The Journal of Logic and Algebraic Programming 81 (2012) 585–605

Contents lists available at SciVerse ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

On the analysis of compensation correctness

Cátia Vaz [a,b,*], Carla Ferreira [c]

[a] DEETC, ISEL, Instituto Politécnico de Lisboa, Portugal
[b] Instituto de Engenharia de Sistemas e Computadores, I&D em Lisboa, Portugal
[c] CITI/Departamento de Informática, FCT, Universidade Nova de Lisboa, Portugal

ARTICLE INFO

Article history:
Received 19 April 2010
Revised 31 December 2011
Accepted 10 April 2012
Available online 3 May 2012

Keywords:
Long running transactions
Compensations
Compensation correctness
Failure handling
Process calculi
Model checking

ABSTRACT

One fundamental idea of service-oriented computing is that applications should be developed by composing already available services. Due to the long running nature of service interactions, a main challenge in service composition is ensuring correctness of transaction recovery. In this paper, we use a process calculus suitable for modelling long running transactions with a recovery mechanism based on compensations. Within this setting, we discuss and formally state *correctness criteria* for compensable processes compositions, assuming that each process is correct with respect to transaction recovery. Under our theory, we formally interpret *self-healing* compositions, that can detect and recover from faults, as correct compositions of compensable processes. Moreover, we develop an automated verification approach and we apply it to an illustrative case study.

1. Introduction

Service-oriented computing is a well known paradigm for creating new services by composing available ones, usually in distributed and heterogeneous environments. This paradigm is particularly suited for describing loosely coupled systems, *i.e.*, systems composed by interacting parts that exchange most information through messages (shared information is minimal). Services are described using appropriate service description languages, published and discovered accordingly to predefined protocols and combined using engines that coordinate the interaction among collaborating services. Additionally, in these systems, transactions may last long periods of time. Thus, contrary to traditional ACID transactions, solutions based on locking are not feasible. Long running transactions and recovery mechanisms based on compensations are used instead.

Web services appear naturally as a service oriented based technology. Usually the Internet is the preferred communication medium and they rely on Internet-based open standards, including the Simple Object Access Protocol (SOAP) [36] for transmitting data and the Web Services Description Language (WSDL) [27] for defining services. In this context, dynamic Web services interactions may occur, leading to the automation of business process within and across enterprises for application and business-to-business integration. Business process integration in real business scenarios involve long running interactions, transactions management and are often driven by a work flow engine that executes a specified business process model to automate the information flow and business operations. This requires Web services composition, that allows new services to be created by composing available ones. The most important Web service composition languages in the past have been IBM WSFL [17] and Microsoft XLANG [33]. These two have then converged into the Web Services Business Process Execution Language (WS-BPEL) [28]. WS-BPEL allows the definition of work flow based composition of services, defining composition from an *orchestration* perspective, *i.e.*, the description of how services interact with each other at message level is from the perspective and is under the control of a main coordinator. This language is the one that best represents the

* Corresponding author at: DEETC, ISEL, Instituto Politécnico de Lisboa, Portugal.
E-mail address: cvaz@cc.isel.ipl.pt (C. Vaz).

1567-8326/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jlap.2012.04.002

orchestration perspective. WS-BPEL can also be generated to some extent from BPMN [29] models, a graphical notation for business process modelling that also allows us to define work flows based on service composition. Another proposal, which also allows the definition of work flow based composition of services, is Web Services Choreography Description Language (WS-CDL) [37]. This language follows a *choreography* approach, *i.e.*, specifies the conversation that should be undertaken by each participant, describing the external visible behaviour of services as a set of message exchanges, not assuming the existence of a coordinator. This approach implies that the overall conversation occurs as in a peer-to-peer composition of collaborating services.

A main challenge in service composition is ensuring the correctness of transaction recovery. In particular, because of the long running nature of transactions, techniques based on locking or rollback are not feasible since parts of a transaction can be impossible to undo. Nowadays, in many work flow engines, such as in WS-BPEL execution engines, compensations are used as activities programmed to recover full or partial executions of transactions, bringing the system again to consistency. Thus, the problem becomes ensuring compensation correctness. Moreover, a notion of compensation correctness is needed for interaction based systems, namely a notion that takes into account the specificities of transaction recovery for a given application context.

In this paper, we introduce a notion of compensation correctness over a process calculus suitable for modelling long running transactions, with a recovery mechanism based on compensations. In this setting, we discuss *correctness criteria* for compensable process compositions, assuming that each process is correct with respect to transaction recovery. A compensation is said to be correct if its execution has the expected behaviour in the sense that it restores the consistency of the transaction. Since the expected behaviour depends on the application context, our model expects a correctness map provided by the programmer, expressing how meaningful interactions can be compensated. Hence, the programmer must provide a set of possible sequences of interactions that compensate each meaningful interaction. Notice that this approach is more general than the insurance by the programmer for cancelling or reversing each action made [5,6,8,9]. In some applications, ensuring transaction consistency will in fact mean that the execution of compensations will revert the effect of each action done before fault signalling. But in other cases, since some actions cannot be cancelled, the programmer will only be interested in approximating the effect of cancellation, bringing the system back to consistency.

One of the main contributions of our work is the insurance that the composition of correct compensable processes is also a correct compensable process, under reasonable correctness criteria. Our correctness criteria are *stateless*, *i.e.*, we assume that all information needed is exchanged through messages. This is known as contextualisation and exchanged messages describe the state of the overall system. The developed theory also provides interesting insights on an important issue in the service oriented approach, namely the reliable automated composition of distributed services. In particular, one important challenge is the *self-healing* composition of services [30], *i.e.*, compositions that automatically detect that some service composition requirements are no longer satisfied and react recovering from inconsistencies. Thus, self-healing implies that when a fault occurs, the system should automatically recover, bringing it back to consistency. Within this setting, we will formally interpret self-healing compositions, relating this concept with correct compositions of compensable processes.

In the paper realm, we have chosen to build the calculus upon the core of the asynchronous π-calculus, with a notion of transaction scope and other primitives to allow dynamic recovery based on compensations. We have chosen the asynchronous polyadic π-calculus based on the fact that most of service oriented architectures, and in particular those involving long running transactions, can be seen as interactive components communicating asynchronously within a distributed system. In our calculus, when transactions abort, one must know to which interaction context they belong to, *i.e.*, the underlying interaction session. Since this paper aims at tackling the problem of consistency of compensable transactions, we only focus on the interaction among transactions occurring within sessions, not on primitives such as service definition and instantiation. However, the calculus can be extended to include these kind of primitives, in a similar way of the work of Honda et al. [16].

Another contribution is an automated verification approach for our theory. We applied model checking techniques to automatically verify the correctness of compensable processes. Thus we have used the ProB Model Checker [23], which allows to model check system specifications and requires only the interpretation of the labelled transition system (LTS) and of the initial state of the system as a set of predefined Prolog [3] predicates. Therefore, we have developed a Prolog interpreter for our calculus, including a parser and the encoding of LTS rules. Here we discuss the ideas behind this interpreter and how we can use it together with ProB to verify the correctness of compensable processes compositions. Furthermore, we discuss the scalability of our approach.

This is an extended version of a preliminary work already published [34], where correctness criteria were presented. All the results about the automated approach are original to this contribution.

2. A compensating calculus

In order to reason about correctness criteria for compensable transactions, we propose a *compensating calculus* for modelling long running transactions with a recovery mechanism based on compensations. The calculus is inspired on $\mathrm{dc}\pi$-calculus [35] and on the calculus presented in [19], but is focused on the relevant primitives for reasoning about the correctness of compensations.

The underlying model of the calculus is the asynchronous polyadic π-calculus, based on the fact that long running transactions can be seen as the evolution of interacting components, communicating asynchronously, within a concurrent

$$
\begin{array}{lcll}
P, Q & ::= & \mathbf{0} & \text{(Inaction)} \\
 & | & \overline{a}\langle\tilde{v}\rangle & \text{(Output)} \\
 & | & \overline{t} & \text{(Failure)} \\
 & | & X & \text{(Process variable)} \\
 & | & \sum_{i\in I} a_i(\tilde{x}_i)\lfloor\lambda X_i.Q_i\rfloor.P_i & \text{(Input guarded choice)} \\
 & | & (P \mid Q) & \text{(Parallel composition)} \\
 & | & (\nu\, x)\ P & \text{(Restriction)} \\
 & | & t[P, Q]_r & \text{(Transaction scope)} \\
 & | & \langle P\rangle_r & \text{(Protected block)}
\end{array}
$$

Fig. 1. The syntax of processes.

setting. The language recovery mechanism allows us to dynamically update the compensations within transactions based on occurring interactions. To achieve that, we associate a process to each act of receiving that defines the compensation to be stored upon message reception. We have chosen not to associate compensations with the sender of the message since, in an asynchronous context, there are limited guarantees about the state of the receiver. The calculus allows for nested transactions, but only if they are within different interaction sessions – see ahead. Fault handling occurs in a nested way, *i.e.*, while the abortion of a transaction is silent to its parent, it causes the abortion of all proper subtransactions and the activation of compensations installed either by the transaction or by its subtransactions.

In case of transaction abortion, all stored compensations of the transaction are activated and protected against external faults. A transaction $t[P, Q]_r$ occurs within session r and behaves as process P until a fault is notified by an output $\overline{t}$ on the name t of the transaction. The transaction name t unequivocally identifies the transaction unit. Note that we must know the context session r for each transaction, since our correctness notion relies on the analysis of the communicated names within each session.

Notice that the proposed calculus is very flexible, allowing several levels of granularity with respect to transaction scope modelling. In our model, the system designer can choose from defining each two compensable transactions within a different interaction session, to define all the compensable transactions within a unique session.

2.1. Syntax

The syntax of our language relies on: a countable set of *channel names* $\mathcal{N}$, ranged over by $a, b, x, y, v, w, a_1, b_1, x_1, y_1, v_1, w_1 \ldots$; a countable set of *transaction identifiers* $\mathcal{T}$, ranged over by $t, u, t_1, u_1, \ldots$; a countable set of *session names* $\mathcal{S}$, ranged over by $s, r, s_1, r_1, \ldots$; and natural numbers, ranged over by $i, j, k, i_1, j_1, k_1, \ldots$ The sets $\mathcal{N}$, $\mathcal{T}$ and $\mathcal{S}$ are disjoint and identifiers $z, z_1, \ldots$ are used to refer to elements of both sets $\mathcal{N}$ and $\mathcal{T}$ when there is no need to distinguish them. The tuple $\tilde{v}$ denotes a sequence $v_1 \cdots v_n$ of such identifiers, for some $n \in \mathbb{N}$, and $\{\tilde{v}\}$ denotes the set of elements of that sequence.

Definition 2.1. The grammar in Fig. 1 defines the syntax of processes.

The calculus includes the core of asynchronous π-calculus processes [26], namely inaction, output, parallel composition and scope restriction. The *transaction scope* $t[P, Q]_r$ is a new primitive that within session r behaves as process P until an error is notified by an output $\overline{t}$ on the transaction name t. In case of fault, P is killed and the compensation Q is activated. The fault signal $\overline{t}$, that sends a fault message to a transaction identified by t, may come both from the internal process P or from an external process. The *protected block* $\langle P\rangle_r$, that behaves as P within session r, cannot be interrupted by external nested faults.

The process P in transaction scope $t[P, Q]_r$ can update the compensation Q, where the compensation update is performed by input prefixes. As said before, we have chosen not to associate compensations with the message sender, since in an asynchronous context there are limited guarantees about the state of the receiver. A compensation update takes the form of a function $\lambda X.Q'$, where the process variable X can occur inside process Q'. Applying such a compensation update to the compensation Q produces a new compensation $Q'\{Q/X\}$. Note that Q may not occur in the resulting compensation, or it may occur more than once. Thus, the form of input prefix is $a(\tilde{x})\lfloor\lambda X.Q'\rfloor.P$, which upon reception of message $\overline{a}\langle\tilde{v}\rangle$ updates the compensation with $\lambda X.Q'\{\tilde{v}/\tilde{x}\}$ and behaves as $P\{\tilde{v}/\tilde{x}\}$.

The compensation mechanism of the calculus allows for both dynamic and static definition of compensations. Note that, if all compensation updates have the form $\lambda X.X$, then the compensation is never changed. We will use id to denote the identity function $\lambda X.X$. The prefix $a(x).P$ can thus be seen as a shortcut for $a(x)\lfloor\mathsf{id}\rfloor.P$.

