On the Expressive Power of Primitives for Compensation Handling*

Ivan Lanese[1], Cátia Vaz[2], and Carla Ferreira[3]

[1] Lab. Focus, Università di Bologna/INRIA, Italy
lanese@cs.unibo.it
[2] INESC-ID / DEETC, ISEL, Instituto Politécnico de Lisboa, Portugal
cvaz@cc.isel.ipl.pt
[3] CITI / Departamento de Informática, FCT, Universidade Nova de Lisboa, Portugal
carla.ferreira@di.fct.unl.pt

Abstract. Modern software systems have frequently to face unexpected events, reacting so to reach a consistent state. In the field of concurrent and mobile systems (e.g., for web services) the problem is usually tackled using long running transactions and compensations: activities programmed to recover partial executions of long running transactions.

We compare the expressive power of different approaches to the specification of those compensations. We consider (i) *static recovery*, where the compensation is statically defined together with the transaction, (ii) *parallel recovery*, where the compensation is dynamically built as parallel composition of compensation elements and (iii) *general dynamic recovery*, where more refined ways of composing compensation elements are provided. We define an encoding of parallel recovery into static recovery enjoying nice compositionality properties, showing that the two approaches have the same expressive power. We also show that no such encoding of general dynamic recovery into static recovery is possible, i.e. general dynamic recovery is strictly more expressive.

1 Introduction

Modern software systems are complex and composed by different interacting components, commonly developed and managed separately. Also, they usually rely on communication infrastructures, such as the Internet or wireless networks, that are unreliable. Thus unexpected events can frequently arise during the execution of such applications: received data items may not have the desired structure, communication partners may disconnect, etc. In this context it is important to use suitable error handling techniques allowing the whole system to reach a correct state even if some of its components have failed.

In the field of concurrent and mobile systems (e.g., in the case of web services), this problem is usually tackled using the concept of long running transaction.

* Research supported by the Project FET-GC II IST-2005-16004 Sensoria, FP7-231620 HATS and by FCT grant SFRH/BD/45572/2008.

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 366–386, 2010.

A long running transaction either succeeds, or a compensation is executed taking the system to a consistent state, possibly different from the one in which the transaction started. This weakens the constraint of ACID transactions from database theory, since it is difficult to guarantee ACID properties when transactions can last for a long time, and when some actions cannot be undone.

In the literature there are different proposals of primitives for long running transactions, from the Java try P catch e Q[1], where Q is in charge of managing exception e raised inside P, to the complex mechanisms of WS-BPEL [1] (the de-facto standard for web services composition), exploiting fault, termination and compensation handlers to deal with different error handling issues.

However, the relationships between the different proposals are not clear, and there has been little work trying to formally compare the expressive power of the proposed mechanisms. This problem is made hard by the fact that different primitives for long running transactions are realized on top of different underlying languages. Thus the different expressive power of the error handling primitives is hidden because of other differences between the underlying languages. Understanding the expressive power of different primitives is important for language design: primitives that do not add expressive power can be left out from the core language and implemented as macros when needed, primitives that add expressive power should be implemented in the core language.

This paper tackles this problem, by presenting a formal comparison of different approaches to long running transactions in a concurrent and mobile setting. To this end we add primitives for error handling, distilled from approaches in the literature, to the same underlying language, so to have a more clear comparison. We have chosen the simplest possible underlying language able to model concurrent and mobile systems: the π-calculus [2]. Then further work is required to apply the results to more complex calculi and real languages (see Section 6).

The approaches to error handling are far too many to be compared here, thus we concentrate on a main feature: whether the compensation code for a transaction is statically defined, or it is dynamically generated. *Static recovery* is for instance the approach of Java try-catch, and is the classic approach of interaction-based models [3–6]. For dynamic recovery we consider two different possibilities: in *parallel recovery* the compensation is incrementally built as parallel composition of simpler compensations, while in *general dynamic recovery* compensations can be both updated and replaced. Parallel recovery is commonly used [1, 7, 8] to execute compensations of subtransactions when a transaction fails, and it is the mechanism exploited by dcπ [9]. Most of the compensable flow approaches [7, 8, 10], where compensations of complex activities are built as compositions of compensations of their constituting activities, execute compensations of sequential activities in backward order. Compensations are always executed in backward order in *backward recovery* [11]. Backward recovery is the main instance of general dynamic recovery, which has been proposed in [12]. Backward recovery has also been applied to Java in [13].

[1] Actually, Java try-catch is designed for exception handling, but can be used also for programming long running transactions.

π	$::=$	π-*calculus prefixes*	P, Q $::=$	π-*calculus processes*
		$\overline{a}\langle \boldsymbol{v} \rangle$ (Output prefix)	$\mathbf{0}$	(Inaction)
	$\mid$	$a(\boldsymbol{x})$ (Input prefix)	$\mid \sum_{i \in I} \pi_i.P_i$	(Guarded choice)
			$\mid\ !\pi.P$	(Guarded replication)
			$\mid\ P \mid Q$	(Parallel composition)
			$\mid\ (\nu\, x)\ P$	(Restriction)

Fig. 1. π-calculus processes

This paper compares the expressive power of static recovery, parallel recovery and general dynamic recovery in the context of π-calculus. Our main results are:

- a compositional encoding of parallel recovery into static recovery;
- a separation result showing that no similar encoding exists from general dynamic recovery (neither from backward recovery) to static recovery.

We also discuss how these results can be applied to other calculi in the literature.

Structure of the work: Section 2 introduces the primitives for long running transactions. Section 3 discusses the conditions that a good encoding must satisfy. Sections 4 and 5 present the main technical results: the encoding of parallel recovery into static recovery, and the impossibility of encoding general dynamic recovery into static recovery. Finally, Section 6 discusses how to apply the results to calculi in the literature. Proofs can be found in [14].

2 Primitives for Compensations

2.1 Syntax

In this section we formalize in the framework of π-calculus [2] some primitives for static, parallel and general dynamic recovery. The relationships between these primitives and other primitives in the literature are discussed in Section 6.

To simplify the understanding and the comparisons, we define the three calculi corresponding to static, parallel and general dynamic recovery in an incremental way. The syntax of all our calculi 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 such names, for some $n \geq 0$, and $\{\boldsymbol{x}\}$ denotes the set of elements in the tuple. As already said, our calculi are built on top of π-calculus, whose syntax is in Fig. 1.

Prefixes in π-calculus can be either outputs $\overline{a}\langle \boldsymbol{v} \rangle$ of a tuple of values $\boldsymbol{v}$ on channel a, or corresponding inputs $a(\boldsymbol{x})$. The π-calculus syntax includes the inactive process $\mathbf{0}$, guarded choice $\sum_{i \in I} \pi_i.P_i$, guarded replication $!\pi.P$, parallel composition $P \mid Q$ of processes P and Q, and restriction $(\nu\, x)\, P$ of name x inside P. 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. We also write $(\nu\, \boldsymbol{x})$ for $(\nu\, x_1) \cdots (\nu\, x_n)$ when $\boldsymbol{x} = x_1, \ldots, x_n$. The formal description of the semantics will be given in Section 2.2 (see also [2]).

The first, and simpler, extension that we present corresponds to static recovery. The syntax is presented in Fig. 2 (left). Static recovery can be realized by adding just two constructs: *transaction scope* and *protected block*. A transaction

$P, Q ::=$	*Static rec. processes*	$P, Q ::=$	*Compensable processes*
$\ldots$	(π-calculus processes)	$\ldots$	(Static rec. processes)
$\mid\ t[P, Q]$	(Transaction scope)	$\mid\ X$	(Process variable)
$\mid\ \langle P \rangle$	(Protected block)	$\mid\ \mathsf{inst}\lfloor \lambda X.Q \rfloor.P$	(Compensation update)

Fig. 2. Static recovery and compensable processes

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. When such a notification is received, the body P of the transaction is killed and compensation Q is executed. Q is executed in a protected block, i.e. not influenced by successive external errors. Error notifications may be generated both from the body P and from external processes. Error notifications are simple output messages (without parameters). Thus one may have nondeterminism, since the same output may be caught either by an input or by a transaction scope. If such a behavior is not desired, it can be avoided by using a simple sorting system. We will not consider this issue. Protected block $\langle P \rangle$ behaves as P, but it is not killed in case of failure of an external transaction.

