On the Expressive Power of Primitives for Compensation Handling☆

Ivan Lanese*

Focus Team, Università di Bologna/INRIA, Italy

Cátia Vaz

INESC-ID / DEETC, ISEL, Instituto Politécnico de Lisboa, Portugal

Carla Ferreira

CITI / Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal

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., in the case of web services) the problem is usually tackled using *long running transactions* and *compensations*. A long running transaction is a computation that either successfully terminates, or it aborts. In case of abort, a compensation is executed to take the system to a consistent state.

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 items and (iii) *general dynamic recovery*, where more refined ways of composing compensation items 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. We prove the two results both in a synchronous and in an asynchronous setting.

Keywords: Long running transactions, compensations, encodings, separation results, process calculi.

☆Research supported by the Project FET FP7-231620 HATS, the Project ANR-ARPEGE 2010 AEOLUS and by FCT grant SFRH/BD/45572/2008.

*Corresponding author

Email addresses: `lanese@cs.unibo.it` (Ivan Lanese), `cvaz@cc.isel.ipl.pt` (Cátia Vaz), `carla.ferreira@di.fct.unl.pt` (Carla Ferreira)

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, results arising from computations may be outside the desired range of values, etc. Nevertheless, applications should provide to users reliable results. 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 [1]), this problem is usually tackled using the concept of *long running transaction*. A long running transaction is a computation that either succeeds, or it aborts. In case of abort a *compensation* is executed taking the system to a consistent state, possibly different from the one in which the transaction started. This weakens the constraints of ACID transactions (atomicity, consistency, isolation, durability) from database theory [2]. This is needed since it is difficult to guarantee ACID properties when transactions can last for a long time (e.g., if human interaction is involved), and when some actions cannot be undone (e.g., the sending of an e-mail).

In the literature there are different proposals of primitives for specifying and programming long running transactions, from the Java $\mathsf{try}\ P\ \mathsf{catch}\ e\ Q$[1], where code Q is in charge of managing exception e raised inside code P, to the complex mechanisms of WS-BPEL [3] (the Oasis 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 work trying to formally compare the expressive power of the proposed mechanisms just started. This problem is made hard by the fact that different primitives for long running transactions are provided 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, or made available via dedicated libraries; 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

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

the literature, to the same underlying language, so to have a clearer comparison. We have chosen the simplest possible underlying language able to model concurrent and mobile systems: the π-calculus [4]. Most of our results can also be stated in CCS [5], but we preferred the π-calculus since it is the common core of many existing approaches. Also, it is easier to transfer the results from π-calculus to CCS than the other way around. Anyway, 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 since the compensation code following the catch is fixed (clearly, changes in the state may allow to obtain different behaviors). Static recovery is also the classic approach of interaction-based models [6, 7, 8, 9]. For dynamic recovery we consider two different possibilities: in *parallel recovery* the compensation is incrementally built as parallel composition of simpler compensation items, while in *general dynamic recovery* compensations can be both updated and replaced. Parallel recovery is commonly used [10, 11, 3] to execute compensations of subtransactions when a transaction aborts, and it is the mechanism dcπ [12] and ATc [13] are based upon. Most of the compensable flow approaches [14, 10, 11], 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* [15]. Backward recovery has also been applied to Java in [16]. Backward recovery is the main instance of general dynamic recovery, which has been proposed in [17].

This paper compares the expressive power of static recovery, parallel recovery and general dynamic recovery in the context of π-calculus. We consider both synchronous [4] and asynchronous [18] π-calculus. Our main results are:

- a compositional encoding of parallel recovery into static recovery, applicable both in the synchronous and in the asynchronous case;
- separation results showing that no similar encoding exists from general dynamic recovery (neither from backward recovery) to static recovery.

We also compare *compensable processes*, our calculus with general dynamic recovery, with other calculi in the literature, discussing how the results above can be applied to them.