In this paper we do not consider primitives for session and transaction initiation, as this would introduce a higher level of complexity and take us away from our objective, which is reasoning about correctness criteria for compensable transactions. As mentioned before, our calculus can although be extended to include these kind of primitives, following the approach proposed by Honda et al. [16]. Such primitives allow us to initiate new sessions and ensure the generation of fresh names, such as fresh transaction names. Note that this is essential to have the input guarded replication primitive in our calculus and, since we do not have primitives for session and transaction initiation, we do not add also input guarded replication to

$$
\begin{aligned}
\mathsf{nl}(n, \mathbf{0}) &= \emptyset \\
\mathsf{nl}(n, \overline{t}) &= \emptyset \\
\mathsf{nl}(n, \overline{a}\langle \tilde{v} \rangle) &= \begin{cases} \{a\} \cup \{\tilde{v}\} & \text{if } n = 0 \\ \emptyset & \text{if } n \neq 0 \end{cases} \\
\mathsf{nl}(n, X) &= \emptyset \\
\mathsf{nl}(n, \langle P \rangle_r) &= \begin{cases} \mathsf{nl}(n-1, P) & \text{if } n \neq 0 \\ \emptyset & \text{if } n = 0 \end{cases} \\
\mathsf{nl}(n, t\,[P, Q]_r) &= \begin{cases} \mathsf{nl}(n, P) \cup \mathsf{nl}(n-1, Q) & \text{if } n \neq 0 \\ \mathsf{nl}(n, P) & \text{if } n = 0 \end{cases} \\
\mathsf{nl}(n, P \mid Q) &= \mathsf{nl}(n, P) \cup \mathsf{nl}(n, Q) \\
\mathsf{nl}(n, (\nu\, x)\, P) &= \mathsf{nl}(n, P) \setminus \{x\} \\
\mathsf{nl}(n, \sum_{i \in I} a_i(\tilde{x}_i)\lfloor \lambda Y_i.Q_i \rfloor.P_i) &= \begin{cases} \bigcup_{i \in I} (\{a_i\} \cup \mathsf{nl}(n, P_i)) & \text{if } n = 0 \\ \bigcup_{i \in I} (\mathsf{nl}(n, P_i) \cup \mathsf{nl}(n-1, Q_i)) & \text{if } n \neq 0 \end{cases}
\end{aligned}
$$

Fig. 2. Channel names of P at level n.

our calculus. Note that, alternatively, guarded replication may also be included by adding a well formed property as used in the work of Lucchi and Mazzara [24], expressing that received names cannot be used as subjects of inputs or of replicated inputs. It is our belief that the first approach is preferable since it allows the creation of fresh interaction sessions and fresh transaction names. Nevertheless, in both approaches, we must ensure that, after each session initiation, the session name is unequivocally and persistently identified. This is a requirement for our notion of correctness, which we will discuss later on.

In the following, we denote the channel names, the session names and the transaction names of a process P as $\mathsf{cn}(P)$, $\mathsf{sn}(P)$ and $\mathsf{tn}(P)$, respectively. The set of names of P, denoted by $\mathsf{n}(P)$, is the union of these three sets.

With respect to bindings, the names in $\{\tilde{x}\}$ and the name x are bound in $a_i(\tilde{x}_i)\lfloor \lambda X_i.Q_i \rfloor.P_i$ and in $(\nu\, x)\, P$, respectively. The other names are free. Furthermore, we use the standard notions of free names of processes. We write $\mathsf{bn}(P)$ (respectively $\mathsf{fn}(P)$) for the set of names that are bound (respectively free) in a process P. Bound names can be α-converted as usual. Also, the variable X is bound in $\lambda X.Q$. We consider only processes with no free variables as processes. As usual, the term $(\nu\, \tilde{x})\, P$ abbreviates $(\nu\, x_1)\ \ldots\ (\nu\, x_n)\, P$, for some $n \geq 0$.

2.2. Well-formedness

For simplicity of the correctness *criteria*, we introduce a *well-formedness criteria* to rule out some wrong process designs. To this aim, we first introduce some terminology.

A *context* is a process term $C[\![\bullet]\!]$ which is obtained by replacing in a process an occurrence of $\mathbf{0}$ with a placeholder $\bullet$. Process $C[\![P]\!]$ is obtained by replacing $\bullet$ with P within $C[\![\bullet]\!]$. The notion of context can be generalised to n-hole contexts as expected. In particular, generic 2-hole contexts will be denoted by $C[\![\bullet_1, \bullet_2]\!]$, with $C[\![P, Q]\!]$ defined as the process obtained by replacing $\bullet_1$ with P and $\bullet_2$ with Q.

Definition 2.2. A *session context* $C[\![\bullet]\!]$ is a context such that the hole $\bullet$ occurs within a transaction scope or within a protected block. If the hole $\bullet$ occurs within a transaction scope or a protected block of a session r, the session context is denoted by C_r.

Definition 2.3. Let P be a process and $r, r' \in \mathsf{sn}(P)$ two interaction session names. We write $r \prec_P r'$ if there are two session contexts C_r and $C'_{r'}$ such that $P = C_r[\![C'_{r'}[\![Q]\!]]\!]$, for some process Q.

Function nl assigns to a natural number n and a process P the channel names that occur at level n in P. Differentiation of names by levels is required to ensure compositional correctness. The goal is the separation, under arbitrary nesting, of normal flow messages from compensation flow messages.

Definition 2.4. The function $\mathsf{nl} : \mathbb{N} \times \mathcal{P} \longrightarrow 2^{\mathcal{N}}$, which gives the set of free channel names of a process P occurring at level n, is defined in Fig. 2.

The need for this kind of differentiation by level is also pointed out by Carbone et al. [10]. However, in contrast to our calculus where compensations are compensable, their exception handlers never fail.

We define now well-formed processes.

$$
\begin{aligned}
\mathsf{extr}(\mathbf{0}) &= \mathbf{0}\\
\mathsf{extr}(\overline{t}) &= \mathbf{0}\\
\mathsf{extr}(\overline{a}\langle\tilde{v}\rangle) &= \mathbf{0}\\
\mathsf{extr}\Big(\sum_{i\in I} a_i(\tilde{x}_i)\lfloor\lambda Y_i.Q_i\rfloor.P_i\Big) &= \mathbf{0}\\
\mathsf{extr}(\langle P\rangle_r) &= \langle P\rangle_r\\
\mathsf{extr}(t\,[P,Q]_r) &= \mathsf{extr}(P) \mid \langle Q\rangle_r\\
\mathsf{extr}(P \mid Q) &= \mathsf{extr}(P) \mid \mathsf{extr}(Q)\\
\mathsf{extr}((\nu\, x)\, P) &= (\nu\, x)\, \mathsf{extr}(P)
\end{aligned}
$$

Fig. 3. Extraction function with nested abortion.

Definition 2.5. A compensable process P is *well formed* if the following conditions hold:

1. Transaction names are distinct. Different transactions cannot share the same activation name and each fault message is able to interrupt only a single transaction.
2. Communications outside sessions or among distinct sessions are not allowed to install compensations. If two transactions belong to different sessions, their communications cannot be compensated. Also, if a process does not occur within a transaction scope, it cannot install compensations (*i.e.*, compensation updates are id).
3. Relation $\prec_P$ is acyclic for all $s \in \mathsf{sn}(P)$ (that is, $\prec_P^+$ is irreflexive).
4. There is no interaction between channel names at different levels, *i.e.*, $a \in \mathsf{nl}(n, P) \Rightarrow \forall_{m\neq n}\, a \notin \mathsf{nl}(m, P)$.
5. All bound names are pairwise distinct and disjoint from the set of free names.

The first property is needed to avoid ambiguity on scope names. The uniqueness of the transaction identifier is an important feature to guarantee that upon a transaction abortion the correct compensation is activated [35]. The second and third properties are for simplicity of the correctness criteria. Namely, the third property is to avoid processes like $t[k[P, Q]_r, S]_r$ and $t[k[P, Q]_{r'}, S]_r \mid t'[k'[P', Q']_r, S']_{r'}$, where a session name is used by both a transaction and any of its inner transactions. The purpose of using levels, namely not allowing interaction between channel names in different levels, is to rule out the communication between the normal flow and the compensation flow of a process, as mentioned before. Since we allow nesting of compensable transactions, the same idea has to be applied to all levels. The last property is for simplicity of correct compensable processes definition.

2.3. Operational semantics

The dynamic behaviour of processes is defined by a labelled transition system which takes into account transaction scope behaviour. Upon transaction interruption, stored compensations must be activated while preserving all inner protected blocks. Therefore, one has to extract the stored compensations and place them in a protected block, and also preserve already existing protected blocks. The extraction is done by the function extr.

Definition 2.6. The function extr is defined in Fig. 3.

The definition of function extr considers a *nested abortion* approach, *i.e.*, when a parent transaction is aborted, all its subtransactions have to be aborted. This is, for instance, the approach of WS-BPEL. Notice also that, if the compensation is defined for a transaction scope within session r, the abortion of the transaction will place the corresponding compensation into a protected block also within session r.

Before presenting the rules of the labelled transition system, we first introduce the transition labels.

Definition 2.7. The syntax of transition labels α is defined in Fig. 4.

Notice that, in internal moves, we keep track of the names that are used as subject of a communication. Also, we keep the session names where the interaction has occurred. If the interaction occurs within different sessions, we keep the name of the session that has received the message. These extra information is necessary for defining correct compensable processes. We use $\perp$ whenever an interaction occurs outside a session.

The operational semantics of the language is given in terms of a labelled transition system (LTS) $(\mathcal{P}, \mathcal{L}, \xrightarrow{\alpha})$, where $\mathcal{P}$ is the set of processes, $\mathcal{L}$ is the set of labels over the set of all names $(\mathcal{N} \cup \mathcal{S} \cup \mathcal{T},)$ and $\xrightarrow{\alpha} \subseteq \mathcal{P} \times \mathcal{P}$ for each $\alpha \in \mathcal{L}$ is a transition relation.

Definition 2.8. The operational semantics of compensable processes is the minimum LTS closed under the rules in Fig. 5 (symmetric rules are considered for rules L-PAR and L-COM).

$$
\begin{array}{lcll}
\alpha & ::= & \textit{label tuples} & \alpha_i \quad ::= \quad \textit{input} \\
& | & (r, \alpha_i, \alpha_c) & \quad\quad | \quad t \\
& | & (r, \alpha_o) & \quad\quad | \quad a(\tilde{v}) \\
& & & \quad\quad | \quad \tau(z) \\
\\
\alpha_c & ::= & \textit{compensation} & \alpha_o \quad ::= \quad \textit{output} \\
& | & \lambda X.R & \quad\quad | \quad (\tilde{w})\overline{a}\langle\tilde{v}\rangle \\
& | & (\tilde{w})\lambda X.R & \quad\quad | \quad \overline{t} \\
& & & \quad\quad | \quad \overline{a}\langle\tilde{v}\rangle
\end{array}
$$

Fig. 4. The syntax of transition labels.

$$
\frac{}{\overline{a}\langle\tilde{v}\rangle \xrightarrow{(\bot,\overline{a}\langle\tilde{v}\rangle)} \mathbf{0}}\ (\text{L-OUT}) \qquad \frac{j \in I}{\sum_{i\in I} a_i(\tilde{x}_i)\lfloor\lambda X_i.R_i\rfloor.P_i \xrightarrow{(\bot,a_j(\tilde{v}),\lambda X_j.R_j\{\tilde{v}/\tilde{x}_j\})} P_j\{\tilde{v}/\tilde{x}_j\}}\ (\text{L-INP})
$$

$$
\frac{P \xrightarrow{(r,z(\tilde{v}),(\tilde{w})\lambda X.R)} P' \qquad Q \xrightarrow{(s,(\tilde{y})\overline{z}\langle\tilde{v}\rangle)} Q' \qquad \{\tilde{y}\} \cap \mathsf{fn}(P) = \{\tilde{w}\} \cap \mathsf{fn}(Q) = \emptyset \qquad r \neq s \Rightarrow R = X}{P \mid Q \xrightarrow{(r,\tau(z),(\tilde{w})\lambda X.R)} (\nu\,\tilde{y})\,(P' \mid Q')}\ (\text{L-COM})
$$

$$
\frac{P \xrightarrow{\alpha} P' \qquad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ (\text{L-PAR}) \qquad \frac{P \xrightarrow{(r,(\tilde{w})\overline{x}\langle\tilde{v}\rangle)} P' \qquad y \neq x \qquad y \in \{\tilde{v}\} \setminus \{\tilde{w}\}}{(\nu\,y)\,P \xrightarrow{(r,(y\tilde{w})\overline{x}\langle\tilde{v}\rangle)} P'}\ (\text{L-OPEN})
$$

$$
\frac{P \xrightarrow{(r,\alpha_i,(\tilde{w})\lambda X.R)} P' \qquad (\alpha_i = a(\tilde{v}) \Rightarrow y \notin \mathsf{n}(\alpha_i)) \qquad y \in \mathsf{fn}(R) \setminus \{\tilde{w}\}}{(\nu\,y)\,P \xrightarrow{(r,\alpha_i,(y\tilde{w})\lambda X.R)} P'}\ (\text{L-OPEN2}) \qquad \frac{P \xrightarrow{\alpha} P' \qquad x \notin \mathsf{n}(\alpha)}{(\nu\,x)\,P \xrightarrow{\alpha} (\nu\,x)\,P'}\ (\text{L-RES})
$$