Compensable processes, which realize general dynamic recovery, extend static recovery processes. The main difference is that in compensable processes the body P of transaction $t[P, Q]$ can update the compensation Q. *Compensation update* is performed by a new 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 compensable processes extends the one of static recovery processes with the compensation update operator and process variables (see Fig. 2 (right)).

We define for compensable processes the usual notions of free and bound names. Names in $\boldsymbol{x}$ are bound in $a(\boldsymbol{x}).P$, while x is bound in $(\nu\, x)\, P$. Other names are free. We denote with $\mathsf{fn}(\bullet)$, $\mathsf{bn}(\bullet)$ and $\mathsf{n}(\bullet)$ the functions computing the sets of free, bound and all the names respectively. Also, variable X is bound in $\lambda X.Q$. Bound names and variables can be α-converted as usual. We consider only processes with no free variables. For simplicity we may drop trailing $\mathbf{0}$s.

Static recovery processes are a subcalculus of compensable processes where compensation update is never used. Also, if a compensation update has the form $\lambda X.Q \mid X$ where X does not occur in Q, then Q is added in parallel to the existing compensation. Thus parallel recovery can be seen as a particular case of compensable processes too. When speaking about parallel recovery we will write a compensation update $\lambda X.Q \mid X$ simply as Q.

Definition 1 (Classes of processes). Compensable processes $\mathcal{CP}$ *are defined by the syntax in Fig. 2 (right).* Parallel recovery processes $\mathcal{PP}$ *are compensable processes where all the compensation updates have the form* $\lambda X.Q \mid X$ *where* Q *is a process without free variables.* Static recovery processes $\mathcal{SP}$ *are compensable processes where the compensation update operator is never used.*

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

Fig. 3. Extraction function with nested failure

The main question that this paper wants to answer is whether the three classes of processes $\mathcal{CP}$, $\mathcal{PP}$ and $\mathcal{SP}$ have the same expressive power or not.

2.2 Operational Semantics

To define the operational semantics of compensable processes we need an auxiliary definition: when a transaction scope $t[P,Q]$ is killed, part of its body P has to be preserved, in particular the part composed of protected blocks.

The definition of function $\mathsf{extr}(P)$ computing the part to be preserved depends on the meaning of transaction nesting. In the literature, two approaches are considered: according to the *nested failure* approach a subtransaction has to be killed when the transaction containing it is killed. This is for instance the approach of SAGAs calculi [8], WS-BPEL [1], and others. In the *non-nested failure* approach instead, subtransactions are unaffected by external failures (however the recovery of a transaction may include the explicit killing of its subtransactions). This is for instance the approach of Webπ [5]. We consider both the possibilities, since they just differ in the definition of function $\mathsf{extr}(\bullet)$. Our results hold in both the cases. One can simulate the non-nested approach using the nested one by protecting each transaction using a protected block, while it is not clear whether the opposite simulation is possible. Clarifying this point is left for future work.

Definition 2 (Extraction function). *We denote the functions corresponding to nested and non-nested failure respectively as* $\mathsf{extr_n}(\bullet)$ *and* $\mathsf{extr_{nn}}(\bullet)$. *The function* $\mathsf{extr_n}(\bullet)$ *is defined in Fig. 3. The definition of function* $\mathsf{extr_{nn}}(\bullet)$ *is the same but for the clause for transaction scope, which becomes* $\mathsf{extr_{nn}}(t\,[P,Q]) = t\,[P,Q]$.

There is no need to define $\mathsf{extr_n}(X)$ or $\mathsf{extr_{nn}}(X)$ since X can occur only inside the compensation update primitive.

We also need an auxiliary predicate $\mathsf{noComp}(P)$ which is true iff P has no pending compensation update. This is needed since a compensation update is performed to reflect in the compensation some change in the state of the executing process, and it should never happen that the state has changed and the compensation update has not been performed. In other words, compensation update should have priority w.r.t. other transitions (see [15] for a discussion on this topic). Priority of compensation update is obtained by ensuring in the semantics that when an action (different from a compensation update) is performed, no compensation update is pending.

Definition 3 ($\mathsf{noComp}(\bullet)$ predicate). *The predicate* $\mathsf{noComp}(P)$ *that verifies the non-existence of pending compensation updates in* P *is defined in Fig. 4.*

$$\begin{array}{rl} \mathsf{noComp}(\mathbf{0}) & \mathsf{noComp}(\langle P \rangle) \text{ if } \mathsf{noComp}(P) \\ \mathsf{noComp}(\sum_{i \in I} \pi_i.P_i) & \mathsf{noComp}(t\,[P, Q]) \text{ if } \mathsf{noComp}(P) \\ \mathsf{noComp}(!\pi.P) & \mathsf{noComp}(P \mid Q) \text{ if } \mathsf{noComp}(P) \text{ and } \mathsf{noComp}(Q) \\ & \mathsf{noComp}((\nu\, x)\, P) \text{ if } \mathsf{noComp}(P) \end{array}$$

Fig. 4. No pending compensation update predicate

In particular, $\mathsf{noComp}(P)$ is false if P is a compensation update primitive.

The operational semantics of compensable processes (and, implicitly, of static recovery and parallel recovery processes) is defined below. We use $a(\boldsymbol{v})$, $(\boldsymbol{w})\overline{a}\langle\boldsymbol{v}\rangle$, τ, $(\boldsymbol{w})\lambda X.Q$ and τ_c as labels. The first three forms of labels are as in π-calculus (but outputs are also used for error notification, and inputs for receiving the notification), while the last two labels are for compensation update. In particular, $(\boldsymbol{w})\lambda X.Q$ requires a compensation update while τ_c is the corresponding internal action. This has to be distinguished from τ since it has priority. We write a for $a(\boldsymbol{v})$ and $\overline{a}$ for $\overline{a}\langle\boldsymbol{v}\rangle$ if $\boldsymbol{v}$ is empty. We use t instead of a to emphasize that the name is used for error notification. Names in $\boldsymbol{w}$ are bound in $(\boldsymbol{w})\overline{a}\langle\boldsymbol{v}\rangle$ and $(\boldsymbol{w})\lambda X.Q$. Other names are free. Functions $\mathsf{fn}(\bullet)$, $\mathsf{bn}(\bullet)$ and $\mathsf{n}(\bullet)$ are extended accordingly. We drop the set of bound names $(\boldsymbol{w})$ from labels if it is empty.

Definition 4 (Operational semantics). *The operational semantics with nested failure of compensable processes $\mathcal{CP}$ is the minimum labeled transition system (LTS) closed under the rules in Fig. 5 (symmetric rules are considered for (*L-Par*) and (*L-Close*)). The operational semantics with non-nested failure of compensable processes $\mathcal{CP}$ is the minimum LTS closed under the rules in Fig. 5 (symmetric rules are considered for (*L-Par*) and (*L-Close*)), but where function $\mathsf{extr}_{\mathsf{n}}(\bullet)$ is replaced by function $\mathsf{extr}_{\mathsf{nn}}(\bullet)$.*

The first seven rules and the ninth extend the corresponding π-calculus rules [2], 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. If the performed action is not a compensation update, then the rule verifies that no compensation update is pending in the other component (last condition). Rule (L-Res) is the classic rule for restriction. Rule (L-Open) allows to extrude bound names. Rule (L-Close) performs communication. If the output action contains some extruded names, restrictions for them are reintroduced.

