Decidability Results for Dynamic Installation of Compensation Handlers

Ivan Lanese, Gianluigi Zavattaro

Decidability Results for Dynamic Installation of Compensation Handlers

Ivan Lanese and Gianluigi Zavattaro

Focus Team, University of Bologna & INRIA, Italy

Abstract. Dynamic compensation installation allows for easier specification of fault handling in complex interactive systems since it enables to update the compensation policies according to run-time information. In this paper we show that in a simple π-like calculus with static compensations the termination of a process is decidable, but it is undecidable in one with dynamic compensations. We then consider three commonly used patterns for dynamic compensations, showing that process termination is decidable for parallel and replacing compensations while it remains undecidable for nested compensations.

1 Introduction

Nowadays, applications are composed of different interacting entities, living in environments such as the Internet or the cloud. Programming applications in this setting is challenging, due to their own complexity, and on the unpredictability of the environment. For instance, a communication partner may disappear during an interaction, or a message may be lost due to an unreliable network. Nevertheless, the users expect their applications to always provide reliable services. To build reliable services in an unreliable environment coping with unexpected events is certainly one of the main challenges.

In the setting of service-oriented computing, *long running transactions* have been put forward to solve this problem. A long running transaction is a computation that either *succeeds*, or it *aborts*. In the second case, a *compensation* is executed to undo unwanted side effects of the aborted computation. Many languages provide nowadays support for long running transactions [25, 26], and different proposals exist in the literature [2, 3, 8–12, 16, 22]. Originally, the compensation of a long running transaction was statically fixed [26]. Recent proposals show however that being able to dynamically update the compensation as far as the computation progresses allows the programmer to write more easily the compensation code for complex interactions [16].

From a language design point of view, the question of whether dynamic compensations are just syntactic sugar, and thus need not be implemented in the core language, or not, is relevant. Strangely, while many papers in the literature put forward proposals of transaction constructs, very little has been done on comparing them. A main work in this direction is [20]. In [20] it is shown that the ability to add new compensation items to be executed in parallel with the

static compensation does not increase the expressive power, while more general patterns do. The study is carried out relying on proofs of encodability and/or non-encodability between the different formalisms. However, there is no clear agreement in the community on which conditions such encodings should satisfy, and the results in [20] strongly depend both on the chosen conditions and on the availability of suitable mechanisms in the compared languages.

We want here to tackle the same problem, but with a completely different approach. In fact, we compare π-like core calculi featuring the basic mechanisms for static and dynamic compensations according to the (un)decidability of *process termination*, that is of the absence of an infinite computation starting from a given process. Clearly, calculi where such a property is undecidable cannot be encoded in calculi where the same property is decidable, and this non-encodability result is valid for all the encodings preserving the property.

We show that process termination is decidable in a π-calculus with static compensations, while it is not in one with dynamic compensations. To better understand where this difference stems from, we limit the expressive power of the dynamic compensation mechanism in different directions. We show that if compensations can only be replaced, then decidability is recovered. If instead compensations can be nested using linear patterns, we are still in an undecidable setting. To further constrain linear patterns aiming at decidability we force the patterns to only add new compensation items in parallel, obtaining again a decidability result.

2 Primitives for Compensations

2.1 Syntax

We base our studies on a π-calculus extended with transactions and primitives for compensation installation. We then consider different fragments, corresponding to various compensation installation patterns. Our calculus is similar to the calculus in [20]. A main difference is that we do not consider restriction. This is forced since, if we add restriction, then process termination (and similar properties) become undecidable even in CCS (without transactions).

The syntax of our calculus relies on a countable set of names N, ranged over by lower case letters. We use $\boldsymbol{x}$ to denote a tuple $x_1, \cdots, x_n$ of names, and $\{\boldsymbol{x}\}$ denotes the set of elements in the tuple. We use $\{\boldsymbol{v}/\boldsymbol{x}\}$ for denoting the substitution of names in $\boldsymbol{v}$ for names in $\boldsymbol{x}$, and we use a similar notation for substitutions of processes for process variables (introduced later).

We start by presenting the syntax of the π-calculus, reported in Fig. 1. Prefixes can be either outputs $\overline{a}\langle\boldsymbol{v}\rangle$ of a tuple of names $\boldsymbol{v}$ on channel a, or corresponding inputs $a(\boldsymbol{x})$, receiving a tuple of names $\boldsymbol{v}$ on channel a and applying substitution $\{\boldsymbol{v}/\boldsymbol{x}\}$ to the continuation. The π-calculus syntax includes the inactive process $\mathbf{0}$, guarded choice $\sum_{i\in I}\pi_i.P_i$, guarded replication $!\pi.P$ and parallel composition $P \mid Q$. We write $\overline{a}$ for $\overline{a}\langle\boldsymbol{v}\rangle$ when $\boldsymbol{v}$ is empty, and a for $a(\boldsymbol{x})$ when $\boldsymbol{x}$ is empty. When I is a singleton, $\sum_{i\in I}\pi_i.P_i$ is shortened into $\pi_i.P_i$. We may also drop trailing $\mathbf{0}$s.

$$
\begin{array}{rll}
\pi ::= & \pi\text{-calculus prefixes} & \\
& \overline{a}\langle \boldsymbol{v} \rangle \text{ (Output prefix)} & \mid \; a(\boldsymbol{x}) \quad \text{(Input prefix)} \\
P, Q ::= & \pi\text{-calculus processes} & \\
& \mathbf{0} \quad \text{(Inaction)} & \mid \; \sum_{i \in I} \pi_i.P_i \quad \text{(Guarded choice)} \\
\mid & !\pi.P \quad \text{(Guarded replication)} & \mid \; P \mid Q \quad \text{(Parallel composition)}
\end{array}
$$

Fig. 1. π-calculus processes.

$$
\begin{array}{rll}
P, Q ::= & \textit{Static compensation processes} & \\
& \ldots & (\pi\text{-calculus processes}) \\
\mid & t[P, Q] & \text{(Transaction scope)} \\
\mid & \langle P \rangle & \text{(Protected block)}
\end{array}
$$

Fig. 2. Static compensation processes.

$$
\begin{array}{rll}
P, Q ::= & \textit{Dynamic compensation processes} & \\
& \ldots & \text{(Static compensation processes)} \\
\mid & X & \text{(Process variable)} \\
\mid & \mathsf{inst}\lfloor \lambda X.Q \rfloor.P & \text{(Compensation update)}
\end{array}
$$

Fig. 3. Dynamic compensation processes.