$$
\frac{}{\overline{t} \xrightarrow{(\bot,\overline{t})} \mathbf{0}}\ (\text{L-FAIL}) \qquad \frac{P \xrightarrow{(\bot,(\tilde{w})\overline{a}\langle\tilde{v}\rangle)} P' \qquad \{\tilde{w}\} \cap (\mathsf{fn}(Q) \cup \{t\}) = \emptyset}{t[P,Q]_s \xrightarrow{(s,(\tilde{w})\overline{a}\langle\tilde{v}\rangle)} t[P',Q]_s}\ (\text{L-SCOPE-OUT})
$$

$$
\frac{P \xrightarrow{(\bot,\alpha_i,(\tilde{y})\lambda X.R)} P' \qquad \{\tilde{y}\} \cap (\mathsf{fn}(Q) \cup \{t\}) = \emptyset}{t[P,Q]_s \xrightarrow{(s,\alpha_i,\mathsf{id})} (\nu\,\tilde{y})\,t[P', R\{Q/X\}]_s}\ (\text{L-SCOPE-IN}) \qquad \frac{}{t[P,Q]_s \xrightarrow{(s,t,\mathsf{id})} \mathsf{extr}(P) \mid \langle Q\rangle_s}\ (\text{L-RECOVER-OUT})
$$

$$
\frac{P \xrightarrow{\alpha} P' \qquad \alpha \in \{(s,\alpha_o),(s,\alpha_i,\alpha_c)\} \qquad s \neq \bot}{t[P,Q]_r \xrightarrow{\alpha} t[P',R]_r}\ (\text{L-SCOPE}) \qquad \frac{P \xrightarrow{(r,\overline{t})} P'}{t[P,Q]_s \xrightarrow{(s,\tau(t),\mathsf{id})} \mathsf{extr}(P') \mid \langle Q\rangle_s}\ (\text{L-RECOVER-IN})
$$

$$
\frac{P \xrightarrow{\alpha} P' \qquad \alpha \in \{(s,\alpha_o),(s,\alpha_i,\alpha_c)\} \qquad s \neq \bot}{\langle P\rangle_r \xrightarrow{\alpha} \langle P'\rangle_r}\ (\text{L-BLOCK}) \qquad \frac{P \xrightarrow{\alpha} P' \qquad \alpha \in \{(\bot,\alpha_o),(\bot,\alpha_i,\alpha_c)\}}{\langle P\rangle_r \xrightarrow{\alpha'} \langle P'\rangle_r \qquad \alpha' = \alpha\{r/\bot\}}\ (\text{L-BLOCK-S})
$$

Fig. 5. LTS for compensable processes. In rule L-COM, whenever z is a transaction identifier, $\tilde{v}$ is the empty tuple.

Rule L-OUT sends a message and rule L-FAIL sends a fault message. Rule L-INP executes an input-guarded choice. Note that the substitution of the received name is applied both to the continuation and to the compensation to be installed. Note also that labels for inputs (and internal moves) are composed by an extra part, the update to the compensation. Rule L-PAR allows one of the components of parallel composition to progress. Rule L-COM allows communication, and propagates the

Scope extension laws:

$$\begin{array}{rcll}
(\nu\, x)\, \mathbf{0} & \equiv & \mathbf{0} & \\
(\nu\, x)\, (\nu\, y)\, P & \equiv & (\nu\, y)\, (\nu\, x)\, P & \\
\langle (\nu\, x)\, P \rangle_r & \equiv & (\nu\, x)\, \langle P \rangle_r & \\
Q \mid (\nu\, x)\, P & \equiv & (\nu\, x)\, (Q \mid P) & \text{if } x \notin \mathsf{fn}(Q) \\
t\, [(\nu\, x)\, P, Q]_r & \equiv & (\nu\, x)\, t\, [P, Q]_r & \text{if } x \notin \mathsf{fn}(Q)
\end{array}$$

Protected block laws:

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

Fig. 6. Structural congruence relation.

compensation update coming from the input. As already said, we keep track of the names that are used as subject. Notice that if the communication does not occur within the same interaction session, the compensation update can only be the identity id. This feature is ensured by condition $r \neq s \Rightarrow R = X$ in the rule L-COM, *i.e.*, there is no real installation of compensations. If we allowed installation between different sessions, we would need more information about the interaction. For instance, we would need information about the communicated names and both the sessions where the message was received and where the message was sent. Such additional information would be required to analyse the correctness of both sessions, making the setting much more complex and requiring session merging. Note also that rule L-COM also handles synchronisation on transaction identifiers for external failures, where communication occurs between two different sessions and, thus, the compensation update is always the identity id. Rule L-RES is the classic rule for restriction. Note that session names are not restricted. Rule L-OPEN allows to extrude bound names. Rule L-OPEN2 allows to extrude names occurring in the compensation update.

Transaction interruptions are modelled by rules L-RECOVER-OUT and L-RECOVER-IN. Rule L-RECOVER-OUT allows external processes to kill a transaction via a fault signal $\bar{t}$. Notice that the remaining process is composed by two parts: the first one extracted from P, and the second one corresponding to compensation Q, which will be executed inside a protected block. Note that we could allow programmers to ensure compensation protection by themselves, but that would be difficult given the proposed dynamic compensation update mechanism. On the other hand, by automatically protecting Q against external faults, we ensure its execution and that it happens within the context of the same session s. Rule L-RECOVER-IN is similar, but in this case the fault message is internal to the transaction, *i.e.*, comes from P. Rule L-SCOPE-IN updates the compensation of a transaction. Rule L-SCOPE-OUT allows outputs to go outside transactions, provided that they are not termination signals for the transaction itself. Rule L-SCOPE is used when input or output is from an inner transaction scope. Finally, rules L-BLOCK and L-BLOCK-S define the behaviour of a protection block, both when an interaction occurs within different sessions or within its own session, respectively.

Definition 2.9. The reduction relation between processes, denoted as $P \rightarrow P'$ is defined as $P \xrightarrow{(r,\tau(z),\alpha_c)} P'$, with $r \in \mathcal{S}$, $z \in \mathcal{N} \cup \mathcal{T}$ and α_c a compensation update. We denote also by $\longrightarrow^*$ the reflexive and transitive closure of the reduction relation.

Lemma 2.10. *Well-formedness is preserved by reductions.*

Proof. It follows by structural induction on P. □

For later convenience, we define also the usual structural congruence relation on processes.

Definition 2.11. Structural congruence $\equiv$ is the smallest congruence relation on processes satisfying the α-conversion law, the abelian monoid laws for parallel and inaction, and the laws in Fig. 6.

The scope laws are standard. The law $\langle P \mid Q \rangle_r \equiv \langle P \rangle_r \mid \langle Q \rangle_r$ flattens nested protected blocks and the law $\langle \mathbf{0} \rangle_r \equiv \mathbf{0}$ is straightforward.

3. Case study

This section provides a simple case study for exemplifying the calculus and for motivating the need of correctness criteria.

3.1. Order transaction

The first version of this case study consists of a simple ordering system that, upon the order request from a client, it must take care of the payments. The system is modelled as depicted in Fig. 7, which for simplicity only considers one client and

$$\textbf{OrderTransaction1} \stackrel{\text{def}}{=} \textbf{Client1} \mid \textbf{Shop1} \mid \textbf{Bank}$$

$$\textbf{Client1} \stackrel{\text{def}}{=} u\lfloor \overline{client} \mid (ack.\bar{t} + ack.recp.\overline{okShop}), \mathbf{0} \rceil_{r_0}$$

$$\textbf{Shop1} \stackrel{\text{def}}{=} t\lfloor client.(\overline{ack} \mid \overline{initchg}) \mid \textbf{Charge} \mid okShop.\overline{ok}, \mathbf{0} \rceil_{r_0}$$

$$\begin{aligned}\textbf{Charge} \stackrel{\text{def}}{=} c\lfloor & initchg.\overline{bank} \\ & \mid (valid\lfloor \lambda X.refunded \mid X \rfloor.(\overline{recp} \mid ok.\overline{end1}) + invalid.\bar{t}) \\ & \mid ended1\lfloor \lambda X.\mathbf{0} \rfloor, \bar{q} \rceil_{r_1}\end{aligned}$$

$$\begin{aligned}\textbf{Bank} \stackrel{\text{def}}{=} q\lfloor & bank.(\nu\, y)\,(\overline{y} \mid (y\lfloor \lambda X.\overline{refunded} \mid X \rfloor.\overline{valid} + y.\overline{invalid})) \\ & \mid end1\lfloor \lambda X.\mathbf{0} \rfloor.\overline{ended1}, \mathbf{0} \rceil_{r_1}\end{aligned}$$

Fig. 7. Ordering system example.

$$\textbf{OrderTransaction2} \stackrel{\text{def}}{=} \textbf{Client2} \mid \textbf{Shop2} \mid \textbf{Bank} \mid \textbf{Warehouse}$$

$$\textbf{Client2} \stackrel{\text{def}}{=} u\lfloor \overline{client} \mid (ack.\bar{t} + ack.(recp.\overline{okShop} \mid delivered.\overline{yesShop})), \mathbf{0} \rceil_{r_0}$$

$$\begin{aligned}\textbf{Shop2} \stackrel{\text{def}}{=} t\lfloor & client.(\overline{ack} \mid \overline{initchg} \mid \overline{initpck}) \\ & \mid \textbf{Charge} \mid \textbf{Pack} \mid okShop.\overline{ok} \mid yesShop.\overline{yes}, \mathbf{0} \rceil_{r_0}\end{aligned}$$

$$\begin{aligned}\textbf{Pack} \stackrel{\text{def}}{=} p\lfloor & initpck.\overline{pack} \\ & \mid (exists\lfloor \lambda X.unpacked \mid X \rfloor.(\overline{delivered} \mid yes.\overline{end2}) \\ & + notExists.\bar{t}) \mid ended2\lfloor \lambda X.\mathbf{0} \rfloor, \bar{k} \rceil_{r_2}\end{aligned}$$

$$\begin{aligned}\textbf{Warehouse} \stackrel{\text{def}}{=} k\lfloor & pack.(\nu\, x)\,(\overline{x} \mid (x\lfloor \lambda X.\overline{unpacked} \mid X \rfloor.\overline{exists} \\ & + x.\overline{notExists})) \mid end2\lfloor \lambda X.\mathbf{0} \rfloor.\overline{ended2}, \mathbf{0} \rceil_{r_2}\end{aligned}$$

Fig. 8. Ordering system with unexpected behaviour.

one order. Notice that the generalisation of this example, where the shop has to interact with several clients, implies the use of a session initiation mechanism, as discussed in Section 2.1.

After receiving a client ordering request, the shop tries to charge the client. In this example, we are considering that the payment is done by the bank. Thus, the shop sends a message to the bank and it starts a new interaction session. Notice that the client may cancel the **Shop** transaction, causing the execution of the compensation of this transaction. If charging is successfully accomplished, the shop sends a message to the client. The client confirms to the shop the receipt delivery. The shop informs the bank that the transaction with the client has ended, removing all the compensations of the **Shop** transaction. Then, after the ending notification, the **Bank** transaction also removes its compensations. Notice that compensations are dynamically built. For example, when the bank starts to interact with the shop, it installs the compensation $\lambda X.\overline{refunded} \mid X$ and, when it receives the message *end1*, it installs the compensation $\lambda X.\mathbf{0}$. The ability of changing compensations within the execution of the process is an important feature of the dynamic installation mechanism. For instance the compensation for the **Bank** transaction is only removed when the bank receives a terminating message from the subtransaction **Charge**, the only transaction that is interacting with it. The installation of $\lambda X.\mathbf{0}$ can be regarded as a transaction commit, since compensations are cleaned and recovery is no longer possible. Notice that the interactions between the client and the shop are done within session r_0, and the interactions between the transaction **Charge** and the bank are done within session r_1. Also, communications within transactions belonging to different interaction sessions do not install compensations.

In this example, it is natural for the programmer to expect the following behaviour of the system: if the subtransactions of the shop have ended, then compensations are not expected to occur; if the client chooses to cancel the **Shop** transaction after the bank has validated the payment but before the transaction ends, a refund must be processed.

3.2. Order transaction with warehouse

The second example extends the previous one by adding order packing to the ordering system. The system is modelled as depicted in Fig. 8 and, similarly to the previous example, it only considers one client and one order.