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 kill a transaction scope via an output $\overline{t}$. The resulting 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. The condition ensures that there are no pending compensation updates. Rule (L-Recover-in) is similar to (L-Recover-out), but now the error notification comes from P. In this case condition

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

$$\text{(L-Rep)}\ \frac{\pi.P \xrightarrow{\alpha} P'}{!\pi.P \xrightarrow{\alpha} P' | !\pi.P} \qquad \text{(L-Par)}\ \frac{P \xrightarrow{\alpha} P' \quad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset \quad \alpha \notin \{(\boldsymbol{w})\lambda X.R, \tau_c\} \Rightarrow \mathsf{noComp}(Q)}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{(L-Res)}\ \frac{P \xrightarrow{\alpha} P' \quad x \notin \mathsf{n}(\alpha)}{(\nu\, x)\, P \xrightarrow{\alpha} (\nu\, x)\, P'}$$

$$\text{(L-Open)}\ \frac{P \xrightarrow{(\boldsymbol{w})\overline{x}\langle \boldsymbol{v}\rangle} P' \quad z \neq x \quad z \in \{\boldsymbol{v}\} \setminus \{\boldsymbol{w}\}}{(\nu\, z)\, P \xrightarrow{(z\boldsymbol{w})\overline{x}\langle \boldsymbol{v}\rangle} P'} \qquad \text{(L-Open2)}\ \frac{P \xrightarrow{(\boldsymbol{w})\lambda X.Q} P' \quad z \in \mathsf{fn}(Q) \setminus \{\boldsymbol{w}\}}{(\nu\, z)\, P \xrightarrow{(z\boldsymbol{w})\lambda X.Q} P'}$$

$$\text{(L-Close)}\ \frac{P \xrightarrow{x(\boldsymbol{v})} P' \quad Q \xrightarrow{(\boldsymbol{z})\overline{x}\langle \boldsymbol{v}\rangle} Q' \quad \{\boldsymbol{z}\} \cap \mathsf{fn}(P) = \emptyset}{P \mid Q \xrightarrow{\tau} (\nu\, \boldsymbol{z})\, (P' \mid Q')}$$

$$\text{(L-Scope-close)}\ \frac{P \xrightarrow{(\boldsymbol{z})\lambda X.R} P' \quad \{\boldsymbol{z}\} \cap (\mathsf{fn}(Q) \cup \{t\}) = \emptyset}{t[P, Q] \xrightarrow{\tau_c} (\nu\, \boldsymbol{z})\, t[P', R\{{}^{Q}/_{X}\}]} \qquad \text{(L-Recover-out)}\ \frac{\mathsf{noComp}(P)}{t[P, Q] \xrightarrow{t} \mathsf{extr_n}(P) \mid \langle Q\rangle}$$

$$\text{(L-Scope-out)}\ \frac{P \xrightarrow{\alpha} P' \quad \alpha \neq (\boldsymbol{z})\lambda X.Q \quad \mathsf{bn}(\alpha) \cap (\mathsf{fn}(Q) \cup \{t\}) = \emptyset}{t[P, Q] \xrightarrow{\alpha} t[P', Q]}$$

$$\text{(L-Recover-in)}\ \frac{P \xrightarrow{\overline{t}} P'}{t[P, Q] \xrightarrow{\tau} \mathsf{extr_n}(P') \mid \langle Q\rangle} \qquad \text{(L-Inst)}\ \frac{}{\mathsf{inst}\lfloor \lambda X.Q \rfloor.P \xrightarrow{\lambda X.Q} P} \qquad \text{(L-Block)}\ \frac{P \xrightarrow{\alpha} P'}{\langle P\rangle \xrightarrow{\alpha} \langle P'\rangle}$$

Fig. 5. LTS for compensable processes

$\mathsf{noComp}(P)$ is redundant since it can be deduced from the derivation. Rule (L-Inst) requires a compensation update (note that the resulting internal action is τ_c) while rule (L-Open2) allows to extrude bound names occurring in it. Rule (L-Scope-close) updates the compensation of a transaction scope (the substitution should not capture free names). If the compensation update includes extruded names, restrictions for these names are reintroduced (similarly to rule (L-Close)). Finally, rule (L-Block) defines the behavior of protected blocks.

Example 1. We give here a few examples of transitions[2].

- Transaction scopes can compute: $\overline{a}\langle b\rangle \mid t[a(x).\overline{x}.0, Q] \xrightarrow{\tau} 0 \mid t[\overline{b}.0, Q]$
- Transaction scopes can be killed: $\overline{t} \mid t[\overline{a}.0, Q] \xrightarrow{\tau} \langle Q\rangle$
- Transaction scopes can commit suicide: $t[\overline{t}.0 \mid \overline{a}.0, Q] \xrightarrow{\tau} \langle Q\rangle$
- New compensations can be added in parallel: $t[\mathsf{inst}\lfloor \lambda X.P|X \rfloor.\overline{a}.0, Q] \xrightarrow{\tau_c} t[\overline{a}.0, P|Q]$

[2] To simplify the presentation we discard some garbage. This can be done using the notion of structural congruence in Definition 14.

– New compensations can be added at the beginning:
$t[\mathsf{inst}\lfloor\lambda X.\overline{b}.X\rfloor.\overline{a}.0, Q] \xrightarrow{\tau_c} t[\overline{a}.0, \overline{b}.Q]$
– Compensations can be deleted: $t[\mathsf{inst}\lfloor\lambda X.0\rfloor.\overline{a}.0, Q] \xrightarrow{\tau_c} t[\overline{a}.0, 0]$

3 Conditions for Good Encodings

When discussing encodability/separation results, a main point is to decide which conditions an encoding has to satisfy in order to be considered a good means for language comparison. In the literature there are different proposals of such conditions [16–19]. The choice of the conditions determines the level of abstraction used when comparing the different languages. Since different expressiveness gaps are visible at different levels of abstraction, there are no universally good sets of conditions. Also, encodability results are stronger if stated at the low level of abstraction, i.e. with more strict conditions, while separation results are more general when proved at the high level of abstraction. However, it is important that related results are proved under the same conditions, thus defining a coherent picture of the expressiveness at the chosen level of abstraction. For these reasons we discuss below the conditions that we use throughout the paper, thus fixing our level of abstraction. We will consider stricter conditions too when proving encodability results, thus strengthening them.

There are two kinds of conditions: (i) syntactic conditions on the form of the translation, and (ii) conditions specifying the kind of behavior that the translation should preserve. We will base the latter on the concepts of divergence and should testing equivalence [20] (this choice will be discussed later).

Definition 5. *Process* P diverges *if there is an infinite sequence of actions* τ *or* τ_c *starting from* P.

Weak transitions are defined as follows: $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau} \cup \xrightarrow{\tau_c}$, while $\overset{\alpha}{\Longrightarrow}$ is $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$.

Definition 6 (Should testing). *Let* P *and* O *be processes and* $\surd$ *a special name occurring in* O *but not in* P. *We call* O *an* observer. P *should* O *iff for each* P' *such that* $P \mid O \Longrightarrow P'$ *we have* $P' \overset{\overline{\surd}}{\Longrightarrow}$. *Two processes* P *and* Q *are* should testing equivalent, *written* $P \simeq_{shd} Q$, *if, for each observer* O, P *should* O *iff* Q *should* O.

We use should testing equivalence as our basic notion of process equivalence. However, we have to restrict its applicability. In fact, we are interested in how compensation update can be realized, but compensation update is only meaningful inside transaction scopes. Thus we have to restrict our attention to well formed processes, i.e. processes that will never feature a compensation update outside a transaction scope.