We now extend the π-calculus with transactions and *static compensations*. The syntax of the extended calculus is in Fig. 2. Static compensations can be programmed by adding just two constructs to π-calculus: *transaction scope* and *protected block*. A transaction scope $t[P, Q]$ behaves as process P until an error is notified to it by an output $\overline{t}$ on the name t of the transaction scope. When such a notification is received the transaction atomically *aborts*: the body P of the transaction scope is killed and compensation Q is executed. Q is executed inside a protected block. In this way it will not be influenced by successive external errors. Error notifications may be generated both from the body P of the transaction scope and from external processes. Error notifications are simply output messages (without parameters). Protected block $\langle P \rangle$ behaves as process P, but it is not killed in case of failure of a transaction scope enclosing it.

The calculus with *dynamic compensations* extends the one with static compensations. The main difference is that with dynamic compensations the body P of transaction scope $t[P, Q]$ can update the compensation Q. *Compensation update* is performed by an additional operator $\mathsf{inst}\lfloor \lambda X.Q' \rfloor.P'$, where function $\lambda X.Q'$ is the compensation update (X can occur inside Q'). Applying such a compensation update to compensation Q produces a new compensation $Q'\{^{Q}/_{X}\}$. Note that Q may not occur at all in the resulting compensation, and it may also occur more than once. For instance, $\lambda X.\mathbf{0}$ deletes the current compensation. The syntax of processes with dynamic compensations extends the one of processes

with static compensations with the compensation update operator and process variables (see Fig. 3). We use X to range over process variables.

We define for processes with dynamic compensations the usual notions of free and bound names. Names in $\boldsymbol{x}$ are bound in $a(\boldsymbol{x}).P$. Other names are free. Also, variable X is bound in $\lambda X.Q$. Bound names and variables inside processes can be α-converted as usual. We consider only processes with no free variables.

Processes with static compensations are processes with dynamic compensations where the compensation update operator is never used. We will show that dynamic compensations are very expressive, making relevant properties undecidable. Thus we consider different subcalculi, constraining the allowed patterns for compensation installation. As a first observation, note that in a compensation update of the form $\lambda X.Q$ there are no constraints on how many times X may occur in Q. Having more than one occurrence of X, allowing to replicate the previous compensation, is rarely used in practice. Thus a meaningful restriction is considering just linear compensations, where X occurs exactly once in Q. We call them *nested compensations*, since the old compensation becomes nested inside the new one, which acts as a context. Another relevant case is when X does not occur at all in Q. We call compensations of this form *replacing compensations*, since the new compensation completely replaces the old one, which is discarded. Finally, a relevant subcase of nested compensations are *parallel compensations*, where Q has the form $Q' \mid X$ and X does not occur in Q'. In this case new and old compensation items are in parallel in the final term.

2.2 Operational Semantics

In this section we define the operational semantics of processes with dynamic compensations. We need however an auxiliary definition. When a transaction scope $t[P, Q]$ is killed, part of its body P may be preserved, in particular the protected blocks inside it.

The definition of function $\mathsf{extr}(P)$ computing the part of process P to be preserved depends on the meaning of nesting of transaction scopes. In the literature, three main approaches are considered. When the enclosing transaction scope is killed, its subtransactions may be *aborted*, *preserved* or *discarded*. The aborting semantics is used by SAGAs calculi [9], WS-BPEL [26], and others. The preserving semantics is, for instance, the approach of Webπ [22]. Finally, the discarding semantics has been proposed by ATc [3] and TransCCS [12]. We consider all the three possibilities, since they just differ in the definition of function $\mathsf{extr}(\bullet)$.

Definition 1 (Extraction function). *We denote the functions corresponding to aborting, preserving, and discarding semantics for transaction nesting respectively as* $\mathsf{extr_a}(\bullet)$, $\mathsf{extr_p}(\bullet)$ *and* $\mathsf{extr_d}(\bullet)$. *The function* $\mathsf{extr_a}(\bullet)$ *is defined in Fig. 4. The definition of function* $\mathsf{extr_p}(\bullet)$ *is the same but for the clause for transaction scope, which is replaced by the clause* $\mathsf{extr_p}(t\,[P, Q]) = t\,[P, Q]$. *The definition of function* $\mathsf{extr_d}(\bullet)$ *instead is obtained by replacing the clause for transaction scope by the clause* $\mathsf{extr_d}(t\,[P, Q]) = \mathbf{0}$.

$$\begin{aligned}
\mathsf{extr_a}(\mathbf{0}) &= \mathbf{0} & \mathsf{extr_a}(\langle P\rangle) &= \langle P\rangle\\
\mathsf{extr_a}(\textstyle\sum_{i\in I}\pi_i.P_i) &= \mathbf{0} & \mathsf{extr_a}(t\,[P,Q]) &= \mathsf{extr_a}(P)\mid\langle Q\rangle\\
\mathsf{extr_a}(!\pi.P) &= \mathbf{0} & \mathsf{extr_a}(P\mid Q) &= \mathsf{extr_a}(P)\mid\mathsf{extr_a}(Q)\\
\mathsf{extr_a}(\mathsf{inst}\lfloor\lambda X.Q\rfloor.P) &= \mathbf{0}
\end{aligned}$$

Fig. 4. Extraction function for aborting semantics.

$$\text{(P-Out)}\ \frac{}{\overline{a}\langle\boldsymbol{v}\rangle.P \xrightarrow{\overline{a}\langle\boldsymbol{v}\rangle} P} \qquad \text{(P-In)}\ \frac{}{a(\boldsymbol{x}).P \xrightarrow{a(\boldsymbol{v})} P\{{}^{\boldsymbol{v}}/_{\boldsymbol{x}}\}} \qquad \text{(L-Choice)}\ \frac{\pi_j.P_j \xrightarrow{\alpha} P'_j \quad j\in I}{\sum_{i\in I}\pi_i.P_i \xrightarrow{\alpha} P'_j}$$

$$\text{(L-Rep)}\ \frac{\pi.P \xrightarrow{\alpha} P'}{!\pi.P \xrightarrow{\alpha} P'|!\pi.P} \qquad \text{(L-Par)}\ \frac{P \xrightarrow{\alpha} P'}{P\mid Q \xrightarrow{\alpha} P'\mid Q} \qquad \text{(L-Synch)}\ \frac{P \xrightarrow{x(\boldsymbol{v})} P' \quad Q \xrightarrow{\overline{x}\langle\boldsymbol{v}\rangle} Q'}{P\mid Q \xrightarrow{\tau} P'\mid Q'}$$