This paper is an extended version of [19]. With respect to [19], we have simplified the semantics of our calculus by considering a local priority for compensation update instead of a global one. This also makes the calculus more suitable for a distributed setting (even if we do not consider distribution in the present paper). We consider here three semantics for transaction nesting, called aborting (nested failure in [19]), preserving (non-nested failure in [19]) and discarding (not considered in [19]). Also, here there are full proofs for all the results. Furthermore, the separation result in the asynchronous case is stated

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

Figure 1: (Synchronous) π-calculus processes.

and proved using weaker hypotheses thanks to a new proof technique. Besides this, the whole paper has been thoroughly revised and improved.

Structure of the work. Section 2 introduces the primitives for long running transactions and compensations we use, defining also their formal semantics. Section 3 discusses the conditions that an encoding has to satisfy to be considered a good means for language comparison. 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 other calculi in the literature and draws some conclusions.

2. Primitives for Compensations

2.1. Syntax

In this section we formalize in the framework of (synchronous and asynchronous) π-calculus [4, 18] 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. We also define their asynchronous fragments. The syntax of all our calculi relies on a countable set of names N, ranged over by lower case letters. We use $\vec{x}$ to denote a tuple $x_1, \cdots, x_n$ of such names, for some $n \geq 0$, and $\{\vec{x}\}$ denotes the set of elements in the tuple. We use $\{\vec{v}/\vec{x}\}$ for denoting the substitution of names in $\vec{v}$ for names in $\vec{x}$, and we use a similar notation for substitutions of processes for process variables (introduced later). As already said, our calculi are built on top of π-calculus (if nothing is said, we refer to the synchronous one), whose syntax is in Fig. 1.

Prefixes in π-calculus can be either outputs $\overline{a}\langle\vec{v}\rangle$ of a tuple of names $\vec{v}$ on a channel named a, or corresponding inputs $a(\vec{x})$, receiving a tuple of names $\vec{v}$ on

P, Q	$::=$		*Asynchronous π-calculus processes*
		$\mathbf{0}$	(Inaction)
		$\overline{a}\langle\vec{v}\rangle$	(Output message)
	$\mid$	$\sum_{i\in I} a_i(\vec{x}_i).P_i$	(Input guarded choice)
	$\mid$	$!a(\vec{x}).P$	(Input guarded replication)
	$\mid$	$P \mid Q$	(Parallel composition)
	$\mid$	$(\nu\, x)\ P$	(Restriction)

Figure 2: Asynchronous π-calculus processes.

P, Q	$::=$		*Static recovery processes*
		$\ldots$	(π-calculus processes)
	$\mid$	$t[P, Q]$	(Transaction scope)
	$\mid$	$\langle P\rangle$	(Protected block)

Figure 3: Static recovery processes.

a channel named a and replacing them for names in $\vec{x}$. 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 $\overline{a}\langle\vec{v}\rangle$ when $\vec{v}$ is empty, and a for $a(\vec{x})$ when $\vec{x}$ is empty. We also write $(\nu\, \vec{x})$ for $(\nu\, x_1) \cdots (\nu\, x_n)$ when $\vec{x} = x_1, \ldots, x_n$. 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, writing e.g. $\overline{a}\langle\vec{v}\rangle$ instead of $\overline{a}\langle\vec{v}\rangle.\mathbf{0}$.

Asynchronous π-calculus [18] is simply defined as the fragment of (synchronous) π-calculus obtained by removing the continuation after output. In other words, instead of guarded choice $\sum_{i\in I} \pi_i.P_i$, in asynchronous π-calculus only input guarded choice $\sum_{i\in I} a_i(\vec{x}_i).P_i$ is allowed, and similarly for replication. Output prefixes, called messages in the asynchronous case, can only occur as parallel terms. The syntax of asynchronous π-calculus is summarized in Fig. 2.

The formal description of the semantics will be given in Section 2.2 (see also [4]), after having presented the syntax of the extensions dealing with error recovery.

The first, and simpler, extension that we consider corresponds to *static recovery*. Its syntax is described in Fig. 3. The extension can be applied to both synchronous and asynchronous π-calculus.