Definition 7 (Well formed processes). *Predicates* $\mathsf{wf}(\bullet)$ *and* $\mathsf{wc}(\bullet)$ *characterizing well formed processes and processes with well formed compensations are defined by mutual induction in Fig. 6.*

$$
\begin{aligned}
&\mathsf{wf}(\mathbf{0})\\
&\mathsf{wf}(\overline{a}\langle \boldsymbol{v}\rangle.P) \text{ if } \mathsf{wf}(P)\\
&\mathsf{wf}(a(\boldsymbol{x}).P) \text{ if } \mathsf{wf}(P)\\
&\mathsf{wf}((\nu\, x)\, P) \text{ if } \mathsf{wf}(P)\\
&\mathsf{wf}(P \mid Q) \text{ if } \mathsf{wf}(P) \wedge \mathsf{wf}(Q)\\
&\mathsf{wf}(\langle P\rangle) \text{ if } \mathsf{wf}(P)\\
&\mathsf{wf}(t[P,Q]) \text{ if } \mathsf{wc}(P) \wedge \mathsf{wf}(Q)\\
&\mathsf{wf}(X)
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{wc}(\mathbf{0})\\
&\mathsf{wc}(\overline{a}\langle \boldsymbol{v}\rangle.P) \text{ if } \mathsf{wc}(P)\\
&\mathsf{wc}(a(\boldsymbol{x}).P) \text{ if } \mathsf{wc}(P)\\
&\mathsf{wc}(\mathsf{inst}\lfloor \lambda X.R \rfloor.P) \text{ if } \mathsf{wf}(R) \wedge \mathsf{wc}(P)\\
&\mathsf{wc}((\nu\, x)\, P) \text{ if } \mathsf{wc}(P)\\
&\mathsf{wc}(P \mid Q) \text{ if } \mathsf{wc}(P) \wedge \mathsf{wc}(Q)\\
&\mathsf{wc}(\langle P\rangle) \text{ if } \mathsf{wf}(P)\\
&\mathsf{wc}(t[P,Q]) \text{ if } \mathsf{wc}(P) \wedge \mathsf{wf}(Q)\\
&\mathsf{wc}(X)
\end{aligned}
$$

Fig. 6. Well formedness predicates

Next definition introduces n-ary contexts.

Definition 8. *An n-ary context $C[\bullet_1, \ldots, \bullet_n]$ is obtained by replacing in a process n occurrences of $\mathbf{0}$ with placeholders $\bullet_1, \ldots, \bullet_n$. Process $C[P_1, \ldots, P_n]$ is obtained by replacing inside $C[\bullet_1, \ldots, \bullet_n]$ each $\bullet_i$ with P_i.*

We describe below the conditions that we require for good encodings. Since we always deal with subcalculi of compensable processes we can use the notion of equivalence defined above for them, e.g. observers in should testing are compensable processes (not necessarily well-formed).

Definition 9 (Conditions for good encodings). *An encoding from a subcalculus $\mathcal{C}_1$ of compensable processes to a subcalculus $\mathcal{C}_2$ of compensable processes is a function $[\![\bullet]\!] : \mathcal{C}_1 \to \mathcal{C}_2$. Such an encoding is* compositional *if:*

1. $[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$;
2. *for each name substitution σ there is a name substitution σ' such that $[\![P\sigma]\!] = [\![P]\!]\sigma'$;*
3. $[\![t\,[P,Q]]\!] = C_t[[\![P]\!], [\![Q]\!]]$, *where $C_t[\bullet_1, \bullet_2]$ is a fixed binary context with parameter t.*

An encoding is correct *if for each well formed process P, P is should testing equivalent to $[\![P]\!]$. It is* divergence reflecting *if $[\![P]\!]$ diverges implies P diverges. An encoding is* good *if it is compositional, correct and divergence reflecting.*

The properties above have been taken from [19], where a general framework for proving encodability and separation results is presented, and then adapted to our setting. In particular, some of the conditions have been simplified since a few issues do not emerge in our work (e.g., since all the calculi are subcalculi of compensable processes). Condition 3, for instance, requires the transaction scope to be translated into a context in the target language, and such a condition is required for each operator in [19]. We have chosen should testing equivalence as correctness criterion. Roughly, it combines operational correspondence and success sensitiveness from [19]. Since we require also divergence reflection, using must testing [21] instead of should testing does not change our results [20].

As we already said, we will show that our encoding satisfies stricter conditions. In particular, we will replace the notion of correctness based on should testing

equivalence with one based on weak bisimilarity (we have chosen should testing instead of must testing since weak bisimilarity implies should testing [20]).

Weak bisimilarity for compensable processes extends weak early π-calculus bisimilarity with features from higher-order bisimilarity [22], since compensation update is a form of higher-order communication.

Definition 10 (Weak bisimulation). *A weak bisimulation is a symmetric binary relation $\mathcal{R}$ such that $P\mathcal{R}Q$ implies:*

- *if $P \xrightarrow{\tau} P'$ or $P \xrightarrow{\tau_c} P'$ then there is Q' such that $Q \Longrightarrow Q'$ and $P'\mathcal{R}Q'$;*
- *if $P \xrightarrow{(\tilde{z})\lambda X.R} P'$ and $\{\tilde{z}\} \cap \mathsf{fn}(Q) = \emptyset$ then there are S, Q' such that $Q \xRightarrow{(\tilde{z})\lambda X.S} Q'$, $P'\mathcal{R}Q'$ and $R\{^T/_X\}\mathcal{R}S\{^T/_X\}$ for all processes T with no free variables;*
- *if $P \xrightarrow{\alpha} P'$ with $\alpha \neq \tau, (\tilde{z})\lambda X.R$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$, then there is Q' such that $Q \overset{\alpha}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$;*
- $\mathsf{extr}(P)\mathcal{R}\,\mathsf{extr}(Q)$.

The function $\mathsf{extr}(\bullet)$ in the last condition should be instantiated to $\mathsf{extr}_{\mathsf{n}}(\bullet)$ or $\mathsf{extr}_{\mathsf{nn}}(\bullet)$ according to the chosen LTS semantics. Closure under the extraction function is required for having a compositional semantics (see [23]).

Definition 11. *Weak bisimilarity $\approx$ is the largest weak bisimulation.*

We will use the notion below as stronger form of correctness.

Definition 12. *An encoding is* bisimilarity preserving *if for each well formed process P, P is weakly bisimilar to $[\![P]\!]$.*

The lemma below proves that a bisimilarity preserving encoding is correct.

Lemma 3.1. *Let P and Q be processes. If $P \approx Q$ then $P \simeq_{shd} Q$.*

4 Parallel Recovery Can Be Implemented Using Static Recovery

In this section we compare the expressive power of parallel recovery and static recovery, considering both the cases of nested failure and non-nested failure. We present an encoding from parallel recovery to static recovery, showing that static recovery is as expressive as parallel recovery. The encoding respects the conditions of Definition 9 and Definition 12.

The encoding associates to each transaction scope a fresh name r. Compensations to be installed are left in the body of the transaction scope, protected by a protected block and guarded by an input on r. When the transaction scope is killed, an output on r, included in the static compensation, becomes enabled and can interact with the stored compensations, enabling them. Each of them also regenerates the output on r to enable further compensation elements.

$$
\begin{aligned}
(\nu\, r)\, t\, [\overline{book}.(\langle r.(\overline{unbook}|\overline{r})\rangle \mid \overline{pay}.\langle r.(\overline{refund}|\overline{r})\rangle), 0 \mid \overline{r}] &\xrightarrow{\overline{book}} \\
(\nu\, r)\, t\, [\langle r.(\overline{unbook}|\overline{r})\rangle \mid \overline{pay}.\langle r.(\overline{refund}|\overline{r})\rangle, 0 \mid \overline{r}] &\xrightarrow{\overline{pay}} \\
(\nu\, r)\, t\, [\langle r.(\overline{unbook}|\overline{r})\rangle \mid \langle r.(\overline{refund}|\overline{r})\rangle, 0 \mid \overline{r}] &\xrightarrow{t} \\
(\nu\, r)\, \langle r.(\overline{unbook}|\overline{r})\rangle \mid \langle r.(\overline{refund}|\overline{r})\rangle \mid \langle \overline{r}\rangle &\xrightarrow{\tau} \\
(\nu\, r)\, \langle r.(\overline{unbook}|\overline{r})\rangle \mid \langle (\overline{refund}|\overline{r})\rangle &\xrightarrow{\tau} \\
(\nu\, r)\, \langle \overline{unbook}|\overline{r}\rangle \mid \langle \overline{refund}\rangle &\xrightarrow{\overline{unbook}} \\
(\nu\, r)\, \langle \overline{r}\rangle \mid \langle \overline{refund}\rangle &\xrightarrow{\overline{refund}} (\nu\, r)\, \langle \overline{r}\rangle \mid \langle 0\rangle
\end{aligned}
$$

Fig. 7. Sample execution

Definition 13 (From parallel to static recovery). *Let r be a fixed fresh name. The encoding $[\![\bullet]\!]_{p2s}$ from parallel recovery processes to static recovery processes is defined as:*

$$
\begin{aligned}
[\![t\, [P, Q]]\!]_{p2s} &= (\nu\, r)\, t\, [[\![P]\!]_{p2s}, [\![Q]\!]_{p2s} \mid \overline{r}] \\
[\![\mathsf{inst}\lfloor \lambda X.Q \mid X \rfloor.P]\!]_{p2s} &= [\![P]\!]_{p2s} \mid \langle r.([\![Q]\!]_{p2s} \mid \overline{r})\rangle
\end{aligned}
$$

and maps all the other operators homomorphically to themselves.

