser. LNCS, no. 4421. Springer p. 3347 (2007)
27. Mazzara, M., Lanese, I.: Towards a unifying theory for web services composition. In: Proc. of WS-FM06. Number 4184 in LNCS, Springer p. 257272 (2006)
28. Miller, T., McBurney, P.: Multi-agent system specification using tcoz. In: Eymann, T., Klügl, F., Lamersdorf, W., Klusch, M., Huhns, M.N. (eds.) MATES. Lecture Notes in Computer Science, vol. 3550, pp. 216–221. Springer (2005)
29. Miller, T., McBurney, P.: Annotation and matching of first-class agent interaction protocols. In: Padgham, L., Parkes, D.C., Müller, J.P., Parsons, S. (eds.) AAMAS (2). pp. 805–812. IFAAMAS (2008)
30. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, part I & part II. Information and Computation 100 no 1, 1–77 (1992)
31. OASIS: Web services business process execution language version 2.0 oasis standard 11 april 2007. OASIS WSBPEL TC pp. http://docs.oasis–open.org/wsbpel/2.0/OS/wsbpel–v2.0–OS.html (2007)
32. Roscoe, A.: The Theory and Practice of Concurrency. Prentice-Hall (Pearson) (1997 revised 2005)
33. Roscoe, A.: On the expressiveness of CSP. Draft of OCT 23,2008 (2008)
34. Roscoe, A.: Understanding Concurrent Systems. Springer (2010)

35. Vajar, B., Schneider, S., Treharne, H.: Mobile CSP | B. In Proc. AVOCS 2009 (2009)
36. Vieira, H., Cairesand J. Seco., L.: The conversation calculus: A model of service-oriented computation. In: Proc. of ESOP08. No 4960 in LNCS, Springer p. 269283 (2008)
37. W3C: Web service choreography interface (WSCI) 1.0 w3c note 8 august 2002 p. http://www.w3.org/TR/wsci/ (2002)
38. W3C: Web services choreography description language version 1.0 W3C candidate recommendation 9 november 2005 pp. http://www.w3.org/TR/2005/CR–ws–cdl–10–20051109/ (2005)
39. W3C: Web services description language (WSDL) version 2.0 part 1: Core language W3C recommendation 26 june 2007 pp. http://www.w3.org/TR/2007/REC–wsdl20–20070626/ (2007)
40. Welch, P., Barnes, F.: A CSP Model for Mobile Channels. Communicating Process Architectures, IOS Press (2008)
41. Yeung, W.: Csp-based verification for web service orchestration and choreography. Simulation (1): 65-74 (2007 83 no 1, 65–74 (2007)