$$\text{(L-Scope-out)}\ \frac{P \xrightarrow{\alpha} P' \quad \alpha \neq \lambda X.Q}{t[P,Q] \xrightarrow{\alpha} t[P',Q]} \qquad \text{(L-Recover-out)}\ \frac{}{t[P,Q] \xrightarrow{t} \mathsf{extr_a}(P)\mid\langle Q\rangle} \qquad \text{(L-Recover-in)}\ \frac{P \xrightarrow{\overline{t}} P'}{t[P,Q] \xrightarrow{\tau} \mathsf{extr_a}(P')\mid\langle Q\rangle}$$

$$\text{(L-Inst)}\ \frac{}{\mathsf{inst}\lfloor\lambda X.Q\rfloor.P \xrightarrow{\lambda X.Q} P} \qquad \text{(L-Scope-inst)}\ \frac{P \xrightarrow{\lambda X.R} P'}{t[P,Q] \xrightarrow{\tau} t[P',R\{{}^{Q}/_{X}\}]} \qquad \text{(L-Block)}\ \frac{P \xrightarrow{\alpha} P'}{\langle P\rangle \xrightarrow{\alpha} \langle P'\rangle}$$

Fig. 5. LTS for dynamic compensation processes.

The operational semantics of dynamic compensations and, implicitly, of static, replacing, parallel and nested compensation processes, is defined below.

We use $a(\boldsymbol{v})$, $\overline{a}\langle\boldsymbol{v}\rangle$, τ and $\lambda X.Q$ as labels, and we use α to range over labels. The first three forms of labels are as in π-calculus: $a(\boldsymbol{v})$ is the input of a tuple of values $\boldsymbol{v}$ on channel a, $\overline{a}\langle\boldsymbol{v}\rangle$ is the corresponding output, and τ is an internal action. However, an output label without parameters can also be used for error notification, and an input without parameters for receiving the notification. The last label, $\lambda X.Q$, is specific of dynamic compensation processes and corresponds to compensation update. We write a for $a(\boldsymbol{v})$ and $\overline{a}$ for $\overline{a}\langle\boldsymbol{v}\rangle$ if $\boldsymbol{v}$ is empty. We may use t instead of a to emphasize that the name is used for error notification.

Definition 2 (Operational semantics). *The operational semantics of dynamic compensation processes with aborting semantics for transaction nesting is the minimum LTS closed under the rules in Fig. 5 (symmetric rules are considered for (*L-Par*) and (*L-Synch*)). The preserving semantics (resp. discarding semantics) is obtained by replacing function* $\mathsf{extr_a}(\bullet)$ *with* $\mathsf{extr_p}(\bullet)$ *(resp.* $\mathsf{extr_d}(\bullet)$*).*

The first six rules are standard π-calculus rules [23], the others define the behavior of transactions, compensations and protected blocks.

Auxiliary rules (P-Out) and (P-In) execute output and input prefixes, respectively. The input rule guesses the received values $\boldsymbol{v}$ in the early style. Rules (L-Choice) and (L-Rep) deal with guarded choice and replication, respectively. Rule (L-Par) allows one of the components of parallel composition to progress while the other one stays idle. Rule (L-Synch) performs communication, synchronizing an input $x(\boldsymbol{v})$ and a corresponding output $\overline{x}\langle\boldsymbol{v}\rangle$.

Rule (L-Scope-out) allows the body P of a transaction scope to progress, provided that the performed action is not a compensation update. Rule (L-Recover-out) allows external processes to abort a transaction scope via an output $\overline{t}$. The resulting process is composed of two parts: the first one extracted from the body P of the transaction scope, and the second one corresponding to compensation Q, which will be executed inside a protected block. Rule (L-Recover-in) is similar to (L-Recover-out), but now the error notification comes from the body P of the transaction scope. Rule (L-Inst) requires to perform a compensation update. Rule (L-Scope-inst) updates the compensation of a transaction scope. Finally, rule (L-Block) defines the behavior of protected blocks. The property of protected blocks of being unaffected by external aborts is enforced by the definition of function $\mathsf{extr}(\bullet)$.

In the following we consider a structural congruence $\equiv$ to rearrange the order of parallel processes and to garbage collect process $\mathbf{0}$. Formally, $\equiv$ is the least congruence such that $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$ and $P \mid \mathbf{0} \equiv P$.

As discussed in the Introduction, we will consider the (un)decidability of process termination: a process P terminates if there exists no infinite sequence of processes $P_1, P_2, \ldots, P_i, \ldots$ such that $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} P_2 \xrightarrow{\tau} \ldots \xrightarrow{\tau} P_i \xrightarrow{\tau} \ldots$.

Example 1. We give here a few examples of transitions.

- Transaction scopes can compute:
  $\overline{a}\langle b\rangle \mid t[a(x).\overline{x}.0, Q] \xrightarrow{\tau} t[\overline{b}.0, Q]$
- Transaction scopes can be killed:
  $\overline{t} \mid t[\overline{a}.0, Q] \xrightarrow{\tau} \langle Q\rangle$
- Transaction scopes can commit suicide:
  $t[\,\overline{t}.0 \mid \overline{a}.0, Q] \xrightarrow{\tau} \langle Q\rangle$
- Protected blocks survive after kill:
  $t[\,\overline{t}.0 \mid \langle\overline{a}.0\rangle, Q] \xrightarrow{\tau} \langle\overline{a}.0\rangle \mid \langle Q\rangle$
- New compensation items can be added in parallel:
  $t[\mathsf{inst}\lfloor\lambda X.P|X\rfloor.\overline{a}.0, Q] \xrightarrow{\tau} t[\overline{a}.0, P|Q]$
- New compensation items can be added at the beginning:
  $t[\mathsf{inst}\lfloor\lambda X.\overline{b}.X\rfloor.\overline{a}.0, Q] \xrightarrow{\tau} t[\overline{a}.0, \overline{b}.Q]$
- Compensations can be deleted:
  $t[\mathsf{inst}\lfloor\lambda X.0\rfloor.\overline{a}.0, Q] \xrightarrow{\tau} t[\overline{a}.0, 0]$

3 Termination Undecidability for Nested Compensations

We now move to the proof of undecidability of termination in the calculus with nested compensations. This contrasts with the decidability of termination for static compensations (the proof of this result is deferred to Corollary 2).