Name r will be α-converted to different names inside different scopes.

Example 2. We apply here the translation to a simple example. Consider a transaction which books some hotel and then pays for it. In case of failure, the booking should be undone by sending a message $\overline{unbook}$, and the payment by sending a message $\overline{refund}$. For simplicity we do not consider the contents of the messages. The transaction can be modeled using parallel recovery processes as

$$t[\overline{book}.\,\mathsf{inst}\lfloor \overline{unbook} \rfloor.\overline{pay}.\,\mathsf{inst}\lfloor \overline{refund} \rfloor, 0]$$

Its translation is:

$$(\nu\, r)\, t[\overline{book}.(\langle r.(\overline{unbook}|\overline{r})\rangle \mid \overline{pay}.\langle r.(\overline{refund}|\overline{r})\rangle), 0 \mid \overline{r}]$$

Figure 7 shows a sample execution, where the hotel is booked and payed, then the transaction scope is killed and the two items of compensation are executed.

It is easy to see that the encoding is compositional. Even more, it maps all the operators but transaction scope and compensation update homomorphically to themselves.

Remark 1. We have presented the encoding in the framework of synchronous π-calculus. The same encoding however can be used for CCS [24] and asynchronous π-calculus [25], extended with the primitives for transactions and compensations. In fact the encoding does not exploit name communication nor synchrony. We have presented it in the most general setting since it is easier to restrict the approach to CCS than to generalize an approach from CCS to π-calculus.

The rest of this section is devoted to prove that $[\![\bullet]\!]_{p2s}$ is a good, bisimilarity preserving encoding (see Definitions 9 and 12). We describe in detail the case of nested failure, the case of non-nested failure requires minimum changes.

Remark 2. If we drop the requirement of well formedness, bisimilarity preservation is no more satisfied, e.g. since

$$[\![\mathsf{inst}\lfloor \lambda X.Q \mid X \rfloor.P]\!]_{p2s} = [\![P]\!]_{p2s} \mid \langle r.([\![Q]\!]_{p2s} \mid \overline{r})\rangle \xrightarrow{r}$$

while $\mathsf{inst}\lfloor \lambda X.Q \mid X \rfloor.P$ has no corresponding transition. Alternatively, one may require that actions on fresh names introduced by the translation, such as r here, are not observed by the behavioral equivalence.

While weak bisimilarity is preserved only for well formed processes, a strict relationship holds also between the behavior of a general process P and of its translation $[\![P]\!]_{p2s}$, as shown by Lemma 4.2 and Lemma 4.3. Roughly, the translation $\tilde{P}$ of a process P such that $P \xrightarrow{\alpha} P'$ evolves to some process $\tilde{P}'$ which is the translation of P'. However, this holds only up to some transformations deleting the garbage produced by the translation. To this end we exploit a *structural congruence* and an *auxiliary reduction relation.*

Definition 14 (Structural congruence). *Structural congruence on compensable processes is the minimum congruence $\equiv$ closed under the rules in Fig. 8.*

$$\begin{array}{c}
\mathbf{0} \mid P \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
(\nu\, x)\, \mathbf{0} \equiv \mathbf{0} \qquad (\nu\, x)\ (\nu\, y)\ P \equiv (\nu\, y)\ (\nu\, x)\, P \qquad \langle (\nu\, x)\, P\rangle \equiv (\nu\, x)\, \langle P\rangle \qquad (\nu\, x)\, \overline{x} \equiv \mathbf{0} \\
P \mid (\nu\, x)\ Q \equiv (\nu\, x)\ (P \mid Q) \quad \text{if } x \notin \mathsf{fn}(P) \\
t\,[(\nu\, x)\ P, Q] \equiv (\nu\, x)\ t\,[P, Q] \quad \text{if } t \neq x,\ x \notin \mathsf{fn}(Q) \\
\langle\langle P\rangle\rangle \equiv \langle P\rangle \qquad \langle P \mid Q\rangle \equiv \langle P\rangle \mid \langle Q\rangle \qquad \langle \mathbf{0}\rangle \equiv \mathbf{0}
\end{array}$$

Fig. 8. Structural congruence relation

Structural congruence includes standard rules from π-calculus, scope extrusion for the operators for transaction and compensation handling and a few rules capturing the properties of protected block. We also consider the simple garbage collection rule $(\nu\, x)\, \overline{x} \equiv \mathbf{0}$, since it simplifies our proofs.

Definition 15 (Auxiliary reduction relation). *The auxiliary reduction relation $\mapsto$ is the minimum congruence generated by the following rule:*

$(\nu\, r)\, \langle \overline{r}\rangle \mid \prod_{i\in\{1,\ldots,n\}} \langle r.(Q_i \mid \overline{r})\rangle \mapsto (\nu\, r)\, \langle \overline{r}\rangle \mid \prod_{i\in\{1,\ldots,n\}} \langle Q_i\rangle$ *if $r \notin \mathsf{fn}(Q_i)$ for each $i \in \{1, \ldots, n\}$.*

The definition below introduces *possible translations*, which generalize the concept of translation. The idea is that each process in the set of possible translations of P behaves as P. Possible translations account for the different shapes that a dynamically created compensation can have, according to how it has been built as a composition of compensation items.

Definition 16 (Possible translations). *Let r be a fixed fresh name. Given a parallel recovery process P the set of its possible translations $\{\!|P|\!\}_{p2s}$ is defined by structural induction on P and then closed under the structural congruence and the auxiliary reduction relation. More precisely:*

- *if $P = t[R, Q]$ for each decomposition $Q \equiv \prod_{i\in\{0,\ldots,n\}} Q_i$, each $\tilde{R} \in \{\!|R|\!\}_{p2s}$ and $\tilde{Q}_i \in \{\!|Q_i|\!\}_{p2s}$, we have that $(\nu\, r)\, t\left[\tilde{R} \mid \prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{Q}_i \mid \overline{r})\rangle, \tilde{Q}_0 \mid \overline{r}\right] \in \{\!|P|\!\}_{p2s}$;*
- *if $P = \mathsf{inst}\lfloor \lambda X.Q \mid X \rfloor.R$ for each $\tilde{Q} \in \{\!|Q|\!\}_{p2s}$ and each $\tilde{R} \in \{\!|R|\!\}_{p2s}$, we have that $\tilde{R} \mid \langle r.(\tilde{Q} \mid \overline{r})\rangle \in \{\!|P|\!\}_{p2s}$;*
- *for each other n-ary operator op, if $P = \mathrm{op}(Q_1, \ldots, Q_n)$ for each $\tilde{Q}_i \in \{\!|Q_i|\!\}_{p2s}$ we have that $\mathrm{op}(\tilde{Q}_1, \ldots, \tilde{Q}_n) \in \{\!|P|\!\}_{p2s}$.*

Furthermore:

- *if $\tilde{P} \in \{\!|P|\!\}_{p2s}$ and $\tilde{P}' \equiv \tilde{P}$ then $\tilde{P}' \in \{\!|P|\!\}_{p2s}$;*
- *if $\tilde{P} \in \{\!|P|\!\}_{p2s}$ and $\tilde{P}' \mapsto \tilde{P}$ then $\tilde{P}' \in \{\!|P|\!\}_{p2s}$.*

The lemma below relates the possible translations of P and of $\mathsf{extr}_{\mathsf{n}}(P)$.