In this case, after receiving the message from the client, the shop starts, within different interaction sessions, two subtransactions, one to charge the client and another to pack the order. Notice that transactions **Charge** and **Bank** are the same as in the previous example. The interaction session of transactions **Pack** and **Warehouse** has a similar behaviour

$$\textbf{OrderTransaction3} \stackrel{\text{def}}{=} \textbf{Client3} \mid \textbf{Shop3} \mid \textbf{Bank} \mid \textbf{Warehouse}$$

$$\textbf{Client3} \stackrel{\text{def}}{=} u\big[\overline{client} \mid (ack.\bar{t} + ack.done), \mathbf{0}\big]_{r_0}$$

$$\textbf{Shop3} \stackrel{\text{def}}{=} t\big[client.(\overline{ack} \mid \textbf{Charge} \mid \textbf{Pack} \mid delivered.recp.(\overline{done} \mid \overline{ok} \mid \overline{yes})), \mathbf{0}\big]_{r_0}$$

Fig. 9. Ordering system with expected behaviour.

to the interaction session of transactions **Charge** and **Bank**, but with different actions. In this example, the client may also cancel the **Shop** transaction. In such case, the **Shop** transaction fails and its compensations are executed. However, in this example, the cancelling notification by the client can happen after the subtransaction **Pack** has been successfully accomplished, being no longer compensable (after *ended2* message has been communicated). In this case, the behaviour of the system is not the expected one, since the client can get the goods for free. Later we shall see how we can detect such wrong behaviour under our formal framework.

A possible solution for overcoming the unexpected behaviour described above is presented in Fig. 9. Later, we shall see how our notion of correctness asserts that this is valid.

4. Correctness criteria

A programmer expects that communicating programs should realise correct conversations, even when one of the interacting partners fails due to an unexpected event. Ensuring the correctness of a compensable process is a challenging task. In fact, when a programmer designs a system, he expects that its transactional behaviour is consistent, *i.e.*, in case of fault the system will recover and reach a consistent state. This behaviour depends on a set of transactional requirements, which differs from one context to another, *i.e.*, the expected behaviour is dependent on the application context. In this section we define a notion of correctness for compensable processes. The proposed notion takes into account that in real world scenarios some actions are not compensable and compensations can be more than a simple "undo".

Before presenting the notion of a correct compensable process, we introduce some useful definitions.

Definition 4.1. Let P be a process and $s \in \mathcal{L}^n$ a trace. We say that P has s as a computation, $P \xrightarrow{s}$, if $s = \alpha_1 \ldots \alpha_n$ and $P \xrightarrow{\alpha_1} P_1 \ldots \xrightarrow{\alpha_n} P_n$. We also define $\mathcal{L}^* = \bigcup_{i \in \mathbb{N}} \mathcal{L}^i$.

Definition 4.2. Let P be a process. We define $L(P) = \{s \in \mathcal{L}^* \mid P \xrightarrow{s}\}$.

Given $s, s' \in \mathcal{L}^*$, we write $s \prec s'$ whenever the trace s is a subsequence of the trace s'.

Definition 4.3. The function $\mathsf{sn} : \mathcal{L}^* \longrightarrow 2^{\mathcal{N}}$, which gives the set of session names occurring in a trace, is defined as:

$$\begin{aligned} \mathsf{sn}(\epsilon) &= \emptyset \\ \mathsf{sn}((r, \alpha_o).s) &= \{r\} \cup \mathsf{sn}(s) \\ \mathsf{sn}((r, \alpha_i, \alpha_c).s) &= \{r\} \cup \mathsf{sn}(s). \end{aligned}$$

Definition 4.4. The function $\mathsf{com} : \mathcal{L}^* \times \mathcal{S} \longrightarrow \mathcal{N}^*$, which maps each trace s to the sequence of communicated names within session r, is defined as:

$$\begin{aligned} \mathsf{com}(\epsilon, r) &= \epsilon \\ \mathsf{com}((r', \alpha_o).s, r) &= \mathsf{com}(s, r) \\ \mathsf{com}((r', a(\tilde{v}), \alpha_c).s, r) &= \mathsf{com}(s, r) \\ \mathsf{com}((r', \tau(a), \alpha_c).s, r) &= \begin{cases} a.\,\mathsf{com}(s, r) & \text{if } r' = r, \\ \mathsf{com}(s, r) & \text{otherwise.} \end{cases} \end{aligned}$$

Since the expected behaviour of a system is dependent on the application context, we propose a notion of correct compensable process that is parametrised by a *correctness map* provided by the programmer. The correctness map should express how meaningful interactions can be compensated, *i.e.*, the programmer gives a set of possible finite sequences of

interactions that compensate each meaningful interaction. This map and the possible sequences are defined over the set of free names. Notice that the correctness mapping may not be, and usually is not, directly equivalent to the compensation pairs that can be extracted from a compensable process.

Definition 4.5. A *correctness mapping* $\varphi : \mathcal{N} \longrightarrow 2^{\mathcal{N}^*}$ maps each name $n \in \mathcal{N}$ to a set of sequences of names.

Consider the previous examples. In **OrderTransaction1**, a feasible correctness map could be defined as: $\varphi(valid) = \{refunded\}$; $\varphi(ok) = \emptyset$; and $\varphi(x) = \{\epsilon\}$ for each $x \in \mathsf{fn}(\textbf{OrderTransaction1}) \setminus \{valid, ok\}$. Notice that a mapping to the set $\{\epsilon\}$ or a mapping to the empty set have completely different meanings. Mapping to $\{\epsilon\}$ means that the programmer does not expect to see a compensation trace for that action. However, the mapping to the empty set will mean, as we shall see, that after the communication of *ok*, the programmer is not expecting to see the previous defined compensations. In fact, this is coherent for this example, since after doing *ok* the client was notified with a receipt and he confirmed the receipt delivery. Thus, in this situation, it would not make sense to execute any compensation.

In the case of the example **OrderTransaction2**, a feasible correctness map could be defined as: $\varphi(valid) = \{refunded\}$; $\varphi(pack) = \{unpacked\}$; $\varphi(x) = \{\epsilon\}$, for each $x \in \mathsf{fn}(\textbf{OrderTransaction2})$ such that $x \notin \{valid, unpacked\}$. Notice that in this case, it does not make sense to define $\varphi(ok) = \emptyset$ and $\varphi(yes) = \emptyset$, since a client may cancel the transaction after the packing has been successfully accomplished. Still, under our correctness *criteria*, this process will not be correct, as we shall see.

Definition 4.6. Let $|||_{} : \mathcal{L}^* \times \mathcal{L}^* \longrightarrow 2^{\mathcal{L}^*}$ be a commutative operator, defined as:

$$\begin{aligned} \epsilon \,|||\, \epsilon &= \{\epsilon\} \\ \epsilon \,|||\, \alpha &= \{\alpha\} \\ \alpha.s \,|||\, \beta.r &= \alpha.(s \,|||\, \beta.r) \cup \beta.(s \,|||\, \beta.r). \end{aligned}$$

We further extend $|||$ to sets as an associative operator, $|||_{} : 2^{\mathcal{L}^*} \times 2^{\mathcal{L}^*} \longrightarrow 2^{\mathcal{L}^*}$, as follows:

$$S \,|||\, S' = \bigcup_{(s,s') \in S \times S'} s \,|||\, s',$$

where $S, S' \subset \mathcal{L}^*$.

The intuition behind the operator $|||$ is that, given two traces, we are able to generate their interleaving. Clearly, the interleaving is not unique and, thus, we may have several different alternative interleaved traces.

Definition 4.7. A process is *passive* if it can only perform an input labelled transition as a first possible action (no internal moves or outputs) or no action at all.

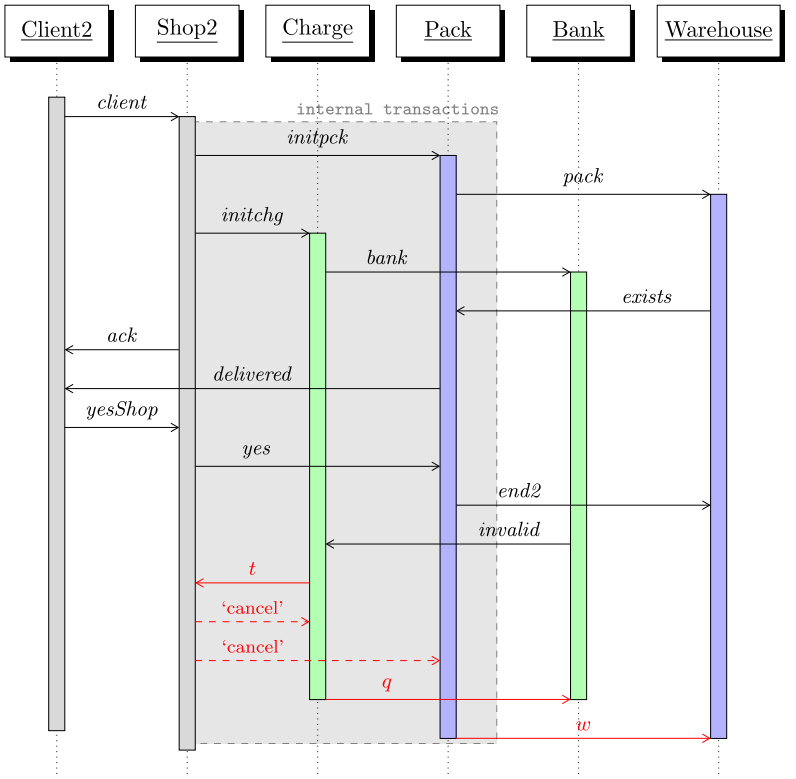
Clearly, we may have a non-passive process P weak bisimilar to a passive process Q. We note however that, assuming non-divergence, the non-passive process P will evolve by internal moves to a passive process P', *i.e.*, $P \Longrightarrow P'$, that is bisimilar to Q.

In the following, we start by defining the notion of a correct compensable process with respect to a correctness map φ and with respect to a session s (φ-correct with respect to s). Then, if the process is φ-correct with respect to all sessions within the process, we say that it is a φ-correct process.

Definition 4.8. Let P be a well formed compensable process, φ a correctness mapping and $r \in \mathsf{sn}(P)$. P is *φ-correct with respect to r* if, whenever $P \xrightarrow{s} \xrightarrow{\alpha} \xrightarrow{s'} Q$, with s and s' traces, $\alpha = (r, t, \mathsf{id})$ or $\alpha = (r, \tau(t), \mathsf{id})$, t a transaction identifier of session r and Q a passive process, exists $s^* \prec s'$ such that $\mathsf{com}(s^*, r) = \beta_1 \ldots \beta_n \in |||_{i=1}^{m} \varphi(\alpha_i)$ and $\beta_1, \ldots, \beta_n \in \mathsf{nl}(k, P)$, with $\mathsf{com}(s, r) = \alpha_1 \ldots \alpha_m$ and $\alpha_1, \ldots, \alpha_m \in \mathsf{nl}(k-1, P)$, for $k > 0$.

The motivation behind previous definition is as follows. If we observe a fault notification on t within session r as process P evolves to process Q, a passive process, then we will also expect each communication within session r to be compensated accordingly to φ. Thus, we extract the communicated names α_i within session r before the fault notification through function com, we compute all possible compensation traces for names α_i by interleaving expected compensation traces as provided by compensation map φ and, finally, we must verify if at least one of the possible compensation traces is observed after fault notification. Note that, in the definition, we require communicated names in the compensation trace to be one level above.

Definition 4.9. Let φ be a correctness mapping. A well formed process P is *φ-correct* if it is φ-correct with respect to all $r \in \mathsf{sn}(P)$ sessions.



$$
\begin{aligned}
s = {} & (r_0, \tau(client), \mathsf{id})(r_2, \tau(initpck), \mathsf{id})(r_2, \tau(pack), \mathsf{id})(r_2, \tau(z), \mathsf{id}) \\
& (r_1, \tau(initchg), \mathsf{id})(r_1, \tau(bank), \mathsf{id})(r_2, \tau(exists), \mathsf{id})(r_0, \tau(ack), \mathsf{id}) \\
& (r_0, \tau(delivered), \mathsf{id})(r_0, \tau(yesShop), \mathsf{id})(r_2, \tau(yes), \mathsf{id})(r_2, \tau(end2), \mathsf{id}) \\
& (r_1, \tau(y), \mathsf{id})(r_1, \tau(invalid), \mathsf{id}))(r_0, \tau(t), \mathsf{id})(r_1, \tau(q), \mathsf{id})(r_2, \tau(k), \mathsf{id})
\end{aligned}
$$

Fig. 10. A trace witnessing that process **OrderTransaction2** is not φ-correct.

This definition allows to conclude that the process *OrderTransaction2* is not φ-correct, where φ is the feasible consistency map previously defined, *i.e.*, a map that is expected to be defined by the programmer under the process requirements. For instance, the trace depicted in Fig. 10 is a witness that *OrderTransaction2* is not φ-correct with respect to session r_2. Note that it was not possible to perform the payment of the already received product, but it was not observed the compensation *unpacked* within session r_2. *OrderTransaction2* is also not φ-correct with respect to session r_1, as it can be seen in Fig. 11. In this case, the client has paid the product but will never receive it or a refund.