Static recovery can be programmed by adding just two constructs to standard 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 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

$P, Q ::=$		*Compensable processes*
	$\ldots$	(Static recovery processes)
$\mid$	X	(Process variable)
$\mid$	$\mathsf{inst}\lfloor \lambda X.Q \rfloor.P$	(Compensation update)

Figure 4: Compensable processes.

both from the body P of the transaction scope and from external processes. Error notifications are simply 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 sorting system similar, e.g., to the one in [20]. We will not consider this issue any more. Protected block $\langle P \rangle$ behaves as process P, but it is not killed in case of failure of a transaction scope enclosing it. Note that a transaction scope $t[P, Q]$ is similar to a Java-like block $\mathsf{try}\ P\ \mathsf{catch}\ t\ Q$ where an output $\overline{t}$ on t raises the exception. However in the case of transactions the fault can be directed to any transaction scope, while for try-catch it is always directed to the nearest enclosing one.

Compensable processes, which realize general dynamic recovery, extend static recovery processes. The main difference is that in compensable processes 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 compensable processes extends the one of static recovery processes with the compensation update operator and process variables (see Fig. 4). We use X to range over process variables.

We define for compensable processes the usual notions of free and bound names. Names in $\vec{x}$ are bound in $a(\vec{x}).P$, 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.

Static recovery processes are compensable processes where the compensation update operator 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 may 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. 4. *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. We define the corresponding asynchronous classes, *asynchronous compensable processes* $\mathcal{ACP}$, *asynchronous parallel recovery processes* $\mathcal{APP}$, and *asynchronous static recovery processes* $\mathcal{ASP}$, by extending asynchronous π-calculus instead of synchronous π-calculus.

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, and similarly for the corresponding asynchronous classes. In order to formally answer these questions we first need to define the semantics of compensable processes.

2.2. Operational Semantics

In this section we define the operational semantics of compensable processes. We need however some auxiliary definitions. First, when a transaction scope $t[P, Q]$ is killed, part of its body P may be preserved, in particular the part composed of protected blocks.

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[2]. When the enclosing transaction scope is killed, its subtransactions may be aborted, preserved or discarded. The aborting semantics is used by SAGAs calculi [11], WS-BPEL [3], and others. The preserving semantics is for instance the approach of Webπ [8]. Finally, the discarding semantics has been proposed by ATc [13] and TransCCS [21]. We consider all the three possibilities, since they just differ in the definition of function $\mathsf{extr}(\bullet)$. Our results hold in all the cases. One can simulate the preserving semantics using the two others by protecting each transaction scope using a protected block, while it is not clear whether the opposite simulations are possible. Also, the relationships between aborting and discarding semantics are not clear. Clarifying these points is left for future work.

Definition 2 (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. 5. 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 $\mathsf{extr_d}(t\,[P, Q]) = \mathbf{0}$.

There is no need to define $\mathsf{extr_a}(X)$, $\mathsf{extr_p}(X)$ or $\mathsf{extr_d}(X)$ since we only consider processes without free variables, thus X can occur only inside the compensation update primitive.

[2]The πt-calculus [6] has yet another approach: it will be discussed in the related work section.

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

Figure 5: Extraction function with nested failure.

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

Figure 6: No pending compensation update predicate.

We also need an auxiliary predicate $\mathsf{noComp}(P)$ which is true iff process 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. The interested reader is referred to [22] for a detailed discussion on this topic. Here we consider just a local form of priority: compensation update has priority w.r.t. other transitions involving the same transaction scope, but not w.r.t. transitions in other transaction scopes. On one side, this approach is easier to implement in a distributed setting than the one in [22, 19], on the other side, it offers the same guarantees that only up-to-date compensations are executed. 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 inside the same transaction scope.

Definition 3 ($\mathsf{noComp}(\bullet)$ predicate). The predicate $\mathsf{noComp}(P)$ that verifies that there are no pending compensation updates inside process P is true in the cases specified in Fig. 6 and false otherwise.

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 exploit to this end a *labeled transition system* (LTS). LTSs are defined as follows.