Lemma 4.1. *Let $\tilde{P} \in \{\!|P|\!\}_{p2s}$. Then $\mathsf{extr}_{\mathsf{n}}(\tilde{P}) \in \{\!|\mathsf{extr}_{\mathsf{n}}(P)|\!\}_{p2s}$.*

The lemmas below relate the behavior of a process with the one of its possible translations. Namely, it will be shown that a possible translation evolves into a possible translation (this does not hold for translations). As already said, we write a compensation update $\lambda X.Q \mid X$ simply as Q.

Lemma 4.2. *Let P be a parallel recovery process and $\tilde{P} \in \{\!|P|\!\}_{p2s}$ one of its possible translations. If $P \xrightarrow{\alpha} P'$ then one of the following holds:*

1. *$\alpha \notin \{(\boldsymbol{z})Q, \tau_c\}$ and $\tilde{P} \xRightarrow{\alpha} \tilde{P}'$ with $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$;*
2. *$\alpha = (\boldsymbol{z})Q$ and $\tilde{P} \Longrightarrow (\nu\, \boldsymbol{z})\,(\tilde{P}' \mid \langle r.(\tilde{Q} \mid \overline{r})\rangle)$ where $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$ and $\tilde{Q} \in \{\!|Q|\!\}_{p2s}$;*
3. *$\alpha = \tau_c$ and $\tilde{P} \Longrightarrow \tilde{P}'$ with $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$.*

Proof. The proof is by structural induction on P, using a case analysis on the last applied rule. □

The following lemma discusses the reverse implication.

Lemma 4.3. *Let P be a parallel recovery process such that $\mathsf{noComp}(P)$ and $\tilde{P} \in \{\!|P|\!\}_{p2s}$ one of its possible translations. If $\tilde{P} \xrightarrow{\alpha} \tilde{P}'$ with $\alpha \neq r$ then $P \xRightarrow{\alpha} P'$ with $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$.*

Proof. The proof is by induction on the derivation of $\tilde{P} \in \{\!|P|\!\}_{p2s}$. □

Theorem 4.1. *Let P be a well formed process. Then $P \approx [\![P]\!]_{p2s}$.*

Proof. First note that $[\![P]\!]_{p2s} \in \{\!|P|\!\}_{p2s}$. The proof is by coinduction. We have to show that the relation $\mathcal{R} = \{(P, \tilde{P}) \mid \mathsf{wf}(P) \wedge \tilde{P} \in \{\!|P|\!\}_{p2s}\}$ is a weak bisimulation. The proof exploits Lemmas 4.1, 4.2 and 4.3. □

Corollary 1. *$[\![\bullet]\!]_{p2s}$ is a good encoding.*

5 General Dynamic Recovery Is More Expressive Than Static Recovery

In this section we compare the expressive power of general dynamic recovery and static recovery, showing that the former is more powerful. We also adapt our result to show that backward recovery is more powerful than static recovery.

The main idea is that with general dynamic recovery it is possible to check the order of execution of parallel actions by observing the compensations that they install, while this is not possible with static recovery. For instance, process $t[a.\,\mathsf{inst}\lfloor\lambda X.a'.\mathbf{0}\rfloor \mid b.\,\mathsf{inst}\lfloor\lambda X.b'.\mathbf{0}\rfloor, \mathbf{0}]$ can perform a computation with labels a, b, t, b' but no computation with labels b, a, t, b', i.e. whether b' is available or not depends on the order of execution of the parallel actions a and b. The proof of the separation result in Theorem 5.1 exploits similar arguments. The proof is based on the fact that the order of installation of compensations is not known statically because of the nondeterminism in the scheduling of parallel processes.

Before proving the theorem we need a few auxiliary notions and results.

Definition 17 (Enabling contexts). Enabling contexts $E[\bullet_1]$ *are unary contexts generated by:*
$E[\bullet_1] ::= \ \bullet_1 \ \mid \ P|E[\bullet_1] \ \mid \ E[\bullet_1]|P \ \mid \ (\nu\, x)\, E[\bullet_1] \ \mid \ t[E[\bullet_1], Q] \ \mid \ \langle E[\bullet_1]\rangle$

Definition 18. *Given a process P the* maximum choice degree $\mathrm{mcd}(P)$ *of P is the maximum number of alternatives in a nondeterministic choice inside P. The* maximum transaction nesting degree $\mathrm{mtd}(P)$ *of P is the maximum level of nesting of transaction scopes inside P.*

Next lemma shows that the maximum choice degree and the maximum transaction nesting degree of a process never increase during computations.

Lemma 5.1. *If $P \xrightarrow{\alpha} P'$ then $\mathrm{mcd}(P') \leq \mathrm{mcd}(P)$ and $\mathrm{mtd}(P') \leq \mathrm{mtd}(P)$.*

Next lemma exploits the definition above to determine structural properties of processes from their behavior.

Lemma 5.2. *Let P be a static recovery process. Assume that $P \xrightarrow{a_i} P'_i$ for each $i \in \{1, \ldots, n\}$. Assume that $n > c + t$ where c is the maximum choice degree of P and t is the maximum transaction nesting degree of P. Then there are an enabling context $E[\bullet_1]$, processes Q_1 and Q_2 and indexes j and k such that $P \Longrightarrow Q = E[Q_1|Q_2]$ with $Q_1 \xrightarrow{a_j} Q'_1$ and $Q_2 \xrightarrow{a_k} Q'_2$.*

Proof. The proof is by structural induction on P. □

We can finally prove the desired separation result.

Theorem 5.1. *There is no good encoding $[\![\bullet]\!]_{g2s}$ of compensable processes into static recovery processes.*

Proof. Suppose by contradiction that such an encoding exists. For each i let $P_i = a_i.\,\mathsf{inst}(\lambda Y_i.b_i.\mathbf{0}).\mathbf{0}$. Consider the process $P = t[\prod_{i\in\{1,\ldots,n\}} P_i, \mathbf{0}]$. Because of conditions 1 and 3 of compositional encodings, its encoding $[\![P]\!]_{g2s}$ should be of the form $C_t[\prod_{i\in\{1,\ldots,n\}}[\![P_i]\!]_{g2s}, [\![\mathbf{0}]\!]_{g2s}]$, which we will denote as $\tilde{P}$. Note that P is well formed, thus $P \simeq_{shd} [\![P]\!]_{g2s}$. Let us consider the observers $O_{j,k} = \overline{a_j}.\overline{a_k}.\overline{t}.\overline{b_k}.\overline{\surd}$ and $O'_{j,k} = \overline{a_j}.\overline{a_k}.\overline{t}.\overline{b_j}.\overline{\surd}$. For each j, k note that P should $O_{j,k}$, while P should not $O'_{j,k}$. Also, given $O_j = \overline{a_j}.\overline{\surd}$, P should O_j for each j. Thanks to correctness, $\tilde{P}$ has to pass the same tests. Test O_j can succeed only if $\tilde{P} \stackrel{a_j}{\Longrightarrow}$. Also, no action τ or τ_c should compromise the possibility of performing a_j for each j, since we are using should testing equivalence.