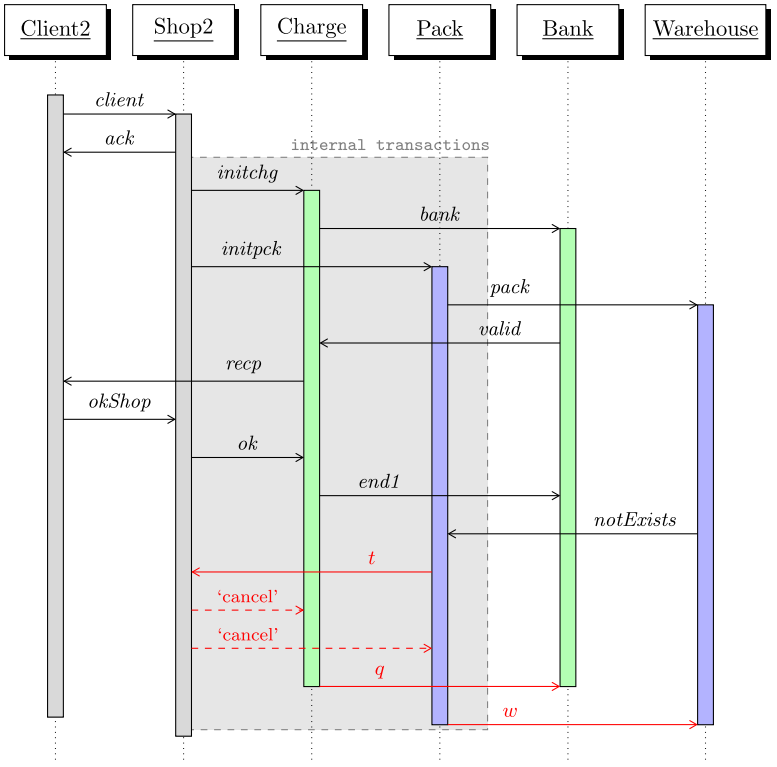
In the following, we will present the conditions that should be preserved to ensure that the composition of correct compensable processes is also a correct process. The first condition describes a notion of independence, which is necessary for parallel composition.

Definition 4.10. Two processes P and Q are *independent* if $\mathsf{sn}(P) \cap \mathsf{sn}(Q) = \emptyset$, $\mathsf{inp}(P) \cap \mathsf{fn}(P) \cap \mathsf{inp}(Q) \cap \mathsf{fn}(Q) = \emptyset$ and $\mathsf{out}(P) \cap \mathsf{fn}(P) \cap \mathsf{out}(Q) \cap \mathsf{fn}(Q) = \emptyset$.

Clearly, if some of the above conditions were false, it could not be assured that each trace of $P \mid Q$ would be φ-correct, for a given correctness mapping φ. For instance, non-empty intersection of inputs could raise undesirable internal communications compromising the compensating trace of P or Q. With this definition we can state the following properties.

Proposition 4.11. *Let P and Q be independent compensable processes such that $P \mid Q$ is a well formed process. Then, the parallel composition $P \mid Q$ is φ-correct if both P and Q are φ-correct.*

Proof. Let $r \in \mathsf{sn}(P \mid Q)$ and $P \mid Q \xrightarrow{s} \xrightarrow{\alpha} \xrightarrow{s'} R$, with R a passive process, $\alpha = (r, t, \mathsf{id})$ or $\alpha = (r, \tau(t), \mathsf{id})$, and t a transaction identifier of r. Since P and Q are independent, we know that $r \notin \mathsf{sn}(P) \cap \mathsf{sn}(Q)$. Let us assume without loss of generality that $r \in \mathsf{sn}(P)$ and let $s_1.\alpha.s_1' \in L(P)$ be the underlying trace of P within $s.\alpha.s'$. Since P and Q are independent, we know



$$
\begin{aligned}
s = {} & (r_0, \tau(client), \mathsf{id})(r_0, \tau(ack), \mathsf{id})(r_1, \tau(initchg), \mathsf{id})(r_1, \tau(bank), \mathsf{id}) \\
& (r_2, \tau(initpck), \mathsf{id})(r_2, \tau(pack), \mathsf{id})(r_1, \tau(y), \mathsf{id})(r_1, \tau(valid), \mathsf{id}) \\
& (r_0, \tau(recp), \mathsf{id})(r_0, \tau(okShop), \mathsf{id})(r_1, \tau(ok), \mathsf{id})(r_1, \tau(end1), \mathsf{id}) \\
& (r_2, \tau(z), \mathsf{id}))(r_2, \tau(notExists), \mathsf{id}))(r_0, \tau(t), \mathsf{id})(r_1, \tau(q), \mathsf{id})(r_2, \tau(k), \mathsf{id})
\end{aligned}
$$

Fig. 11. A trace witnessing that process **OrderTransaction2** is not φ-correct.

that $\mathsf{com}(s_1.\alpha.s_1', r) = \mathsf{com}(s.\alpha.s', r)$. Moreover, since P is φ-correct, we know that it exists a trace $s_1^* \prec s_1'$ such that $\mathsf{com}(s_1^*, r) = \beta_1 \ldots \beta_n \in |||_{i=1}^m \varphi(\alpha_i)$, with $\mathsf{com}(s_1, r) = \alpha_1 \ldots \alpha_m$. Then, consider $s^* \prec s'$ such that s_1^* is the underlying trace of P within s^*. Since P and Q are independent, s^* is such that $\mathsf{com}(s^*, r) = \mathsf{com}(s_1^*, r) = \beta_1 \ldots \beta_n \in |||_{i=1}^m \varphi(\alpha_i)$, with $\mathsf{com}(s, r) = \mathsf{com}(s_1, r) = \alpha_1 \ldots \alpha_m$. Therefore, the thesis holds. □

Proposition 4.12. *Let P and Q be independent compensable processes such that $t[P, Q]_r$ is a well formed process. Then, $t[P, Q]_r$ is φ-correct if both P and Q are φ-correct.*

Proof. Let $r' \in \mathsf{sn}(P \mid Q) \cup \{r\}$ and $t[P, Q]_r \xrightarrow{s} \xrightarrow{\alpha} S \xrightarrow{s'} R$, with R a passive process, $\alpha = (r, p, \mathsf{id})$ or $\alpha = (r, \tau(p), \mathsf{id})$, and p a transaction identifier of r'. (1) If $r = r'$, since $t[P, Q]_r$ is a well formed process, $r \notin \mathsf{sn}(P \mid Q)$ and p is a transaction identifier of session r, *i.e.*, $p = t$. Moreover, it can be easily proved by induction that, if $r \notin \mathsf{sn}(P)$, then $r \notin \mathsf{sn}(s)$, for all $s \in L(P)$. In particular, $\mathsf{com}(s, r) = \epsilon$ and, therefore, the thesis holds choosing $s^* = \epsilon$ accordingly to Definition 4.8. (2) We have three cases, (2.1) t does not occur in s or s', (2.2) t occurs in s' or (2.3) t occurs in s. (2.1) If t does not occur in s or s', then $r' \in \mathsf{sn}(P)$ and p is a transaction identifier of r'. Thus, since s is a trace of P and P is φ-correct, the thesis holds. (2.2) If t occurs in s', then t does not occur in s and $r' \in \mathsf{sn}(P)$ with p a transaction identifier of r'. Since t has occurred within s', we have $t[P, Q]_r \xrightarrow{s} \xrightarrow{\alpha} S \xrightarrow{s_1'} t[P', Q]_r \xrightarrow{\alpha'} \mathsf{extr}(P') \mid \langle Q \rangle_r \xrightarrow{s_2'} R$, with $s' = s_1'.\alpha'.s_2'$ and either $\alpha' = (r, t, \mathsf{id})$ or $\alpha' = (r, \tau(t), \mathsf{id})$. As in the previous case, s is a trace of P and let s'' be the underlying trace of P within $s_1'.\alpha'.s_2'$. By Definition 4.8 and because P is φ-correct, there is s^* such that $s^* \prec s''$ and $\mathsf{com}(s^*, r') = \beta_1 \ldots \beta_n \in |||_{i=1}^m \varphi(\alpha_i)$, with $\mathsf{com}(s, r') = \alpha_1 \ldots \alpha_m$. Moreover, $\beta_1, \ldots, \beta_n \in \mathsf{nl}(k, t[P, Q]_r)$ and $\alpha_1 \ldots \alpha_m \in \mathsf{nl}(k-1, t[P, Q]_r)$, for some $k > 0$. Thus, $\mathsf{com}(s^*, r')$ occurs at an higher level and is part of the traces of compensations found within P'. Since these compensations are protected, they are not interrupted by t. Because P and Q are independent, $\mathsf{extr}(P')$ and $\langle Q \rangle_r$ are also independent since extr does not introduce names. Thus, there is $s^{**} \prec s_1'.\alpha'.s_2'$ such that $\mathsf{com}(s^{**}, r') = \mathsf{com}(s^*, r')$, and the thesis holds. (2.3) If t occurs in s, then $t[P, Q]_r \xrightarrow{s_1} \xrightarrow{\alpha'} \mathsf{extr}(P') \mid \langle Q \rangle_r \xrightarrow{s_2} \xrightarrow{\alpha} S \xrightarrow{s'} R$, with $s = s_1.\alpha'.s_2$ and either $\alpha' = (r, t, \mathsf{id})$ or

$$C_\varphi[\![\bullet]\!] ::= \bullet \mid C_\varphi[\![\bullet]\!]|P \mid P|C_\varphi[\![\bullet]\!] \mid \langle C_\varphi[\![\bullet]\!]\rangle_s \mid t\,[C_\varphi[\![\bullet]\!], P]_r \mid (\nu\, x)\, C_\varphi[\![\bullet]\!], \text{ if } x \notin \mathrm{dom}(\varphi) \cup \mathrm{img}(\varphi)$$

$$D_\varphi[\![\bullet, \bullet]\!] ::= C^1_\varphi[\![\bullet]\!]|C^2_\varphi[\![\bullet]\!]$$

Fig. 12. Syntax of φ-safe contexts.

$\alpha' = (r, \tau(t), \mathrm{id})$. Again, because P and Q are independent, $\mathtt{extr}(P')$ and $\langle Q\rangle_r$ are also independent since extr does not introduce names. Moreover, $\mathtt{extr}(P')$ is equivalent to trigger a set of transaction identifiers within P'. Thus, since P and $\langle Q\rangle$ are φ-correct and independent, by Proposition 4.11, the thesis holds. □

Interactions can happen in different execution contexts. Since all our interactions are binary, we introduce double execution contexts, *i.e.*, two execution contexts that can interact. We defined next a grammar that generates *φ-safe execution contexts* for a correctness mapping φ.

Definition 4.13. The grammar in Fig. 12 inductively defines φ-safe contexts, denoted by $C_\varphi[\![\bullet]\!]$, and double φ-safe contexts, denoted by $D_\varphi[\![\bullet, \bullet]\!]$.

The Propositions 4.14 and 4.15 show that for safe contexts with respect to a correctness map φ, the composition of correct processes is also a correct process.

Proposition 4.14. *Let φ be a correctness mapping, P be φ-correct process and $C_\varphi[\![\bullet]\!]$ be a safe context such that $C_\varphi[\![P]\!]$ is a well formed process. If $C_\varphi[\![\mathbf{0}]\!]$ is φ-correct and independent with respect to P, then $C_\varphi[\![P]\!]$ is φ-correct.*

Proof. The proof is by induction on the context C_φ and by Propositions 4.11 and 4.12. □

Proposition 4.15. *Let φ be a correctness mapping, P and Q be a φ-correct and independent processes , and $D_\varphi[\![\bullet, \bullet]\!]$ be a safe double context such that $D_\varphi[\![P, Q]\!]$ is a well formed process. If $D_\varphi[\![\mathbf{0}, \mathbf{0}]\!]$ is φ-correct and independent with respect to process P and Q, then $D_\varphi[\![P, Q]\!]$ is φ-correct.*

Proof. The proof is by induction on the context D_φ and by Propositions 4.11 and 4.14. □

In some cases, we are interested in a more relaxed notion of composition. Consider the example **OrderTransaction3** and the previous correctness mapping φ. We can see that both processes **Warehouse** | **Pack** and **Bank** | **Charge**, within session r_2 and session r_1 respectively, are φ-correct. Also, they are independent processes. However, we cannot apply the previous results to prove that their composition is also φ-correct, because **Pack** and **Charge** are subtransactions of **Shop**, but **Warehouse** and **Bank** are not. So, the following results generalise the idea of composition in order to extend the correctness result to this kind of generalised composition.