The undecidability proof is by reduction from the termination problem in Random Access Machines (RAMs) [24], a well-known Turing powerful formalism based on registers containing non-negative natural numbers. The registers are used by a program, that is a set of indexed instructions I_i of two possible kinds:

- $i : Inc(r_j)$ that increments the register r_j and then moves to the execution of the instruction with index $i + 1$ and
- $i : DecJump(r_j, s)$ that attempts to decrement the register r_j; if the register does not hold 0 then the register is actually decremented and the next instruction is the one with index $i + 1$, otherwise registers are unchanged and the next instruction is the one with index s.

We assume that given a program $I_1, \cdots, I_n$, it starts by executing I_1. It terminates when an undefined program instruction is reached. Since the computational model is Turing complete, the termination of a RAM program is undecidable.

We encode RAMs as follows. Each register r_j containing the value n is encoded as a transaction $r_j[R_j, Q_j]$ where Q_j is a process $\overline{u}.\overline{u}.\cdots.\overline{u}.\overline{z}$ with exactly n prefixes $\overline{u}$. The process R_j is responsible for updating its compensation Q_j by performing $\mathsf{inst}\lfloor\lambda X.\overline{u}.X\rfloor$ every time the register must be incremented. Each instruction I_i will be encoded as a process $!p_i.P_i$: the instruction will be activated by $\overline{p_i}$ and then P_i will be performed. If $i : Inc(r_j)$ is an increment instruction on r_j, P_i will interact with R_j in order to activate the update of its compensation Q_j. If $i : DecJump(r_j, s)$ is a decrement/jump instruction, on the other hand, P_i will terminate the transaction r_j so that the compensation Q_j becomes active. If Q_j is $\overline{z}$ then the value of the register is 0. In this case a new instance of the register $r_j[R_j, \overline{z}]$ is spawn and the jump is executed. If Q_j is $\overline{u}.\cdots.\overline{z}$ then the register is not 0. In this case, a new instance of the register $r_j[R_j, \overline{z}]$ is spawn and a protocol is started to initialize correctly this new register. The protocol is between the process R_j and the compensation $\overline{u}.\cdots.\overline{z}$ left by the previous instance of the register. The process R_j consumes the remaining prefixes $\overline{u}$, and for each of them performs an $\mathsf{inst}\lfloor\lambda X.\overline{u}.X\rfloor$ action in order to update its compensation accordingly. In this way, at the end of the protocol, the new register instance will have a compensation $\overline{u}.\cdots.\overline{z}$ with one prefix $\overline{u}$ less w.r.t. the previous register instance.

Formally, the translation of register j storing value n is as follows:

$$[\![r_j = n]\!] \triangleq r_j[!inc_j.\,\mathsf{inst}\lfloor\lambda X.\overline{u}.X\rfloor.\overline{ack} \mid !rec_j.\big(u.\,\mathsf{inst}\lfloor\lambda X.\overline{u}.X\rfloor.\overline{rec_j} + z.\overline{ack}\big),\ \overline{u}^n.\overline{z}]$$

where $\overline{u}^n$ is a sequence of n prefixes $\overline{u}$. The encoding of instructions is as follows:

$$\begin{aligned}
&[\![i : Inc(r_j)]\!] &&\triangleq !p_i.\overline{inc_j}.ack.\overline{p_{i+1}}\\
&[\![i : DecJump(r_j, s)]\!] &&\triangleq !p_i.\overline{r_j}.\big(z.([\![r_j = 0]\!]|\overline{p_s}) + u.(\overline{rec_j}|[\![r_j = 0]\!]|ack.\overline{p_{i+1}})\big)
\end{aligned}$$

Hence, given a RAM program $I_1, \cdots, I_n$ with registers $r_1, \ldots, r_m$ with initial values $n_1, \ldots, n_m$ the corresponding encoding is:

$$\overline{p_1}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_m = n_m]\!]$$

In the proof of correctness of the encoding we use $P \rightarrow^k_{\equiv} Q$ to denote the existence of $Q_1, \cdots, Q_k$ such that $P \xrightarrow{\tau} Q_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} Q_k$ and $Q_k \equiv Q$.

Theorem 1.
Given $P \equiv \overline{p_l}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_j = n_j]\!]|\ldots|[\![r_m = n_m]\!]$ *we have:*

1. $I_l : Inc(r_j)$ *iff*
   $P \rightarrow^4_{\equiv} \overline{p_{l+1}}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_j = n_j + 1]\!]|\ldots|[\![r_m = n_m]\!]$;
2. $I_l : DecJump(r_j, s)$ *and* $n_j = 0$ *iff*
   $P \rightarrow^3_{\equiv} \overline{p_s}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_j = 0]\!]|\ldots|[\![r_m = n_m]\!]$;
3. $I_l : DecJump(r_j, s)$ *and* $n_j \neq 0$ *iff*
   $P \rightarrow^k_{\equiv} \overline{p_{l+1}}|[\![I_1]\!]|\ldots|[\![I_n]\!]|[\![r_1 = n_1]\!]|\ldots|[\![r_j = n_j - 1]\!]|\ldots|[\![r_m = n_m]\!]$ *with* $k = 3(n_j - 1) + 5$;
4. I_l *is undefined iff there exists no* P' *s.t.* $P \xrightarrow{\tau} P'$.

Proof. In each case there is just one possible computation, that we describe by listing the channels on which synchronizations happen or the installation of compensation performed:

1. p_l, inc_j, $\mathsf{inst}\lfloor \lambda X.\overline{u}.X \rfloor$, ack: 4 transitions;
2. p_l, r_j, z: 3 transitions;
3. p_l, r_j, u, rec_j, then the sequence u, $\mathsf{inst}\lfloor \lambda X.\overline{u}.X \rfloor$, rec_j repeated $n_j - 1$ times, and finally z, ack: $3(n_j - 1) + 5$ transitions;
4. no synchronization is possible. □

We finally conclude with the proof of the undecidability result.