We show now by contradiction that $\tilde{P} \Longrightarrow Q$ for some Q such that $Q \stackrel{a_i}{\longrightarrow} Q'_i$ for each $i \in \{1,\ldots,n\}$. We assume that such a Q does not exist and build an infinite computation composed by transitions τ and τ_c, contradicting divergence reflection. Since we assume Q does not exist, in particular, for some a_i there is no transition $\tilde{P} \stackrel{a_i}{\longrightarrow}$. Thus since $\tilde{P} \stackrel{a_i}{\Longrightarrow}$ we have $\tilde{P} \stackrel{\tau}{\longrightarrow}^{+} Q_1 \stackrel{a_i}{\longrightarrow}$ where $\stackrel{\tau}{\longrightarrow}^{+}$ denotes a non empty sequence of transitions τ and τ_c. Also, Q_1 must still satisfy the tests. Since Q does not exist, there is also some a_j such that there is no transition $Q_1 \stackrel{a_j}{\longrightarrow}$. Thus we can further extend the computation. By iterating the procedure we get an infinite sequence of transitions τ and τ_c. Since P does not diverge, and the encoding has to be divergence reflecting, we have a contradiction.

Now observe that thanks to condition 2 of compositional encoding all $[\![P_i]\!]_{g2s}$ are equal up to name substitution and thus have the same maximum choice degree and maximum transaction nesting degree. Thus the maximum choice degree c and maximum transaction nesting degree t of $\tilde{P}$ do not depend on n. In particular, we can choose $n > c + t$. Thanks to Lemma 5.1 the same relation holds also for Q. Thus we can apply Lemma 5.2 to prove that $Q = E[Q_1|Q_2]$ with $Q_1 \stackrel{a_j}{\longrightarrow} Q'_1$ and $Q_2 \stackrel{a_k}{\longrightarrow} Q'_2$ for some enabling context $E[\bullet]$. We have $E[Q_1|Q_2] \stackrel{a_j}{\longrightarrow} E[Q'_1|Q_2] \stackrel{a_k}{\longrightarrow} E[Q'_1|Q'_2]$ and $E[Q_1|Q_2] \stackrel{a_k}{\longrightarrow} E[Q_1|Q'_2] \stackrel{a_j}{\longrightarrow} E[Q'_1|Q'_2]$. The final process $E[Q'_1|Q'_2]$ is the same in both the cases.

These computations can be observed using observers $O_{j,k} = \overline{a_j}.\overline{a_k}.\overline{t}.\overline{b_k}.\overline{\surd}$ and $O'_{k,j} = \overline{a_k}.\overline{a_j}.\overline{t}.\overline{b_k}.\overline{\surd}$ above. $\tilde{P} \mid O \Longrightarrow E[Q'_1|Q'_2] \mid \overline{t}.\overline{b_k}.\overline{\surd}$ for both $O = O_{j,k}$ and $O = O'_{k,j}$. From $\tilde{P}$ should $O_{j,k}$ we deduce $E[Q'_1|Q'_2] \mid \overline{t}.\overline{b_k}.\overline{\surd} \stackrel{\overline{\surd}}{\Longrightarrow}$, while from $\tilde{P}$ should not $O'_{k,j}$ we deduce that this computation cannot exist.

This is a contradiction, thus the encoding $[\![\bullet]\!]_{g2s}$ does not exist. □

The theorem above holds for both nested and non-nested failure.

Remark 3. We have presented this separation result in the framework of synchronous π-calculus. The same result however can be proved for CCS [24], extended with the primitives for transactions and compensations. In fact the used processes and observers are all CCS processes.

It is interesting to see how the result and the proof change if conditions for good encodings are modified, in particular as far as correctness is concerned. First note

that requiring bisimilarity preservation instead of correctness weakens the result. However, this weaker result can be easily extended to asynchronous compensable processes, which are obtained by disallowing continuation after the output prefix, as done for π-calculus [25]. In particular, no compensation update can become enabled because of the execution of an output action.

Corollary 2. *There is no good bisimilarity preserving encoding* $[\![\bullet]\!]_{g2s-a}$ *of asynchronous compensable processes into asynchronous static recovery processes.*

We have not been able to prove the result above without the condition of bisimilarity preservation, since it is difficult for asynchronous observers to force an order of execution for parallel actions.

Many approaches in the literature, such as [19], use as observers in the target language the encoding of the observers in the source language. In our case we can use the same observers since the target language is a sublanguage of the starting one. We can restate our results using the approach in [19], but we need some more conditions on the translation (e.g., preservation of the behavior of sequential CCS processes).

The theorem above concerns general dynamic recovery, however a similar result can be obtained for *backward recovery*. Backward recovery is easily defined in a calculus with sequential composition by requiring all the compensation updates to have the form $\lambda X.P; X$ where ; is sequential composition and X does not occur in P. It is easy to see that just having a very constrained form of backward recovery, where P is a single prefix, is enough to increase the expressive power beyond static recovery. This can be easily stated in our framework by allowing only compensation updates of the form $\lambda X.\pi.X$ where π is any prefix.

Corollary 3. *There is no good encoding* $[\![\bullet]\!]_{b2s}$ *of backward recovery processes into static recovery processes.*

Proof. It is enough to consider $P_i = a_i.\,\mathsf{inst}\lfloor \lambda Y_i.b_i.Y_i \rfloor.\mathbf{0}$ instead of the process $P_i = a_i.\,\mathsf{inst}\lfloor \lambda Y_i.b_i.\mathbf{0} \rfloor.\mathbf{0}$ in the proof of Theorem 5.1. □

From the results of previous section we also deduce that both general dynamic recovery and backward recovery are more expressive than parallel recovery.

6 Applications and Related Works

We discuss here how to apply the results in sections 4 and 5 to other calculi and languages in the literature. The calculi more related to ours are the so-called interaction-based calculi [3–5, 9], which are obtained by adding primitives for compensation handling on top of concurrent calculi such as π-calculus [2] or Join [26]. These calculi differ on many design choices. The main differences are summarized in Table 1 and their impact on our results discussed below.

Dcπ. Dcπ [9] is a calculus with parallel recovery based on asynchronous π-calculus [25]. For this reason, compensation update is allowed only after input

prefix. Actually, in dcπ, input prefix and compensation update are combined in an atomic primitive $a(\boldsymbol{x})\%Q.P$ that, after receiving values $\boldsymbol{v}$ on channel a, continues as $P\{\boldsymbol{v}/\boldsymbol{x}\}$ and adds $Q\{\boldsymbol{v}/\boldsymbol{x}\}$ in parallel to the current compensation. The same behavior can be obtained in parallel recovery processes by writing $a(\boldsymbol{x}).\,\mathsf{inst}\lfloor\lambda X.Q \mid X\rfloor.P$, thanks to priority of compensation update. Thus dcπ can be seen as the asynchronous fragment of parallel recovery processes with nested failure where compensation update can occur only after input prefix. Both the encoding in Section 4 and the separation result for asynchronous calculi in Corollary 2 can be easily adapted to dcπ.

Webπ and Webπ_∞. Webπ [5] is a calculus with static recovery based on asynchronous π-calculus [25]. It provides timed transactions, which add an orthogonal degree of expressive power. Its untimed fragment, Webπ_∞ [23] instead corresponds exactly to the asynchronous fragment of static recovery with non-nested failure where all messages are inside protected blocks. The encoding in Section 4 can be adapted to both the calculi. The main change required is to implement protected block using a transaction scope with bound name. Also the separation result in Corollary 2 can be easily applied to the two calculi.

πt-calculus. The πt-calculus [3] is based on asynchronous π-calculus. When a component inside a transaction aborts, abortion or completion of parallel components is waited for. Then the compensation of the transaction (called failure manager) is executed, followed by the parallel composition of the compensations of the already terminated subtransactions. It is difficult to adapt our encoding to πt-calculus, since this will require to change the semantics of abortion allowing a transaction to abort even if it contains protected blocks. On the other hand the separation result in Corollary 2 can be applied, referred to an extension of the πt-calculus where the failure manager can be updated dynamically.