Lemma 4.16. *Let φ be a correctness mapping, $P \mid Q$ be a φ-correct compensable process, and $C_\varphi[\![\bullet]\!]$ be a φ-safe context such that $P \mid C_\varphi[\![Q]\!]$ is a well formed process, $C_\varphi[\![0]\!]$ is independent of P and $C_\varphi[\![\bullet]\!]$ does not bind $x \in \mathrm{fn}(P)$. If $C_\varphi[\![\mathbf{0}]\!]$ is φ-consistent, then $P \mid C_\varphi[\![Q]\!]$ is φ-consistent.*

Proof. The proof is by induction on context $C_\varphi[\![\bullet]\!]$. □

Theorem 4.17. *Let φ be a correctness mapping, $P_1 \mid Q_1$ and $P_2 \mid Q_2$ be independent and φ-correct compensable processes, and $D_\varphi[\![\bullet, \bullet]\!]$ be a safe double context such that $D_\varphi[\![P_1 \mid P_2, Q_1 \mid Q_2]\!]$ is a well formed process. If $D_\varphi[\![\mathbf{0}, \mathbf{0}]\!]$ is φ-correct, then $D_\varphi[\![P_1 \mid P_2, Q_1 \mid Q_2]\!]$ is φ-correct.*

Proof. The proof is by induction on the context D_φ and by Proposition 4.11 and Lemma 4.16. □

We are now able to interpret the notion self-healing systems [30], *i.e.*, systems that can detect and recover from failures, as correct compensable processes. Such a system should perceive that is not operating correctly and make the necessary adjustments to restore itself to consistency. Thus, for this interpretation, it is necessary to express the behaviour of the system should have in case of interruption. Within our setting, this is done by a correctness mapping. We define self-healing with respect to a correctness mapping as follows.

Definition 4.18. Let P be a process and φ a correctness mapping. P is *self-healing with respect to φ* if P is φ-correct.

Moreover, a *self-healing composition* should be able to automatically detect that some service composition requirements are no longer satisfied by the implementation and react to requirement violations [30]. Our notion of correctness describes

the idea of restoring the consistency of these kind of systems. In particular, our results provide a first attempt to formally build self-healing compositions from existing ones.

From the above results, if the programmer of a system ensures correctness of compensable transactions (or of self-healing system), then by satisfying the conditions defined above, many kinds of compositions are also correct under our theory.

5. A model checking approach

To demonstrate the viability of our approach, we have implemented a Prolog interpreter to be used together with the ProB model checker [23]. In Section 5.1, we briefly describe the most relevant aspects of ProB and we discuss how it can be used to model check our specifications. In Section 5.2, we briefly describe the Linear Temporal Logic (LTL) [12] and an extended version, named LTL^e, supported by the ProB tool. Then, in Section 5.3, we discuss how to specify correctness properties in LTL^e. Finally, in Section 5.4, we model previous examples and we analyse their correctness properties with ProB.

5.1. Automated verification

The ProB model checker allows to model check a wide range of system specifications, requiring only a specification to be described using Prolog predicates. We must describe the system transitions between the different states through the predicates `start/1`, `trans/3` and `prop/2`, where:

- `start(S)` defines the initial state for the system;
- `trans(Act,S1,S2)` computes for every state `S1` the outgoing transitions `Act` and resulting new states `S2`;
- `prop(S,C)` states that condition `C` holds at state `S`.

Then, the main task becomes translating our language rules and specifications to a set of `trans/3`, as well as, `start/1` and `prop/2` predicates. As defined in Section 2.3, the system transitions of our compensating calculus are given by a LTS, *i.e.*, a triple $(\mathcal{P}, \mathcal{L}, \{\xrightarrow{\alpha}: \alpha \in \mathcal{L}\})$ where $\mathcal{P}$ is a set of states (processes), $\mathcal{L}$ is a set of transition labels (the actions that processes perform) and $\xrightarrow{\alpha} \subset \mathcal{P} \times \mathcal{P}$ for each $\alpha \in \mathcal{L}$ is a transition relation. Hence, it is necessary to interpret our LTS with these predicates.

We will reproduce below some of our operational semantics rules and their corresponding encoding in Prolog as required by ProB. First we must define what are the states and the labels in our interpretation. Within our interpreter, a *state* or process configuration is described by a configuration of four parameters `conf(P, Bag, SMap, CMap)`, where `P` is the process that is being executed, `Bag` is the set of names in `P`, `SMap` maps each session to a pair that contains the state of the session and the list of sets of compensation traces, with at most one trace, that need to be executed if a fault occurs in that instant, and `CMap` is the correctness map. Tracking the set of names `Bag` in `P` is to avoid name captures when we rename bound names in `P`. The map `SMap` has an important role in our interpretation, namely for checking if a process is correct with respect to the correctness map, which in this interpretation is given by `CMap`. In this context, each session name is mapped in `SMap` to a pair that contains the state of the session and a list of sets of compensation traces. Each set of this list contains at most a compensation trace. This trace is given by `CMap`, which is associated to a given action. Thus, it is added to `SMap` when the action is observed. Notice that this is a simplification of the definition of correctness map, since the original definition of correctness map in Section 4 allows each action to be associated with a set of alternative compensation traces. Notice that, for renaming and correctness verification purposes, a state within this interpretation includes more information than the state of the LTS of our compensating calculus. The reason to track the compensation traces in `SMap` and the state of a session will be further discussed in Section 5.3.

The predicate `start/1` is an initial configuration of our model, where the process, which has to be written in Prolog language, should correspond to a process in our compensating calculus that describes the initial system. A process in our interpretation is given by the grammar in Fig. 13.

Definition 5.1. The grammar in Fig. 13 defines de grammar of processes in the Prolog interpretation.

In the syntax in Fig. 13, A, B and X denote channel names, V, W and Y lists of these names, T represents transaction identifiers, Z denotes lists that contain channel names or transaction identifiers, and S and R are used for session names. Note that the keyword `hole` corresponds to the process variable X and it should occur within a compensation update. For instance, if in the compensating calculus the compensation update is $\lambda X.X \mid P$ then the same is represented in the Prolog interpretation as `hole | P`. Note also that, for model checking purposes, we interpret the process variable X as a single place holder and, thus, we rule out processes where such variables are captured beyond the current execution level, *e.g.*, such as in the process $a(\tilde{x})\lfloor\lambda X.b(\tilde{y})\lfloor\lambda Y.X \mid Y\rfloor\rfloor$.

Definition 5.2. Given a process $P \in \mathcal{P}$, its interpretation $f(P)$ in Prolog is define as follows:

$$
\begin{aligned}
I &::= \texttt{inp}(\texttt{ch}(A), Y, P)\\
PP, QQ &::= I\\
&\mid \texttt{prefix}(I, P)\\
&\mid \texttt{choice}(PP, QQ)\\
P, Q &::= \texttt{inaction}\\
&\mid \texttt{hole}\\
&\mid \texttt{out}(\texttt{ch}(A), V)\\
&\mid \texttt{out}(\texttt{ti}(T))\\
&\mid PP\\
&\mid \texttt{parallel}(P, Q)\\
&\mid \texttt{restriction}(X, P)\\
&\mid \texttt{transaction}(\texttt{ti}(T), P, Q, R)\\
&\mid \texttt{block}(P, R)
\end{aligned}
$$

Fig. 13. The syntax of processes in the Prolog interpretation.

- $f(\mathbf{0}) =$ `inaction`;
- $f(\overline{a}\langle\tilde{v}\rangle) =$ `out(ch(a), [v1,...,vn])`;
- $f(\overline{t}) =$ `out(ti(t))`;
- $f(X) =$ `hole`;
- $f(a(\tilde{x})\lfloor\lambda X.Q\rfloor) =$ `inp(ch(a), [x1,...,xn],` $f(Q)$`)`;
- $f(a(\tilde{x})\lfloor\lambda X.Q\rfloor.P) =$ `prefix(`$f(a(\tilde{x})\lfloor\lambda X.Q\rfloor)$`,` $f(P)$`)`;
- $f(\sum_{i\in\{i_1,\ldots,i_n\}} a_i(\tilde{x}_i)\lfloor\lambda X.Q_i\rfloor.P_i) =$ `choice(`$f(a_{i_1}\lfloor\lambda X.Q_{i_1}\rfloor.P_{i_1})$`,` $f(\sum_{i\in\{i_2,\ldots,i_n\}} a_i(\tilde{x}_i)\lfloor\lambda X.Q_i\rfloor.P_i)$`)`;
- $f(P \mid Q) =$ `parallel(`$f(P)$`,` $f(Q)$`)`;
- $f((\nu\, x)\ P) =$ `restriction(x,` $f(P)$`)`;
- $f(t[P, Q]_r) =$ `transaction(ti(t),` $f(P)$`,` $f(Q)$`, r)`;
- $f(\langle P\rangle_r) =$ `block(`$f(P)$`, r)`;

and where $\tilde{v} = \{v_1, \ldots, v_n\}$ and $\tilde{x} = \{x_1, \ldots, x_n\}$.

The predicate `trans/3` has associated an action or operation label `Act`. In our interpretation, an *action label* `label(St, S, Tp, A, Y, C, W)` has seven parameters where: `St` is the state of the session, which can be `active`, `compensated` or `failed`; `S` is the name of the session; `Tp` is the type of the action, which can be `inp`, `out` or `tau`; `A` is the name associated to the channel or to the transaction identifier (in this last case we will use `T` instead); `Y` is the list of names associated to subject `A`; `C` if the compensation update, if any; and `W` is the list of bounded names in the label.

Similarly to the states, the labels in the Prolog interpretation ($\mathcal{L}_p$) include more information than the labels $\mathcal{L}$ defined for the LTS for the compensating calculus, such as the state of the session required to verify if the process is correct. We will discuss the need for this information in Section 5.3. We define next the correspondence map between $\mathcal{L}$ and $\mathcal{L}_p$.

Definition 5.3. Let $\mathcal{L}$ be the set of labels of the operational semantics and $\mathcal{L}_p$ the set of labels for their Prolog interpretation. We define the interpretation map for the labels $\eta : \mathcal{L} \to \mathcal{L}_p$ as follows:

- $\eta((r, t, id)) =$ `label(_, r, inp, ti(t), [ ], hole, [ ])`;
- $\eta((r, a(\tilde{x}), \lambda X.Q)) =$ `label(_, r, inp, ch(a),` $\tilde{x}$`, hole, [ ])`;
- $\eta((r, a(\tilde{x}), (\tilde{w})\lambda X.Q)) =$ `label(_, r, inp, ch(a),` $\tilde{x}$`, hole,` $\tilde{w}$`)`;
- $\eta((r, \tau(a), \lambda X.Q)) =$ `label(_,r, tau, ch(a),_,`$\gamma(X, Q)$`,[ ])`;
- $\eta((r, \tau(a), (\tilde{w})\lambda X.Q)) =$ `label(_, r, tau, ch(a),_,`$\gamma(X, Q)$`,` $\tilde{w}$`)`;
- $\eta((r, (\tilde{w})\overline{a}\langle\tilde{v}\rangle)) =$ `label(_, r, out, ch(a),` $\tilde{v}$`, empty,` $\tilde{w}$`)`;
- $\eta((r, \overline{t})) =$ `label(_, r, out, ti(t), [ ], empty, [ ])`.

Note that $\tilde{x}$ is mapped to `[x1,...,xn]` and $\tilde{w}$ is mapped to `[w1,...,wm]`. If $r = \perp$, then r is mapped to the reserved word `bottom`. The _'s are defined by the `trans/3` predicate, not following from the labels in the operational semantics. The function $\gamma(X, Q)$ replaces the variable X by the keyword `hole` in Q.

In the Prolog interpretation, we distinguish channel names (`ch`) from transaction identifiers (`ti`). This will be important for selecting the right rule to apply for each system transition.

We must define the predicate `trans/3` for each rule of the operational semantics. In what follows, we provide the definition of `trans/3` for three rules. The rule L-COM is encoded as

```
trans(label(St, S, tau, A, V, C, W),
        conf(parallel(P, Q), Bag, SMap, CMap),
        conf(NewProc, NewBag, NewSMap, CMap)) :-
   trans(label(Stq, R, out, B, V, empty, Wq),
        conf(Q, Bag, SMap, CMap),
        conf(Q1, _, _, CMap)),
   trans(label(Stp, S, inp, A, Y, Cp, W),
        conf(P, Bag, SMap, CMap),
        conf(TmpP, _, _, CMap)),
  complement(A, Y, B, V),
  substitution(W, Wq, Y, V, P, Q, TmpP, P1, Bag, NewBag),
  composeParallel(P1, Q1, TmpProc),
  composeRestriction(Wq,TmpProc, NewProc),
  setCompensation(S, R, Cp, C),
  updateSession(SMap, CMap, S, A, NewSMap, St).
```

As mentioned in Section 2.3, this rule allows communication, keeping track of the name of the subject, and propagating the compensation update coming from the input. Also, if communication does not occur within the same interaction session, the compensation update can only be the identity, represented by the keyword `hole`. This last condition is guaranteed by the predicate `setCompensation`. The predicate `substitution` allows the substitution of the communicated name, performing α-conversions whenever needed. The predicates `composeParallel` and `composeRestriction` are required to construct the process that remains after the communication. The predicate `updateSession` will be responsible for changing the state of the session, if needed. We will further discuss this in Section 5.3. Notice that we define also `trans/3` for the symmetric rule of L-COM.