Definition 4 (Labeled transition system). A labeled transition system is a triple $(\mathcal{P}, \Lambda, \rightarrow)$ where $\mathcal{P}$ is a set of processes, Λ is a set of labels and $\rightarrow \subseteq \mathcal{P} \times \Lambda \times \mathcal{P}$ is the transition relation. If $P, Q \in \mathcal{P}$ and $\alpha \in \Lambda$, then $(P, \alpha, Q) \in \rightarrow$ is written as $P \xrightarrow{\alpha} Q$.

In the case of compensable processes, we use $a(\vec{v})$, $(\vec{w})\overline{a}\langle\vec{v}\rangle$, τ and $(\vec{w})\lambda X.Q$ as labels. The first three forms of labels are as in π-calculus: $a(\vec{v})$ is the input

of a tuple of values $\vec{v}$ on channel a, $(\vec{w})\overline{a}\langle\vec{v}\rangle$ the corresponding output, where names in vector $\vec{w}$ are extruded, 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, $(\vec{w})\lambda X.Q$, is peculiar of compensable processes and corresponds to compensation update. Again, vector $\vec{w}$ contains the extruded names. Extruded names form a vector so to fix the order the names will be reintroduced in the term by closure rules such as (L-Close), but this order will become irrelevant after we introduce structural congruence in Definition 21. For this reason we consider labels that differ only on the order of extruded names as equal.

We write a for $a(\vec{v})$ and $\overline{a}$ for $\overline{a}\langle\vec{v}\rangle$ if $\vec{v}$ is empty. We may use t instead of a to emphasize that the name is used for error notification. Names in $\vec{w}$ are bound in $(\vec{w})\overline{a}\langle\vec{v}\rangle$ and $(\vec{w})\lambda X.Q$. Other names are free. Functions $\mathsf{fn}(\bullet)$, $\mathsf{bn}(\bullet)$ and $\mathsf{n}(\bullet)$ are extended to labels accordingly. We drop the vector of bound names $(\vec{w})$ from labels if it is empty.

Definition 5 (Operational semantics). The operational semantics of compensable processes $\mathcal{CP}$ with aborting semantics for transaction nesting is the minimum LTS closed under the rules in Fig. 7 (symmetric rules are considered for (L-Par) and (L-Close)). The operational semantics of compensable processes $\mathcal{CP}$ with preserving semantics (resp. discarding semantics) for transaction nesting is the minimum LTS closed under the rules in Fig. 7 (symmetric rules are considered for (L-Par) and (L-Close)), but where function $\mathsf{extr_a}(\bullet)$ in rules (L-Recover-out) and (L-Recover-in) is replaced by function $\mathsf{extr_p}(\bullet)$ (resp. $\mathsf{extr_d}(\bullet)$).

The first eight rules are standard π-calculus rules [4], 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 $\vec{v}$ 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-Res) is the classic rule for restriction, allowing to propagate label α provided that the restricted name x does not occur inside α. Rule (L-Open) allows to extrude bound name z. Rule (L-Close) performs communication, synchronizing an input $x(\vec{v})$ and a corresponding output $(\vec{z})\overline{x}\langle\vec{v}\rangle$. If the vector $\vec{z}$ of extruded names is not empty, restrictions for these names 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 and that there is no pending compensation update to be executed. Rule (L-Recover-out) allows external processes to abort a transaction scope via an output $\overline{t}$. The resulting process is composed by 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. The condition $\mathsf{noComp}(P)$ ensures that there are no pending compensation updates in P. Rule

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

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

$$\text{(L-Open)}\quad \frac{P \xrightarrow{(\vec{w})\overline{x}\langle\vec{v}\rangle} P' \quad z \neq x \quad z \in \{\vec{v}\} \setminus \{\vec{w}\}}{(\nu\, z)\, P \xrightarrow{(z\vec{w})\overline{x}\langle\vec{v}\rangle} P'}$$