C-join. C-join [4] is a calculus with static recovery built on top of Join calculus [26]. However here transactions can be dynamically merged, and their

Table 1. Features of interaction-based calculi and languages

	underlying language	compens. definition	nested vs non-nested	protection operator	encoding applicable	separation res. applicable
dcπ	asynch. π	parallel	nested	yes	yes	asynch.
Webπ/Webπ_∞	asynch. π	static	non-nested	implem.	yes	asynch.
πt	asynch. π	static	nested	no	no	asynch.
C-join	Join	static	nested	no	yes[a]	no
SOCK	-	dynamic	nested	implem.	yes	no
COWS	-	static	nested	yes	yes	no
Jolie	-	dynamic	nested	implem.	yes	no
WS-BPEL	-	static	nested	implem.	yes	no

[a] If a protection operator is added.

compensations are composed in parallel, thus obtaining some form of parallel recovery. Our encoding is not directly applicable since C-join has no protected block operator, but becomes applicable as soon as such an operator is introduced. As far as the separation result is concerned, join patterns are more powerful than π-calculus communication, and we conjecture that they can be used to implement general dynamic recovery.

Service oriented calculi. Many service oriented calculi have been recently proposed [27–32]. Long running transactions are an important aspect of service oriented computing thus many of these calculi include primitives for compensation handling. We discuss here the ones more related to our approach.

SOCK [28] is a language for composing service invocations and definitions using primitives from sequential languages and concurrent calculi. It has been extended with primitives for general dynamic recovery in [12]. Our encoding can be applied to SOCK using signals for mimicking CCS communication. Actually, in SOCK, the protected block is just used in the definition of the semantics, but it can be implemented too. Since SOCK has no restriction operator, fresh signal names should be statically generated, and the behavioral correspondence result should be restated along the lines of Remark 2. The separation result instead does not apply: SOCK services are stateful, and the state can be used to keep track of the order of execution of parallel activities. All the observations made for SOCK hold also for Jolie [33, 34], a service oriented language based on it.

COWS [30] communication is in the style of fusion calculus [35]. COWS has a kill primitive and a protected block. This allows to program static recovery (see [30]). Our encoding can be applied to program also parallel recovery. The separation result instead cannot be easily extended, since COWS communication and kill have priorities, thus allowing parallel processes to influence each other.

Other service oriented calculi include only mechanisms for exception handling [31] or notification of session failure [27, 32].

Compensable flow calculi. Calculi based on the compensable flow approach such as SAGAs calculi [8] or StAC [7], use backward recovery for sequential activities and parallel recovery for parallel ones. Thus our separation result does not apply. Also, since there is no communication, atomicity constraints are less strong. However we are not aware of good encodings of compensable flow calculi into static recovery calculi. For instance, the mapping in [6] of cCSP [10] into the conversation calculus [31] is not compositional.

WS-BPEL. WS-BPEL [1] is the de-facto standard for web services composition. Compensations are statically defined, and they are composed using backward recovery for sequential subtransactions, and parallel recovery for parallel ones. Our separation result does not apply because of the reasons discussed for SOCK and for compensable flow calculi. As far as the encoding is concerned, the same approach used for SOCK can be applied.

Future work. As we already discussed throughout the paper, many open issues concerning the expressive power of mechanisms for long running transactions

remain. In fact this topic, while relevant, has been neglected until now: a few papers, such as [36, 37], study the expressive power of primitives for interruption, more than primitives for compensation as in our case. We think that the techniques presented in this paper can be successfully applied to answer some of the open issues. We refer in particular to the analysis of whether nested failure can be implemented using non-nested failure, and to the encodability of BPEL-style recovery into static recovery. We conjecture that this encoding is possible, thus BPEL-style recovery could be defined as a macro on top of static recovery. After those problems have been analyzed in a simple setting, additional work is required to transfer the results to other calculi/languages. Another important topic that deserves further investigation is the impact of communication primitives more powerful than π-calculus message passing, such as join patterns, on our separation result. It would also be interesting to generalize the techniques of this paper to deal with languages for adaptation [38], since dynamic compensations can be seen as an approach for adaptation of compensations.

Acknowledgments. We thank R. Bruni, F. Montesi, G. Zavattaro, F. Tiezzi and the anonymous reviewers for useful suggestions and comments.

References

1. 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`
2. Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Inf. Comput. 100(1), 1–40, 41–77 (1992)
3. Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 124–138. Springer, Heidelberg (2003)
4. Bruni, R., Melgratti, H., Montanari, U.: Nested commits for mobile calculi: Extending join. In: Proc. of IFIP TCS 2004, pp. 563–576. Kluwer, Dordrecht (2004)
5. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
6. Caires, L., Ferreira, C., Vieira, H.: A process calculus analysis of compensations. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 87–103. Springer, Heidelberg (2009)
7. Butler, M.J., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 87–104. Springer, Heidelberg (2004)
8. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Proc. of POPL 2005, pp. 209–220. ACM Press, New York (2005)
9. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 201–215. Springer, Heidelberg (2009)
10. Butler, M.J., Hoare, C., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)

11. Garcia-Molina, H., et al.: Coordinating multi-transaction activities. Technical Report CS-TR-2412, University of Maryland, Dept. of Computer Science (1990)
12. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: Proc. of ACSD 2008, pp. 190–199. IEEE Computer Society Press, Los Alamitos (2008)
13. Weimer, W., Necula, G.: Finding and preventing run-time error handling mistakes. In: Proc. of OOPSLA 2004, pp. 419–431. ACM Press, New York (2004)
14. Lanese, I., Vaz, C., Ferreira, C.: On the expressive power of primitives for compensation handling, TR (2010), http://www.cs.unibo.it/~lanese/publications/fulltext/TR-ESOP2010.pdf
15. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundamenta Informaticae 95(1), 73–102 (2009)
16. Parrow, J.: Expressiveness of process algebras. In: Proc. of the LIX Colloquium on Emerging Trends in Concurrency Theory. ENTCS, vol. 209, pp. 173–186. Elsevier, Amsterdam (2008)
17. Palamidessi, C.: Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In: Proc. of POPL 1997, pp. 256–265 (1997)
18. Nestmann, U.: What is a "good" encoding of guarded choice? Inf. Comput. 156(1-2), 287–319 (2000)
19. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 492–507. Springer, Heidelberg (2008)
20. Rensink, A., Vogler, W.: Fair testing. Inf. Comput. 205(2), 125–198 (2007)
21. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. 34, 83–133 (1984)
22. Thomsen, B.: Calculi for Higher Order Communicating Systems. PhD thesis, Imperial College (1990)
23. Mazzara, M., Lanese, I.: Towards a unifying theory for web services composition. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 257–272. Springer, Heidelberg (2006)
24. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
25. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
26. Fournet, C., Gonthier, G.: The reflexive CHAM and the join-calculus. In: Proc. of POPL 1996, pp. 372–385. ACM Press, New York (1996)
27. Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., Zavattaro, G.: SCC: a Service Centered Calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
28. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
29. Lanese, I., Martins, F., Vasconcelos, V., Ravara, A.: Disciplining orchestration and conversation in service-oriented computing. In: Proc. of SEFM 2007, pp. 305–314. IEEE Computer Society Press, Los Alamitos (2007)
30. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)

31. Vieira, H., Caires, L., Seco, J.: The conversation calculus: A model of service-oriented computation. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008)
32. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 19–38. Springer, Heidelberg (2008)
33. Montesi, F., Guidi, C., Zavattaro, G.: Composing services with JOLIE. In: Proc. of ECOWS 2007, pp. 13–22. IEEE Computer Society Press, Los Alamitos (2007)
34. Jolie team: Jolie website (2009), `http://www.jolie-lang.org/`
35. Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: Proc. of LICS 1998, pp. 176–185 (1998)
36. Aceto, L., Fokkink, W., Ingólfsdóttir, A., Nain, S.: Bisimilarity is not finitely based over bpa with interrupt. Theor. Comput. Sci. 366(1-2), 60–81 (2006)
37. Bravetti, M., Zavattaro, G.: On the expressive power of process interruption and compensation. Math. Stru. Comp. Sci. 19(3), 565–599 (2009)
38. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009)