Corollary 1. *Termination is undecidable in* π*-calculus with nested compensations.*

Proof. By Theorem 1 each step of a RAM precisely corresponds to a finite number of steps of its encoding, thus a RAM terminates iff its encoding terminates. Thus, termination of RAMs reduces to termination in π-calculus with nested compensations. Since termination in RAMs is undecidable then also termination in π-calculus with nested compensations is undecidable. □

4 Decidability for Parallel and Replacing Compensations

We now consider the cases in which all dynamic compensation installations follow the replace or the parallel patterns. In the first case, only finitely many different compensation processes can be considered. In the second case, infinitely many compensations can be reached, but all of them are parallel compositions of finitely many distinct processes (the processes Q occurring in the updates $\lambda X.Q \mid X$, and static compensations R in $t[P, R]$). This property of the calculus allows us to apply the theory of Well-Structured Transition Systems (WSTSs) to prove that termination is decidable.

We start by recalling some basic notions about WSTSs [1, 15]. A reflexive and transitive relation is called *quasi-ordering*. A *well-quasi-ordering* (wqo) is a quasi-ordering $(X, \leq)$ such that, for every infinite sequence $x_1, x_2, x_3, \cdots$, there exist $i < j$ with $x_i \leq x_j$. From this, it follows that there exists also an infinite increasing subsequence $x_{k_1}, x_{k_2}, x_{k_3}, \cdots$ such that $x_{k_l} \leq x_{k_m}$ for every

$l < m$. Given a wqo $(X, \leq)$, we denote its extension to k-tuples as $(X^k, \leq^k)$: $\langle x_1, \cdots, x_k \rangle \leq^k \langle y_1, \cdots, y_k \rangle$ iff $x_i \leq y_i$ for $1 \leq i \leq k$. Dickson's lemma [14] states that if $(X, \leq)$ is a wqo, then also $(X^k, \leq^k)$ is a wqo. Given a wqo $(X, \leq)$, we denote its extension to finite sequences as $(X^*, \leq^*)$: $\langle x_1, \cdots, x_n \rangle \leq^* \langle y_1, \cdots, y_m \rangle$ iff there exists a subsequence $\langle y_{l_1}, \cdots, y_{l_n} \rangle$ of the latter s.t. $x_i \leq y_{l_i}$ for $1 \leq i \leq n$. Higman's lemma [17] states that if $(X, \leq)$ is a wqo, then also $(X^*, \leq^*)$ is a wqo. We now report a definition of WSTS appropriate for our purposes.

Definition 3. *A WSTS is a transition system $(\mathcal{S}, \rightarrow, \preceq)$ where $\preceq$ is a wqo on $\mathcal{S}$ which is* compatible *with $\rightarrow$, i.e., for every $s_1 \preceq s'_1$ such that $s_1 \rightarrow s_2$, there exists $s'_1 \rightarrow s'_2$ such that $s_2 \preceq s'_2$. Moreover, the function Succ(s), returning the set $\{s' \in \mathcal{S} \mid s \rightarrow s'\}$ of immediate successors of s, is computable.*

A state s in a WSTS *terminates* if there exists no infinite computation $s \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$. The proposition below is a special case of Theorem 4.6 in [15].

Proposition 1. *Termination is decidable for WSTSs.*

Given a process P with replacing or parallel compensations, we prove that a transition system that includes all the derivatives of P is a WSTS. By derivatives, denoted with $\mathsf{der}(P)$, we mean the processes that can be reached from P via transitions labeled with τ, denoted simply with $\rightarrow$ in the following. We first observe that given a process Q, the set of its immediate successors according to $\rightarrow$ is finite (and computable). This follows from the limitation to τ-labeled transitions: the labeled transition system in Fig. 5 is not finitely branching because the rule (P-In) has an instantiation for each of the infinitely many possible vectors of values $\boldsymbol{v}$, but if we restrict to τ transitions, only finitely many names can be actually received because in our calculus no new names can be dynamically generated. Concerning names, we also make the nonrestrictive assumption that in process P the free names used in output actions are all distinct from the bound names used in input actions. In this way, it is not necessary to apply α-conversions to avoid name captures during substitutions. This guarantees that only the names initially present in P will occur in its derivatives.

We now move to the definition of our wqo. Intuitively, a process P is smaller than a process Q if Q can be obtained from P by adding some processes in parallel while preserving the same structure of transaction scopes and protected blocks.

Definition 4. *Let P, Q be two processes. We write $P \preceq Q$ iff there exist P', S, n, m, $t_1, \ldots, t_n$, $P_1, \ldots, P_n, P'_1, \ldots, P'_n$, $Q_1, \ldots, Q_n, Q'_1, \ldots, Q'_n$, $R_1, \ldots, R_m$ and $R'_1, \ldots, R'_m$ such that*

$$P \equiv P' \mid \prod_{i=1}^{n} t_i[P_i, Q_i] \mid \prod_{j=1}^{m} \langle R_j \rangle$$
$$Q \equiv P' \mid S \mid \prod_{i=1}^{n} t_i[P'_i, Q'_i] \mid \prod_{j=1}^{m} \langle R'_j \rangle$$

with $P_i \preceq P'_i$ and $Q_i \preceq Q'_i$, for $1 \leq i \leq n$, and $R_j \preceq R'_j$, for $1 \leq j \leq m$.

In order to prove that $\preceq$ is indeed a wqo over the derivatives of P we need some more notation and preliminary results. First we define the maximum nesting level $\mathsf{depth}(P)$ of nested transactions and protected blocks in a process P.

Definition 5. *Let P be a process. We define* $\mathsf{depth}(P)$ *inductively as follows:*

$$\begin{array}{l}\mathsf{depth}(\mathbf{0}) = \mathsf{depth}(X) = 0\\ \mathsf{depth}(\sum_{i\in I}\pi_i.P_i) = max_{i\in I}\,\mathsf{depth}(P_i)\\ \mathsf{depth}(!\pi.P) = \mathsf{depth}(P)\\ \mathsf{depth}(\mathsf{inst}\lfloor\lambda X.Q\rfloor.P) = max(\mathsf{depth}(P),\mathsf{depth}(Q))\\ \mathsf{depth}(P \mid Q) = max(\mathsf{depth}(P),\mathsf{depth}(Q))\\ \mathsf{depth}(t[P,Q]) = 1 + max(\mathsf{depth}(P),\mathsf{depth}(Q))\\ \mathsf{depth}(\langle P\rangle) = 1 + \mathsf{depth}(P)\end{array}$$

It is trivial to see that the extraction functions do not increase the maximum nesting levels in all the three considered cases. Formally, $\mathsf{depth}(\mathsf{extr}_{\mathsf{a}}(P)) \leq \mathsf{depth}(P)$, $\mathsf{depth}(\mathsf{extr}_{\mathsf{p}}(P)) \leq \mathsf{depth}(P)$ and $\mathsf{depth}(\mathsf{extr}_{\mathsf{d}}(P)) \leq \mathsf{depth}(P)$. We now prove that also the labeled transitions do not increase the nesting levels.