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

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

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

$$\text{(L-Inst)}\quad \frac{}{\mathsf{inst}\lfloor\lambda X.Q\rfloor.P \xrightarrow{\lambda X.Q} P} \qquad \text{(L-Open-inst)}\quad \frac{P \xrightarrow{(\vec{w})\lambda X.Q} P' \quad z \in \mathsf{fn}(Q) \setminus \{\vec{w}\}}{(\nu\, z)\, P \xrightarrow{(z\vec{w})\lambda X.Q} P'}$$

$$\text{(L-Scope-close)}\quad \frac{P \xrightarrow{(\vec{z})\lambda X.R} P' \quad \{\vec{z}\} \cap (\mathsf{fn}(Q) \cup \{t\}) = \emptyset}{t[P, Q] \xrightarrow{\tau} (\nu\, \vec{z})\, t[P', R\{Q/X\}]} \qquad \text{(L-Block)}\quad \frac{P \xrightarrow{\alpha} P'}{\langle P\rangle \xrightarrow{\alpha} \langle P'\rangle}$$

Figure 7: LTS for compensable processes.

(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 while rule (L-Open-inst) allows to extrude bound names occurring in a compensation update. Rule (L-Scope-close) updates the compensation of a transaction scope. If the vector $\vec{z}$ of extruded names is not empty, restrictions for these names are reintroduced, similarly to what rule (L-Close) does. 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 $\mathsf{extr}(\bullet)$.

Example 1. We give here a few examples of transitions[3].

- 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:
  $t[\,\overline{t}.0 \mid \overline{a}.0, Q] \xrightarrow{\tau} \langle Q\rangle$
- 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:
  $t[\mathsf{inst}\lfloor\lambda X.0\rfloor.\overline{a}.0, Q] \xrightarrow{\tau} t[\overline{a}.0, 0]$

Note that none of the last three transaction scopes can be killed, since there is a pending compensation update which has priority over the kill action.

We can now formalize the idea of Definition 1, taking into account also the semantics of processes.

Definition 6 (Subcalculus). A calculus is defined by its labeled transition system $(\mathcal{P}, \Lambda, \rightarrow)$.

Let $(\mathcal{P}_1, \Lambda_1, \rightarrow_1)$ be a calculus. A calculus $(\mathcal{P}_2, \Lambda_2, \rightarrow_2)$ is a subcalculus of $(\mathcal{P}_1, \Lambda_1, \rightarrow_1)$ if $\mathcal{P}_2 \subseteq \mathcal{P}_1$, $\Lambda_2 \subseteq \Lambda_1$ and $\rightarrow_2 \subseteq \rightarrow_1$. Furthermore, for each $P \xrightarrow{\alpha}_1 Q$, if $P \in \mathcal{P}_2$ then $P \xrightarrow{\alpha}_2 Q$.

We will drop the references to the set of labels Λ and the transition relation $\rightarrow$ when they are understood. Given a calculus $(\mathcal{P}_1, \Lambda_1, \rightarrow_1)$, one can define a subcalculus $(\mathcal{P}_2, \Lambda_2, \rightarrow_2)$ of $(\mathcal{P}_1, \Lambda_1, \rightarrow_1)$ just by specifying the set of processes $\mathcal{P}_2$ and considering the induced transition relation and set of labels. One only has to check that the set of processes $\mathcal{P}_2$ is closed under the transition relation.

Proposition 1. *Static recovery processes $\mathcal{SP}$ are a subcalculus of parallel recovery processes $\mathcal{PP}$. Parallel recovery processes $\mathcal{PP}$ are a subcalculus of compensable processes $\mathcal{CP}$.*

[3] To simplify the presentation we discard some garbage. This can be done using the notion of structural congruence in Definition 21.

Asynchronous static recovery processes $\mathcal{ASP}$ are a subcalculus of asynchronous parallel recovery processes $\mathcal{APP}$. Asynchronous parallel recovery processes $\mathcal{APP}$ are a subcalculus of asynchronous compensable processes $\mathcal{ACP}$.

Asynchronous static recovery processes $\mathcal{ASP}$ are a subcalculus of static recovery processes $\mathcal{SP}$. Asynchronous parallel recovery processes $\mathcal{APP}$ are a subcalculus of parallel recovery processes $\mathcal{PP}$. Asynchronous compensable processes $\mathcal{ACP}$ are a subcalculus of compensable processes $\mathcal{CP}$.