Let us now consider the rule L-SCOPE-IN, which we encode as

```
trans(label(St, S, Tp, A, V, hole, []),
        conf(transaction(ti(T), P, Q, S), B, SMap, CMap),
        conf(Process, NewBag, NewSMap, CMap)) :-
    trans(label(St, bottom, Tp, A, V, C, W),
        conf(P, B, SMap, CMap),
        conf(P1, NewBag, NewSMap, CMap)),
    (   Tp == inp
    ;   Tp == tau),
    nameConflict(W, T, Q),
    composeCompensation(C, Q, F),
    composeRestriction(W,
        transaction(ti(T), P1, F, S), Process).
```

This rule updates the compensation of a transaction, which in the translation is done by the predicate `composeCompensation`. The predicate `composeRestriction` is needed once again if the label has bounded names.

The rule L-RECOVER-OUT is also an interesting case since in our interpretation not only we must update the state of the actual session, as well as the inner ones. We encode it as:

```
trans(label(St, S, inp, ti(T), [], hole, []),
        conf(transaction(ti(T), P, Q, S), Bag, SMap, CMap),
        conf(NewProc, Bag, NewSMap, CMap)) :-
  extr(P, R, SMap, TmpSMap),
  composeParallel(R, block(Q, S), NewProc),
  turnToFail(TmpSMap, S, NewSMap, St).
```

If compensation session is `active` the predicate `turnToFail` updates the state of session `S` and, as we will discuss in Section 5.3, the update can become `failed` or `compensated`. Otherwise, if the session is already `failed` or `compensated`, the predicate maintains its state. The extraction of stored compensations within inner transactions, while preserving protection blocks, is provided by the predicate `extr`. Notice that, in this implementation, this predicate also updates the state of the inner sessions, through the predicate `turnToFail`.

We are now in position to state the correspondence between the LTS defined in Section 2.3 and the Prolog interpretation described above.

Table 1
Mapping between LTL operators and their ASCII version counterparts, as used in the ProB tool.

LTL:	$\neg$	$\wedge$	$\vee$	$\Rightarrow$	X	U	F	G
ASCII ProB:	`not`	`&`	`or`	`=>`	`X`	`U`	`F`	`G`

Proposition 5.4. *Let $P \in \mathcal{P}$ be a process, where process variables X are only captured on current execution context, $f(P)$ its Prolog interpretation and φ a simplified correctness map, i.e., a map where each action is compensated by just one sequence of names. Given the LTS defined in Fig. 5 and the Prolog* `trans` *predicate as defined above, we have that $P \xrightarrow{\alpha} P'$ if and only if*

$$\texttt{trans(}\eta(\alpha)\texttt{, conf(}f(P)\texttt{, n(}P\texttt{), SMap, }\varphi\texttt{),}\\ \texttt{conf(}f(Q)\texttt{, n(}Q\texttt{), SMap', }\varphi\texttt{))},$$

with $Q \equiv P'$, and where `SMap` *and* `SMap'` *are the two session maps containing, for each session, its state and its set of compensation traces, before and after the labelled transition, respectively.*

Proof. The proof is by induction on P, then by case analysis on $\xrightarrow{\alpha}$ and `trans`. Although it follows straightforwardly, we should note that careful must be taken on α-renaming as it must be consistent, or at least made consistent through α-conversion, all along. □

5.2. Linear temporal logic

In the ProB tool we can express and check properties as Linear Temporal Logic (LTL) formulas. The LTL has been widely used for expressing correctness properties of concurrent programs [31] and, given a set of atomic propositions P, an LTL formula is inductively defined by the grammar (where $p \in P$):

$$\phi, \psi ::= \mathit{true} \mid p \mid (\phi) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \mid X\phi \mid \phi U \phi \mid G\phi \mid F\phi.$$

The basic boolean operators are the $\neg$ (negation) and the $\wedge$ (and). The others are abbreviations in the usual way: $\phi \vee \psi$ abbreviates $\neg(\neg\phi \wedge \neg\psi)$ and $\phi \Rightarrow \psi$ abbreviates $\neg\phi \vee \psi$. Also, the boolean constant *true* abbreviates $(\phi \vee \neg\phi)$, while *false* abbreviates $(\neg \mathit{true})$. With respect to the temporal operators, $\phi U \psi$ means that ψ does eventually hold and that ϕ will hold everywhere prior to ψ, and $X\phi$ holds now if and only if ϕ holds at the next moment. The other temporal operators are also abbreviations: $F\phi$, which means sometimes ϕ holds, abbreviates $\mathit{true} U \phi$ and $G\phi$, which means always ϕ holds, abbreviates $\neg F \neg \phi$. We refer the reader to the work by Emerson [12] for more details about LTL semantics and other subtle issues.

The above temporal operators assume that the underlying state space structure is a countable and totally ordered set, *i.e.*, isomorphic to the natural numbers with their usual ordering. The state spaces underlying our interpretations are however more complex. As usual with LTS, for each state we may have many possible transitions and, thus, our state spaces are directed graphs, *i.e.*, partially ordered sets. Notice that we have finite state spaces, because we did not allow replication. Nevertheless, we have many possible paths for each state and we say that an LTL formula ϕ is true if it holds for all possible paths. This is the usual approach when we embed LTL within CTL* [11], being equivalent to precede each LTL formula with the operator A from CTL*. This is also the assumption within the ProB model checker. As the following examples will be written in an ASCII-based LTL version, which is used in ProB, we provide a mapping between the usual LTL symbols and their ASCII version counterparts in Table 1.

As mentioned before, the states of our systems are fully described by the exchanged messages, *i.e.*, by the labels of the LTS. Therefore, our propositions are over the labels instead of the states and we consider an extended version of LTL supported by the ProB tool, herein named LTL^e. In contrast to standard LTL, LTL^e also supports propositions on transitions, not only on states. In practice, each atomic proposition is an instance of the operator `[regex]`, where `regex` is a regular expression over the LTS labels – see examples ahead.

5.3. Correctness map to LTL

In Section 4, we defined a correctness map as a function $\varphi : \mathcal{N} \longrightarrow 2^{\mathcal{N}^*}$. Here, for simplicity, we assume that a correctness map is simply a function $\varphi : \mathcal{N} \longrightarrow \{\{s\} : s \in \mathcal{N}^*\} \cup \{\emptyset\}$, *i.e.*, for each name we provide a single sequence of names or none. Although our results could be extended to the full definition of the correctness map, it would require a more evolved solution in what concerns data structures to ensure efficiency, which is beyond the scope of this paper.

Given a correctness map φ, we would like to express it as an LTL^e formula. It turns out that it is not easy. Even if we assume that, for a given execution trace leading to a failure, each compensation action is unique, *i.e.*, it is executed at most once, the notion of φ-correct for a well formed compensable process P with respect to a session $r \in \mathrm{sn}(P)$ would be expressed in

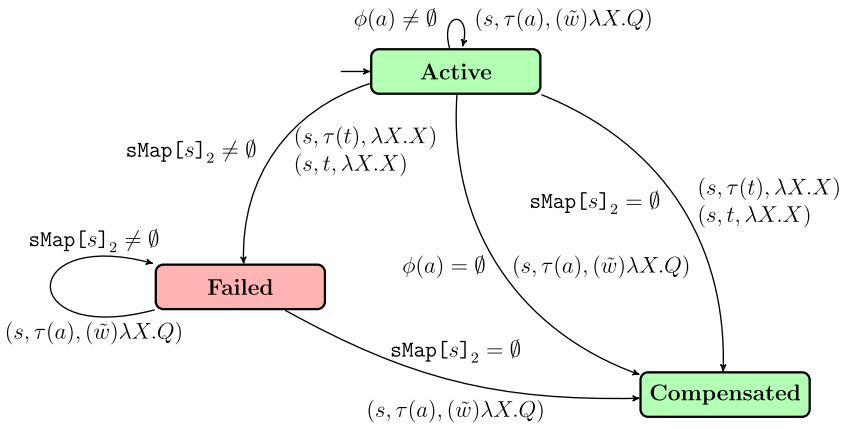


Fig. 14. Session states evolution, where squares denote session states and arrows denote possible transitions.

LTL^e as

$$\&_{a \in N_r} \&_{t \in T_r} \texttt{G([\_.}r\texttt{.tau.ch.}a\texttt{.\_)] =>}$$
$$\texttt{G(}\eta(\alpha_t)\texttt{ => F([\_.}r\texttt{.tau.ch.}a_1\texttt{.\_]…\& F([\_.}r\texttt{.tau.ch.}a_n\texttt{.\_)])...))),}$$

where $N_r \subseteq \mathrm{n}(P)$ is the set of names in P within session r, $T_r \subseteq \mathrm{tn}(P)$ is the set of transaction identifiers within session r, $\alpha_t \in \{(r, \tau(t), id), (r, t, id)\}$ and $\varphi(a) = a_1,\ldots, a_n$. Although this formula is already complex, we cannot assume that each name appears only once within a compensation execution. For instance, consider a generalisation of the case study where, instead we have only one client interacting with the shop and with the bank, we have several clients interacting with one or more shops and banks. Thus, we should generalise the above formula for expressing that, for all possible sequences of communications within a session, after a fault, one sequence of communications corresponding to the interleaving of the compensation paths given by the correctness map would occur. In such case, the formula would not only be complex, but hard to check by the model checker.

To overcome the above issue, we have added in our process configuration two components, namely the correctness map and a session map. The correctness map φ is represented in the process configuration through an association list `CMap`, associating each free name $a \in N$ (the key) to a list that represents the compensation trace $\varphi(a)$ (the value). The session map is also an association list, that at each moment contains the state of the session (`active`, `failed` or `compensated`) and the compensation traces that should be executed in case of failure. Thus the session map associates the name of each session (the key) to a pair (the value) composed by the current state of the session and the list of accumulated sets of compensation traces, that have to be executed in case of failure. We have also added to the label the state `St` of the session, information that will be useful for verifying if a process is correct with respect to a correctness map – see ahead.

The rules that update the session map `SMap` are L-COM, L-RECOVER-IN and L-RECOVER-OUT. Fig. 14 depicts the transitions that change the session state or the `SMap` structure. The remaining transitions do not change either the session state or the `SMap` structure.

When a session s is active, the communication within that session may change the session map or not. For instance, if the label is $(s, \tau(a), \lambda X.Q)$ and $\varphi(a) = \{\epsilon\}$ (represented by `[ ]`), the session map remains the same, as well as the session state. However, if $\varphi(a) = \emptyset$ (represented by `clean`) all the compensation traces of the session are removed and the state of the session changes to compensated, as we can see in Fig. 14. On the other hand, if $\varphi(a)$ is a singular set with the non empty sequence ($\varphi(a) \neq \{\epsilon\}$ and $\varphi(a) \neq \emptyset$), it is added to the session map. These updates are made by the predicate `updateSession`.

The transitions made by rules L-RECOVER-IN or L-RECOVER-OUT may change the state of an active session s, through the predicate `turnToFail`.

```
turnToFail(SMap, S, NewSMap, St) :-
  avl:avl_fetch(S, SMap, Value),
  getProj2(Value, X),
  getProj1(Value, Y),
  ( (Y == failed,
     NewSMap = SMap,
     St = failed)
  ; (Y == compensated,
     NewSMap = SMap,
     St = compensated)
  ; (Y == active,
```

```
    lenght(X, 0),
    avl:avl_change(S, SMap, Value, NewSMap,
           pair(compensated,[])),
    St = compensated)
; (Y == active,
    lenght(X, N),
    N >= 1,
    avl:avl_change(S, SMap, Value, NewSMap,
           pair(failed, X)),
    St = failed)
).
```

These rules also change the state of the active sessions that are totally or partially inner sessions of s. To do this, the predicate `extract`, that is used in both rules, calls the predicate `turnToFail`. This predicate checks if there is any compensation trace for the corresponding session. If there is at least one, the state of the session changes to `failed` (see Fig. 14 when $SMap[s]_2 = \emptyset$), otherwise it changes to `compensated`. If the session state is `failed`, the state of the session stays `failed` until all the accumulated compensation traces in the session map are executed, *i.e.*, all the communications corresponding to those traces occur within the session (recall that the session of the communication is always the session of the input). After all compensation traces are executed, the session becomes compensated and the corresponding map is updated.

Therefore, the notion of φ-correct for a well formed compensable process P, with respect to a session $r \in \mathrm{sn}(P)$, can be expressed in LTL^e simply as

```
G([label.failed.r._] =>  F([label.compensated.r._])),
```

where _ stands for any possible label suffix (the operator _ belongs to the LTL^e syntax).