Proposition 2. *Let P be a process with replacing or parallel compensations. If $P \xrightarrow{\alpha} Q$ then* $\mathsf{depth}(Q) \leq \mathsf{depth}(P)$.

Proof. We first observe that for every transition $T \xrightarrow{\lambda X.S} T'$ we have that $\mathsf{depth}(S) \leq \mathsf{depth}(T)$. In the light of this preliminary result the thesis can be easily proved by induction on the depth of the proof of $P \xrightarrow{\alpha} Q$. The unique interesting case is when the rule (L-Scope-inst) is used. Consider the transition $t[P,Q] \xrightarrow{\lambda X.R} t[P', R\{^{Q}/_{X}\}]$ inferred by $P \xrightarrow{\lambda X.R} P'$. We have that $t[P', R\{^{Q}/_{X}\}]$ does not have a greater maximum nesting level because $\mathsf{depth}(R) \leq \mathsf{depth}(P)$, for the above observation, and $\mathsf{depth}(R\{^{Q}/_{X}\}) \leq max(\mathsf{depth}(Q),\mathsf{depth}(R))$ due to the specificity of the replace and parallel update patterns. □

As a trivial corollary we have that the maximum nesting level of the derivatives of P (i.e. processes in $\mathsf{der}(P)$) is smaller or equal to $\mathsf{depth}(P)$. This result will be used to define a superset of $\mathsf{der}(P)$ for which we will prove that $\preceq$ is indeed a wqo. In the definition of this superset we also need the notion of a sequential subprocess of P, that is a subterm of P in which the top operator is not a parallel composition, a transaction or a protection block.

Definition 6. *Let P be a process. The set* $\mathsf{seq}(P)$ *containing all the sequential subprocesses of P is defined inductively as follows:*

$$\begin{array}{l}\mathsf{seq}(\mathbf{0}) = \{\mathbf{0}\}\\ \mathsf{seq}(\sum_{i\in I}\pi_i.P_i) = \{\sum_{i\in I}\pi_i.P_i\} \cup \bigcup_{i\in I}\mathsf{seq}(P_i)\\ \mathsf{seq}(!\pi.P) = \{!\pi.P\} \cup \mathsf{seq}(P)\\ \mathsf{seq}(\mathsf{inst}\lfloor\lambda X.Q\rfloor.P) = \mathsf{inst}\lfloor\lambda X.Q\rfloor.P \cup \mathsf{seq}(P) \cup \mathsf{seq}(Q)\\ \mathsf{seq}(X) = \emptyset\\ \mathsf{seq}(P \mid Q) = \mathsf{seq}(t[P,Q]) = \mathsf{seq}(P) \cup \mathsf{seq}(Q)\\ \mathsf{seq}(\langle P\rangle) = \mathsf{seq}(P)\end{array}$$

The intuition is that no new sequential subprocesses can be generated by derivatives. To be more precise, after the execution of an input action, new subprocesses

can be reached due to name substitution. But, as observed above, the names in a derivative in $\mathsf{der}(P)$ already occur in P, thus they are finite. This allows us to characterize a superset of $\mathsf{der}(P)$ as follows.

Definition 7. *Given a process Q, we use $\mathsf{names}(Q)$ to denote the set of names occurring in Q. Let P be a process and n be a natural number; we denote with*

$$\mathsf{comb}_P(n) = \{Q \mid \mathsf{names}(Q) \subseteq \mathsf{names}(P), \mathsf{depth}(Q) \leq n, \\ \forall Q' \in \mathsf{seq}(Q).\exists P' \in \mathsf{seq}(P).Q' = P\{\boldsymbol{v}/\boldsymbol{x}\} \text{ for some } \boldsymbol{v} \text{ and } \boldsymbol{x}\}$$

the set of processes with names that already occur in P, with maximum nesting level smaller than n, and containing sequential subprocesses that already occur in P (up-to renaming).

We now prove that $\mathsf{comb}_P(\mathsf{depth}(P))$ is actually a superset of $\mathsf{der}(P)$.

Proposition 3. *Let P be a process with replacing or parallel compensations. Then $\mathsf{der}(P) \subseteq \mathsf{comb}_P(\mathsf{depth}(P))$.*

Proof. We first observe that $P \in \mathsf{comb}_P(\mathsf{depth}(P))$. Then we consider a process $Q \in \mathsf{comb}_P(\mathsf{depth}(P))$ such that $Q \to Q'$, and we show that also $Q' \in \mathsf{comb}_P(\mathsf{depth}(P))$. By Proposition 2 we have that $\mathsf{depth}(Q') \leq \mathsf{depth}(Q)$ hence also $\mathsf{depth}(Q') \leq \mathsf{depth}(P)$. Moreover, it is easy to see that Q' does not introduce new sequential subprocesses (it can at most apply a name substitution to sequential subprocesses of Q). Notice that in case the transition is a compensation update, no new sequential subprocesses can be obtained because either the replace or the parallel pattern is used. □

We are finally ready to prove that $(\mathsf{comb}_P(\mathsf{depth}(P)), \preceq)$ is indeed a wqo, by proving a slightly more general result.

Theorem 2. *Let P be a process and let n be a natural number. The relation $\preceq$ is a wqo over $\mathsf{comb}_P(n)$.*

Proof. Take an infinite sequence $P_1, P_2, \ldots, P_i, \ldots$, with $P_i \in \mathsf{comb}_P(n)$ for every $i > 0$. We prove, by induction on n, that there exist k and l such that $P_k \preceq P_l$.

Let $n = 0$. All the processes P_i do not contain neither transactions nor protected blocks because $\mathsf{depth}(P_i) \leq 0$. For this reason, we have that $P_i = \prod_{j=1}^{n_i} P_{i,j}$ with $P_{i,j}$ equal to some sequential subprocess of P (up-to renaming by using names already in P). This set is finite, then process equality $=$ is a wqo over this set. By Higman's lemma we have that also $=^*$ is a wqo over finite sequences of such processes. Hence there exists k and l such that $P_{k,1} \ldots P_{k,n_k}$ is a subsequence of $P_{l,1} \ldots P_{l,n_l}$, hence we have $P_k \preceq P_l$.