Proof. By induction on the labeled transition system. □

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. The ones that had more influence on our work are [23, 24, 25, 26, 27]. The choice of the conditions determines the level of abstraction used when comparing the different languages, and is fundamental to decide when an encoding can be safely used. For instance, some conditions allow to use the encoded process in any context in the target language, while others allow only to use it inside contexts which are in the image of the encoding. Also, some encodings can be applied independently to different components of the system, and the encoded components can then be composed, while other encodings have to be applied to the whole system at once. Thus the choice of the conditions depends on the aim of the encoding, and there are no universally good sets of conditions. Another main point is that 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, i.e. under weaker conditions. However, it is also 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. We will also exploit slightly different conditions for adapting our results to asynchronous compensable processes.

The aim of this work is to understand whether some primitives for long running transactions, namely transactions with dynamic compensation update, can be defined as macros on top of a more basic language, or they need to be defined as part of the core language. For these reasons, we choose conditions ensuring the following intuitive properties:

1. macros can be replaced in place, without changing the rest of the program;
2. the behavior of the program after macros have been replaced by their definitions matches the expected behavior of the original program.

These two requirements match the two kinds of conditions proposed in [27]:

1. syntactic conditions on the form of the encoding;

2. semantic conditions specifying the kind of behavior that the encoding should preserve.

Our framework is however different from the one of [27], where encodings between arbitrary languages are considered, since we always analyze encodings of a calculus into one of its subcalculi, and we always consider calculi which are subcalculi of compensable processes.

For these reasons, we define an encoding as follows.

Definition 7 (Encoding). For us an encoding is a function $[\![\bullet]\!] : \mathcal{P}_1 \to \mathcal{P}_2$ from a subcalculus $\mathcal{P}_1$ of compensable processes to a subcalculus $\mathcal{P}_2$ of $\mathcal{P}_1$. Given a process $P_1 \in \mathcal{P}_1$ we may refer to $[\![P_1]\!]$ as its translation.

We will now define syntactic and semantic conditions ensuring that encodings satisfy the intuitive requirements above. For stating the syntactic conditions we need to define contexts.

Definition 8 (Context). 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.

An encoding is *compositional* if it respects the syntactic conditions below.

Definition 9 (Compositional encoding). An encoding $[\![\bullet]\!] : \mathcal{P}_1 \to \mathcal{P}_2$ is *compositional* if:

1. for each $P, Q \in \mathcal{P}_1$, $[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$;
2. for each name substitution σ there is a name substitution σ' such that, for each $P \in \mathcal{P}_1$, $[\![P\sigma]\!] = [\![P]\!]\sigma'$;
3. for each $P, Q \in \mathcal{P}_1$ and each name t, $[\![t\,[P, Q]]\!] = C_t[[\![P]\!], [\![Q]\!]]$, where $C_t[\bullet_1, \bullet_2]$ is a fixed binary context with parameter t.

We will now discuss the semantic conditions. First of all, an encoding should not introduce divergence, which is defined as below.

Definition 10 (Divergence). Process P *diverges* if there is an infinite sequence of processes P_i for $i \in \mathbb{N}$ such that $P = P_0$ and $P_i \xrightarrow{\tau} P_{i+1}$ for each $i \in \mathbb{N}$.

Definition 11 (Divergence reflecting encoding).
An encoding $[\![\bullet]\!] : \mathcal{P}_1 \to \mathcal{P}_2$ is *divergence reflecting* if for each $P \in \mathcal{P}_1$, $[\![P]\!]$ diverges implies P diverges.

The behavior of a program and of its encoding should be indistinguishable with respect to some notion of observational equivalence. We define below the notions of observational equivalence that we use. Since we consider weak equivalences, we start by defining weak transitions.

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

Figure 8: Well formedness predicates.

Definition 12 (Weak transitions). Weak transitions $\Longrightarrow$ and $\overset{\alpha}{\Longrightarrow}$ are defined as follows: $\Longrightarrow$ is the reflexive and transitive closure of $\overset{\tau}{\rightarrow}$, while $\overset{\alpha}{\Longrightarrow}$ is $\Longrightarrow\overset{\alpha}{\rightarrow}\Longrightarrow$ if $\alpha \neq \tau$, $\Longrightarrow$ otherwise.