5.4. Case study

Let us now check the case study for its correctness. We will discuss each of the three examples, Figs. 7–9. For instance, the correctness map for the first example is defined as

```
CMap[pack]=[unpacked]
CMap[valid]=[refunded]
```

```
 1. label(active,r0,tau,ch(client),[],hole,[]).
 2. label(active,r0,tau,ch(ack),[],hole,[]).
 3. label(active,r1,tau,ch(initchg),[],hole,[]).
 4. label(active,r1,tau,ch(bank),[],hole,[]).
 5. label(active,r2,tau,ch(initpck),[],hole,[]).
 6. label(active,r2,tau,ch(pack),[],hole,[]).
 7. label(compensated,r1,tau,ch(y),[],hole,[]).
 8. label(active,r1,tau,ch(valid),[],hole,[]).
 9. label(active,r0,tau,ch(recp),[],hole,[]).
10. label(active,r0,tau,ch(okShop),[],hole,[]).
11. label(active,r1,tau,ch(ok),[],hole,[]).
12. label(active,r1,tau,ch(end1),[],hole,[]).
13. label(compensated,r2,tau,ch(x),[],hole,[]).
14. label(active,r2,tau,ch(notExists),[],hole,[]).
15. label(compensated,r0,tau,ti(t),[],hole,[]).
16. label(failed,r1,tau,ti(q),[],hole,[]).
17. label(compensated,r2,tau,ti(k),[],hole,[]).
```

Fig. 15. A counterexample for the example in Fig. 8, showing that it is not φ-correct, considering the φ defined for this process – see text. Note the state `failed` for session `r1` (line 16).

Table 2
ProB statistics for the case study verification.

Case	# atoms	# transitions	Time (s)	Memory (MB)
1	695	734	8.01	1.25
2	36,753	7211	3722.02	2.00
3	37,403	65,900	9818.84	2.25

```
CMap[ok]=clean
CMap[yes]=clean
```

and the other free names mapped to `[ ]`, that represents the empty sequence. To verify that this example is correct, we must verify the formula

```
G([label.failed.r0._] => F([label.compensated.r0._])) &
G([label.failed.r1._] => F([label.compensated.r1._])).
```

We checked this formula and the result was the expected, *i.e.*, true. The same correctness map was used the verify that the example in Fig. 9 was correct. With respect to the example in Fig. 8, discussed in Section 4, a correctness map should be defined as

```
CMap[pack]=[unpacked]
CMap[valid]=[refunded]
```

and the other free names mapped to `[ ]`. In this case, the formula to be verified is

```
G([label.failed.r0._] => F([label.compensated.r0._])) &
G([label.failed.r1._] => F([label.compensated.r1._])) &
G([label.failed.r2._] => F([label.compensated.r2._])).
```

However, this is not true, as it is confirmed by the counterexample trace, provided by ProB, which can be seen in Fig. 15. This counterexample corresponds to the case when the client has paid but has not received the package. As mentioned above, we checked the correctness of the three examples with the ProB tool. In Table 2, we provide some statistics about the state space, memory requirements and running time. The interpreter and examples can be downloaded from http://pwp.net.ipl.pt/cc.isel/cvaz/dcpi/.

6. Related work and concluding remarks

We have developed a notion of compensation correctness over a process calculus suitable for modelling long running transactions with a recovery mechanism based on compensations. Moreover, in this setting, we discuss correctness criteria for compensable process compositions.

As mentioned, the process calculus presented in this paper is based on $\mathtt{dc}\pi$-calculus [35] and on the calculus presented in [19], focusing on the relevant primitives for reasoning about the correctness of compensations. These three calculi were built around several ideas, such as primitives for partially recovery and compensation installation, found in previous works. Bocchi et al. introduced πt-calculus [1], which supports compensation definition and is inspired by BizTalk, consisting of an extension of asynchronous polyadic π-calculus [32] with the notion of transactions. In this case compensations are statically defined, *i.e.*, they are not incrementally built. This is also the case with cJoin calculus [4], in which completed transactions cannot be compensated. Butler and Ferreira [7] propose the StAC language, which is inspired by BPBeans. The language includes the notion of compensation pair, similar to the sagas concept defined by Gargia-Molina and Salem [13]. In this language, a long running transaction is seen as a composition of one or more subtransactions, where each of them has an associated compensation. In contrast to our calculus, StAC is flow composition based and includes explicit operators for running or discarding installed compensations. Compensating CSP [8], denoted by cCSP, and Sagas calculi [5] are also examples of composition flow based calculi, namely they adopt a centralised coordination mechanism, but with different compensation policies. More related with our approach, we have $\mathtt{web}\pi$ [20], and its untimed version known as $\mathtt{web}\pi_{\infty}$ [25], with which we share some syntax similarities, although we have followed different principles. Namely, in both calculi the nested transactions are flattened and, hence, these calculi do not provide built in nested abortion. These calculi assume also that completed transactions cannot be compensated and, as in πt-calculus, support only statically defined compensations. SOCK [14] and COWS [22] are also examples of calculi supporting compensation handling through explicitly constructs. Notably, COWS adds the notion of protection block and fault signalling of a process within a scope. Finally, CaSPIS [2], a session based calculi, also includes primitives for handling the termination of interacting partners and a mechanism to statically program compensations by means of listeners.

There are also other approaches for reasoning about the correctness of compensations. Korth et al. [18] have defined compensation soundness in terms of the properties that compensations have to guarantee. The correctness notions are based on the existence of state and on state equivalence. Nevertheless, the authors do not provide a formal framework for specifying their definitions. Caires et al. [9] proposed a formal framework to reason about correctness of compensating transactions, which is also based on the existence of an appropriate notion of equivalence on system states. Even though their approach supports distributed transactions, the compensable processes do not interact. A different approach is given by Butler et al. [8], that proposes a notion of compensation soundness and a stateless equivalence notion. They define a cancellation semantics based on a cancellation function that analyses traces, extracting forward and compensation (reverse) actions from process traces.

In what concerns primitives for instantiation and definition of multiparty asynchronous sessions, we consider the approach proposed by Honda et al. [16] the most interesting one. Such primitives allow us to initiate new sessions and ensure the generation of fresh names and, in particular, fresh transaction identifiers. As mentioned before, the inclusion of input guarded replication in our calculus depends on the inclusion of primitives for session and transaction initiation. In such setting, the results presented herein hold if we track all session names and if we disallow also α-renaming for sessions after initiation. Note that this is required to ensure unambiguity on tracking session communications. A second aspect that we must consider when including input guarded replication is that correctness maps are defined over free names. Nevertheless, this turns out not to be a problem since transition labels are session aware and, hence, we are able to distinguish communications on a same name but within different sessions.

Due to the stateless assumption of the service oriented paradigm, our correctness criteria are not based on the existence of an equivalence notion on states. Hence, the proposed criteria could be used on other paradigms based on a minimal shared knowledge among the interacting parts.

Regarding future work, we plan to develop a type system to guarantee the properties needed to ensure correctness of compensable transactions. We plan also to investigate the use of our correctness criteria and setting on other calculi with a recovery mechanism based on compensations [1,4,15,21,25].

Acknowledgements

We thank to Michael Leuschel, Jens Bendisposto and Daniel Plagge for their support on the ProB tool. We thank also anonymous reviewers for helpful suggestions that improved the clarity of this manuscript. Cátia Vaz was partially supported by the Portuguese FCT, via SFRH/BD/45572/2008. This work was also partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

References

[1] L. Bocchi, C. Laneve, G. Zavattaro, A calculus for long-running transactions, in: FMOODS, LNCS, vol. 2884, Springer, 2003, pp. 124–138.
[2] M. Boreale, R. Bruni, R. De Nicola, M. Loreti, Sessions and pipelines for structured service programming, in: Proc. of FMOODS'08, LNCS, vol. 5051, Springer, 2008, pp. 19–38.
[3] I. Bratko, Prolog Programming for Artificial Intelligence, Addison-Wesley, 2001.
[4] R. Bruni, H.C. Melgratti, U. Montanari, Nested commits for mobile calculi: extending join, in: IFIP TCS, Kluwer, 2004, pp. 563–576.
[5] R. Bruni, H.C. Melgratti, U. Montanari, Theoretical foundations for compensations in flow composition languages, in: POPL, ACM, 2005, pp. 209–220.
[6] R. Bruni, M.J. Butler, C. Ferreira, C.A.R. Hoare, H.C. Melgratti, U. Montanari, Comparing two approaches to compensable flow composition, in: CONCUR, LNCS, vol. 3653, 2005, pp. 383–397.
[7] M.J. Butler, C. Ferreira, An operational semantics for StAC, a language for modelling long-running business transactions, in: Proc. of COORDINATION'04, LNCS, vol. 2949, Springer, 2004, pp. 87–104.
[8] M.J. Butler, C.A.R. Hoare, C. Ferreira, A trace semantics for long-running transactions, in: 25 Years Communicating Sequential Processes, LNCS, vol. 3525, Springer, 2004, pp. 133–150.
[9] L. Caires, C. Ferreira, H.T. Vieira, A process calculus analysis of compensations, in: TGC, LNCS, vol. 5474, Springer, 2008, pp. 87–103.
[10] M. Carbone, K. Honda, N. Yoshida, Structured interactional exceptions for session types, in: CONCUR, LNCS, Springer, 2008, pp. 402–417.
[11] E.M. Clarke, I.A. Draghicescu, Expressibility results for linear-time and branching-time logics, in: REX Workshop, LNCS, vol. 354, Springer, 1988, pp. 428–437.
[12] E.A. Emerson, Temporal and modal logic, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B), MIT Press, 1990, pp. 995–1072.
[13] H. Garcia-Molina, K. Salem, Sagas, in: U. Dayal, I.L. Traiger (Eds.), SIGMOD Conference, ACM Press, 1987, pp. 249–259.
[14] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, G. Zavattaro, A calculus for service oriented computing, in: A. Dan, W. Lamersdorf (Eds.), ICSOC, LNCS, vol. 4294, Springer, 2006, pp. 327–338.
[15] C. Guidi, I. Lanese, F. Montesi, G. Zavattaro, On the interplay between fault handling and request–response service invocations, in: ACSD, IEEE Computer Society Press, 2008, pp. 190–199.
[16] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: POPL, ACM, 2008, pp. 273–284.
[17] IBM, Web Services Flow Language v1.0, 2001. Available from: <http://xml.coverpages.org/WSFL-Guide-200110.pdf>.
[18] H.F. Korth, E. Levy, A. Silberschatz, A formal approach to recovery by compensating transactions, in: VLDB, 1990, pp. 95–106.
[19] I. Lanese, C. Vaz, C. Ferreira, On the expressive power of primitives for compensation handling, in: ESOP, LNCS, vol. 6012, Springer, 2010, pp. 366–386.
[20] C. Laneve, G. Zavattaro, Foundations of web transactions, in: V. Sassone (Ed.), FoSSaCS, LNCS, vol. 3441, Springer, 2005, pp. 282–298.
[21] C. Laneve, G. Zavattaro, Foundations of web transactions, in: FoSSaCS, LNCS, vol. 3441, Springer, 2005, pp. 282–298.
[22] A. Lapadula, R. Pugliese, F. Tiezzi, A calculus for orchestration of web services, in: Proc. of ESOP'07, LNCS, vol. 4421, Springer, 2007, pp. 33–47.
[23] M. Leuschel, M.J. Butler, ProB: an automated analysis toolset for the B method, Int. J. Softw. Tools Technol. Transf. 10 (2) (2008) 185–203.
[24] R. Lucchi, M. Mazzara, A pi-calculus based semantics for WS-BPEL, J. Logic Algebr. Program. 70 (1) (2007) 96–118.
[25] M. Mazzara, I. Lanese, Towards a unifying theory for web services composition, in: WS-FM, LNCS, vol. 4184, Springer, 2006, pp. 257–272.
[26] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I and II, Inform. and Comput. 100 (1) (1992) 1–77.
[27] Oasis, Web Services Description Language v2.0, 2007. Available from: <http://www.w3.org/TR/wsdl20/>.
[28] Oasis, Web Services Business Process Execution Language v2.0, 2007. Available from: <http://docs.oasis-open.org/wsbpel/2.0/>.
[29] OMG, Business Process Model and Notation (BPMN) v2.0, 2011. Available from: <http://www.omg.org/spec/BPMN/2.0/>.
[30] M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computing: a research roadmap, Int. J. Cooperative Inf. Syst. 17 (2) (2008) 223–255.
[31] A. Pnueli, The temporal logic of programs, in: FOCS, IEEE Computer Society, 1977, pp. 46–57.
[32] D. Sangiorgi, D. Walker, Pi-Calculus: A Theory of Mobile Processes, Cambridge University Press, 2001.
[33] S. Thatte, XLANG: Web Services for Business Process Design, Tech. Rep., Microsoft Corporation, 2001.
[34] C. Vaz, C. Ferreira, Towards compensation correctness in interactive systems, in: International Workshop on Web Services and Formal Methods, WS-FM 2009, Lecture Notes in Computer Science 6194, Springer, pp. 161–177.
[35] C. Vaz, C. Ferreira, A. Ravara, Dynamic recovering of long running transactions, in: TGC, LNCS, vol. 5474, Springer, 2008, pp. 201–215.
[36] W3C, SOAP v1.2 Part 1: Messaging Framework, second ed., 2007. Available from: <http://www.w3.org/TR/soap12-part1/>.
[37] W3C, Web Services Choreography Description Language v1.0, 2007. Available from: <http://www.w3.org/TR/ws-cdl-10/>.