For the inductive step, let $n > 0$ and assume that the thesis holds for $\mathsf{comb}_P(n-1)$. We have that the following holds for every P_i:

$$P_i \equiv \prod_{j=1}^{n_i} P_{i,j} \mid \prod_{j=1}^{m_i} t_{i,j}[Q_{i,j}, R_{i,j}] \mid \prod_{j=1}^{o_i} \langle S_{i,j} \rangle$$

with $P_{i,j}$ equal to some sequential subprocess of P (up-to renaming by using names already in P), $t_{i,j} \in \mathsf{names}(P)$ and $Q_{i,j}$, $R_{i,j}$, $S_{i,j}$ have a maximum nesting level strictly smaller than n, hence $Q_{i,j}, R_{i,j}, S_{i,j} \in \mathsf{comb}_P(n-1)$. We now consider every process P_i as composed of 3 finite sequences: $P_{i,1} \cdots P_{i,n_i}$, $\langle t_{i,1}, Q_{i,1}, R_{i,1} \rangle \cdots \langle t_{i,m_i}, Q_{i,m_i}, R_{i,m_i} \rangle$, and $S_{i,1} \cdots S_{i,o_i}$. As observed above, $=^*$ is a wqo over the sequences $P_{i,1} \cdots P_{i,n_i}$. For this reason we can extract an infinite subsequence of $P_1, P_2, \ldots$ making the finite sequences $P_{i,1} \cdots P_{i,n_i}$ increasing w.r.t. $=^*$. We now consider the triples $\langle t_{i,j}, Q_{i,j}, R_{i,j} \rangle$. Consider the ordering $(\mathsf{comb}_P(n-1) \cup \mathsf{names}(P), \sqsubseteq)$ such that $x \sqsubseteq y$ iff $x = y$, if $x, y \in \mathsf{names}(P)$, or $x \preceq y$, if $x, y \in \mathsf{comb}_P(n-1)$. As $\mathsf{names}(P)$ is finite and due to the inductive hypothesis according to which $(\mathsf{comb}_P(n-1), \preceq)$ is a wqo, we have that also $(\mathsf{comb}_P(n-1) \cup \mathsf{names}(P), \sqsubseteq)$ is a wqo. By Dickson's lemma we have that $\sqsubseteq^3$ is a wqo over the considered triples $\langle t_{i,j}, Q_{i,j}, R_{i,j} \rangle$. We can apply the Higman's lemma as above to prove that it is possible to extract, from the above infinite subsequence, an infinite subsequence making the finite sequences $\langle t_{i,1}, Q_{i,1}, R_{i,1} \rangle \cdots \langle t_{i,m_i}, Q_{i,m_i}, R_{i,m_i} \rangle$ increasing w.r.t. $(\sqsubseteq^k)^*$. Finally, as $S_{i,j} \in \mathsf{comb}_P(n-1)$ and by inductive hypothesis, we can finally apply again Higman's lemma to extract, from the last infinite sequence, an infinite subsequence making the finite sequences $S_{i,1} \cdots S_{i,o_i}$ increasing w.r.t. $\preceq^*$. It is now sufficient to take from this last subsequence two processes P_k and P_l, with $k < l$, and observe that $P_k \preceq P_l$. □

We now move to the proof of compatibility between the ordering $\preceq$ and the transition system $\rightarrow$.

Lemma 1. *If $P \preceq P'$ and $P \xrightarrow{\alpha} Q$ then there exists Q' such that $Q \preceq Q'$ and $P' \xrightarrow{\alpha} Q'$.*

Proof. By induction on the depth of the proof of $P \xrightarrow{\alpha} Q$. □

As the transitions $\rightarrow$ correspond to transitions labeled with τ, as a trivial corollary we have the compatibility of $\preceq$ with $\rightarrow$. Hence, we can conclude that given a process P with replacing or parallel (as well as static) compensations, $(\mathsf{comb}_P(\mathsf{depth}(P)), \rightarrow, \preceq)$ is a WSTS. As a consequence, we obtain our decidability result.

Corollary 2. *Let P be a process with replacing, parallel or static compensations. The termination of P is decidable.*

Proof. By definition, P terminates iff there exists no infinite computation $P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \ldots$. For replacing and parallel compensations, by Proposition 3, this holds iff P terminates in the transition system $(\mathsf{comb}_P(\mathsf{depth}(P)), \rightarrow)$. But this last problem is decidable, by Proposition 1, because $(\mathsf{comb}_P(\mathsf{depth}(P)), \rightarrow, \preceq)$ is a WSTS. The result for static compensations follows since they form a subcalculus of replacing/parallel compensations. □

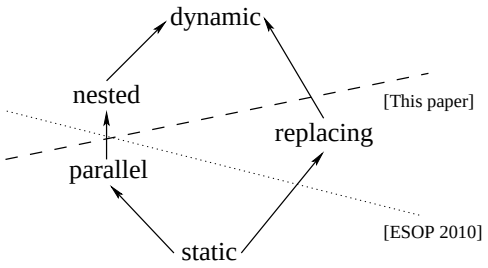

Fig. 6. Separation results for compensation mechanisms.

5 Related Work and Conclusion

In this paper we studied decidability properties of π-calculus extended with primitives for specifying transactions and compensations. Fig. 6 shows all the considered calculi. Arrows denote the subcalculus relation. As already said, [20] is the closest paper to ours. There, relying on syntactic conditions restricting the allowed class of encodings and requiring some strong semantic properties to be preserved, the authors proved the separation result represented by the dotted line. The results in this paper instead, requiring only termination preservation, prove the separation result represented by the dashed line. Besides separation, [20] also showed an encoding proving the equivalence of static and parallel compensations. This result, compatible with our separation result, cannot be straightforwardly applied in our setting since it relies on the restriction operator. However, if one disallows transactions under a replication prefix, our decidability results still hold and the encoding in [20] can be applied. It would be interesting to look for termination-preserving encodings of dynamic into nested compensations and replacing into static compensations (such an encoding should violate some of the conditions in [20]).