We base our separation results on *should testing* [28]. We consider should testing instead of must testing [29] since it is more suited for dealing with processes that may diverge, and since its relations with weak bisimilarity [30] and may testing [29] (other equivalences that we use, defined later) are more straightforward. Should testing has been used in the framework of long running transactions in [31]. Anyway our results can be stated also using must testing and may testing instead of should testing.

Definition 13 (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 $P' \overset{\surd}{\Longrightarrow}$. 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 will require preservation of should testing equivalence as correctness condition for an encoding, however we have to restrict the set of processes the condition is applied to. 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 processes that will never feature a compensation update outside a transaction scope. A syntactic characterization of this class of processes is given by the definition of well formed processes below.

Definition 14 (Well formed processes). Predicates $\mathsf{wf}(\bullet)$ and $\mathsf{wc}(\bullet)$ characterizing well formed processes and processes with well formed compensations respectively are defined by mutual induction. They are true in the cases specified in Fig. 8 and false otherwise.

The correctness of the characterization is given by the two lemmas below: the first one shows that well formed processes are closed under arbitrary transitions,

the second one shows that a well formed process never asks the environment to perform a compensation update.

Lemma 1. *Let P be a well formed process. If $P \xrightarrow{\alpha} P'$ then P' is well formed.*

Lemma 2. *Let P be a well formed process. If $P \xrightarrow{\alpha} P'$ then α is not of the form $(\vec{z})\lambda X.R$.*

We defer the proofs of those lemmas to Section 4. See Lemma 14 and Lemma 13 respectively.

We can now state the correctness requirement that we impose on encodings.

Definition 15 (Correct encoding). An encoding $[\![\bullet]\!] : \mathcal{P}_1 \to \mathcal{P}_2$ is *correct* if for each well formed process $P \in \mathcal{P}_1$, P is should testing equivalent to $[\![P]\!]$.

We will restrict our attention to good encodings, defined below.

Definition 16 (Conditions for good encodings). An encoding is *good* if it is compositional, correct and divergence reflecting.

The conditions for good encodings have been inspired by [23, 24, 25, 26, 27], and tailored to our case. In particular, condition 1 and condition 2 for compositional encoding are taken from the definition of uniform encoding in [24] (actually, condition 2 is less strict than the one in [24], thus generalizing our separation result). Condition 3 is referred to as compositionality in [23], and a slightly more general condition has been proposed in [26]. In those two works, the condition is required for each operator, while for our separation result we need it only for transaction scope. Anyway, our encoding will satisfy the condition for all the operators. Moving to the semantic conditions, divergence reflection is required for instance by [26], and can be seen as a particular instance of preservation of observables as defined in [23], where divergence is the chosen observable. Our notion of correctness can also be seen as a form of preservation of observables, where the chosen observable is the result of should tests. This is also related to the notions of operational correspondence and success sensitiveness from [26]. However the relation between the two notions is not immediate, since the framework in [26] allows to compare processes from unrelated languages, thus there is the need to encode also the test processes, while this is not the case for us.

As we already said, we will show that our encoding satisfies stricter conditions than the ones required for proving our separation results, thus making the result stronger. In particular, we will add to the notion of correctness based on should testing equivalence a stronger form of preservation of the observable behavior based on *weak bisimilarity*.

Weak bisimilarity for compensable processes extends weak early π-calculus bisimilarity [32] with features from higher-order bisimilarity [33], since compensation update is a (limited) form of higher-order communication.

Definition 17 (Weak bisimulation). A *weak bisimulation* is a symmetric binary relation $\mathcal{R}$ such that $P\mathcal{R}Q$ implies:

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

The extraction function $\mathsf{extr}(\bullet)$ in the last condition should be instantiated to $\mathsf{extr_a}(\bullet)$, $\mathsf{extr_p}(\bullet)$ or $\mathsf{extr_d}(\bullet)$ according to the semantics for transaction nesting used in the LTS. Closure under the extraction function is required for having a compositional semantics. The interested reader is referred to [34] for a discussion of the topic in the framework of the calculus webπ_∞.