The only other results comparing the expressive power of primitives for transactions and compensations are in the field of SAGAs [9]/cCSP [10], but their setting allows only for isolated activities, since it does not consider communication. There are two kinds of results: a few papers compare different variants of SAGAs [6, 18, 7], while others use SAGAs-like calculi as specifications for π-style processes [21, 11]. Both kinds of results cannot be easily compared with ours.

Interestingly, our results have been studied in the framework of π-calculus since it is the base of most proposals in the literature, but can similarly be stated in CCS. Sticking to π-calculus, adding priority of compensation installation to the calculus, as done by [20, 16, 27], does not alter the undecidability of termination for nested and dynamic compensations. For the decidability in parallel and replacing compensations instead, the proof cannot be applied. Note however that priority of compensation installation reduces the set of allowed traces, thus termination without priority ensures termination with priority (but the opposite is not true).

Decidability and undecidability results are a well-established tool to separate the expressive power of process calculi. We restrict our discussion to few recent papers. In [5] two operators for modeling the interruption of processes are considered: $P \lhd Q$ that behaves like P until Q starts and $\texttt{try}\, P\, \texttt{catch}\, Q$ that behaves like P until a `throw` action is executed by P to activate Q. Termination is proved to be undecidable for $\texttt{try}\, P\, \texttt{catch}\, Q$ while it is decidable for $P \lhd Q$. The undecidability proof is different from the one in this paper since it exploits unbounded nesting of try-catch constructs. The decidability proof requires to use a weaker ordering (tree embedding) in order to deal with unbounded nesting of interrupt operators. Such ordering is not appropriate for the calculus in the present paper because compatibility is broken by the prefix $\mathsf{inst}\lfloor \lambda X.Q \rfloor$ that synchronizes with the nearest enclosing transaction and not with any of the outer transactions. In [13] higher-order π-calculus without restriction is considered. Despite higher-order communication is rather different w.r.t. dynamic compensations, a similar decidability result is proved: if the received processes cannot be modified when they are forwarded, termination becomes decidable, while this is not the case if they can [19]. The decidability proof is simpler w.r.t. the one in this paper because there is no operator, like $t[P, Q]$, that requires the exploitation of Dickson's lemma. Finally, we mention [4] where a calculus for adaptable processes is presented: running processes can be dynamically modified by executing update patterns similar to those used in this paper. A safety property is proved to be decidable if the update pattern does not add prefixes in front of the adapted process, while it becomes undecidable if a more permissive pattern is admitted. The undecidability proof in the present paper is more complex because update patterns can be executed only on inactive processes (the compensations). The decidability proof in [4] is similar to the one in [5]: the same comments above holds also in this case.

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. of LICS'96*, pages 313–321. IEEE, 1996.
2. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *Proc. of FMOODS'03*, volume 2884 of *LNCS*, pages 124–138. Springer, 2003.
3. L. Bocchi and E. Tuosto. A Java inspired semantics for transactions in SOC. In *Proc. of TGC 2010*, volume 6084 of *LNCS*, pages 120–134. Springer, 2010.
4. M. Bravetti, C. D. Giusto, J. A. Pérez, and G. Zavattaro. Adaptable processes. *Logical Methods in Computer Science*, 8(4), 2012.
5. M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Math. Struct. Comp. Sci.*, 19(3):565–599, 2009.
6. R. Bruni, M. J. Butler, C. Ferreira, C. A. R. Hoare, H. C. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. In *Proc . of CONCUR'05*, volume 3653 of *LNCS*, pages 383–397. Springer, 2005.
7. R. Bruni, A. Kersten, I. Lanese, and G. Spagnolo. A new strategy for distributed compensations with interruption in long-running transactions. In *Proc. of WADT 2010*, volume 7137 of *LNCS*, pages 42–60. Springer, 2010.

8. R. Bruni, H. C. Melgratti, and U. Montanari. Nested commits for mobile calculi: Extending join. In *Proc. of IFIP TCS'04*, pages 563–576. Kluwer, 2004.
9. R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. of POPL '05*, pages 209–220. ACM Press, 2005.
10. M. J. Butler, C. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years CSP*, volume 3525 of *LNCS*, pages 133–150. Springer, 2004.
11. L. Caires, C. Ferreira, and H. Vieira. A process calculus analysis of compensations. In *Proc. of TGC'08*, volume 5474 of *LNCS*, pages 87–103. Springer, 2008.
12. E. de Vries, V. Koutavas, and M. Hennessy. Communicating transactions. In *Proc. of CONCUR 2010*, volume 6269 of *LNCS*, pages 569–583. Springer, 2010.
13. C. Di Giusto, J. A. Pérez, and G. Zavattaro. On the expressiveness of forwarding in higher-order communication. In *Proc. of ICTAC'09*, volume 5684 of *LNCS*, pages 155–169. Springer, 2009.
14. L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amer. J. Math.*, 35(4):413–422, 1913.
15. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256:63–92, 2001.
16. C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. Dynamic error handling in service oriented applications. *Fundamenta Informaticae*, 95(1):73–102, 2009.
17. G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc., 3rd series*, 2:326–336, 1952.
18. I. Lanese. Static vs dynamic sagas. In *Proc. of ICE 2010*, volume 38 of *EPTCS*, pages 51–65, 2010.
19. I. Lanese, J. A. Pérez, D. Sangiorgi, and A. Schmitt. On the expressiveness and decidability of higher-order process calculi. In *Proc. of LICS'08*, pages 145–155. IEEE Computer Society, 2008.
20. I. Lanese, C. Vaz, and C. Ferreira. On the expressive power of primitives for compensation handling. In *Proc. of ESOP 2010*, volume 6012 of *LNCS*, pages 366–386. Springer, 2010.
21. I. Lanese and G. Zavattaro. Programming Sagas in SOCK. In *Proc. of SEFM'09*, pages 189–198. IEEE Computer Society Press, 2009.
22. C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proc. of FoSSaCS'05*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.
23. R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inf. Comput.*, 100(1):1–40,41–77, 1992.
24. M. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Englewood Cliffs, 1967.
25. F. Montesi, C. Guidi, and G. Zavattaro. Composing services with JOLIE. In *Proc. of ECOWS'07*, pages 13–22. IEEE Computer Society, 2007.
26. Oasis. *Web Services Business Process Execution Language Version 2.0*, 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.
27. C. Vaz, C. Ferreira, and A. Ravara. Dynamic recovering of long running transactions. In *Proc. of TGC'08*, volume 5474 of *LNCS*, pages 201–215. Springer, 2008.