Definition 18. *Weak bisimilarity* $\approx$ is the largest weak bisimulation.

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

Definition 19. Let $\mathcal{R}$ be an equivalence relation. An encoding $[\![\bullet]\!] : \mathcal{P}_1 \to \mathcal{P}_2$ is $\mathcal{R}$*-preserving* if for each well formed process $P \in \mathcal{P}_1$, $P\,\mathcal{R}\,[\![P]\!]$.

We call bisimilarity preserving an encoding which is $\approx$-preserving. The proposition below proves that a bisimilarity preserving encoding is correct.

Proposition 2. *Let P and Q be processes. If $P \approx Q$ then $P \simeq_{shd} Q$.*

Proof. The main point to prove is that weak bisimilarity is preserved by parallel composition. The proof is similar to the one for π-calculus [32]. Note that there is no need to use higher-order techniques, since labels for compensation update are just propagated by parallel composition[4].

Then, from $P \approx Q$ one can infer that, for each observer O, $P \mid O \approx Q \mid O$. The thesis follows easily. □

A similar result for must testing would require some additional condition, since, in general, weak bisimilarity does not imply must testing equivalence [28].

[4]Proving that weak bisimilarity is preserved by transaction scope would be far more difficult, since this requires higher-order techniques.

4. Parallel Recovery Can Be Implemented Using Static Recovery

In this section we compare the expressive power of parallel recovery and static recovery, considering the three possible semantics for nested transactions: aborting, preserving and discarding. We present an encoding from parallel recovery to static recovery, showing that static recovery is as expressive as parallel recovery. The same encoding can be used with all the semantics for nested transactions. The encoding respects the conditions of Definition 16. Even more, it preserves (for well formed processes) also weak bisimilarity. Thus the encoding is a good, bisimilarity preserving encoding.

The encoding associates to each transaction scope $t[P, Q]$ a fresh name r, to be used for activating compensation items installed by P. Compensation items to be installed are left in the body of the transaction scope, enclosed by a protected block (so that they are not deleted by abortion) and guarded by an input on r (so that they are not enabled before abortion). When the transaction scope aborts, an output on r, included in the static compensation Q, becomes enabled and can interact with the inputs on r guarding the stored compensation items, enabling them. Also, each of them regenerates the output on r so to enable further compensation items.

Definition 20 (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:

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

All the remaining operators are homomorphically mapped to themselves.

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

Example 2. We show here how the encoding can be applied to a simple example. Consider a transaction which books some hotel and then pays for it. In case of abort, the booking should be undone by sending a message $\overline{unbook}$, and the payment refunded by sending a message $\overline{refund}$. For simplicity, we do not consider the contents of the messages (e.g., data of the booking, credit card details for paying), adding them is however immediate. The transaction can be modeled using parallel recovery processes as

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

Figure 9 shows a sample execution[5]. The transition labels show how the transaction scope may interact with the environment. In line 1 the hotel is booked,

[5] To simplify the presentation we discard some garbage. This can be done using the notion of structural congruence in Definition 21.

$$
\begin{array}{rcl}
t[\overline{book}.\,\mathsf{inst}\lfloor\overline{unbook}\rfloor.\overline{pay}.\,\mathsf{inst}\lfloor\overline{refund}\rfloor, 0] & \xrightarrow{\overline{book}} & (1)\\
t[\mathsf{inst}\lfloor\overline{unbook}\rfloor.\overline{pay}.\,\mathsf{inst}\lfloor\overline{refund}\rfloor, 0] & \xrightarrow{\tau} & (2)\\
t[\overline{pay}.\,\mathsf{inst}\lfloor\overline{refund}\rfloor, 0 \mid \overline{unbook}] & \xrightarrow{\overline{pay}} & (3)\\
t[\mathsf{inst}\lfloor\overline{refund}\rfloor, 0 \mid \overline{unbook}] & \xrightarrow{\tau} & (4)\\
t[0, 0 \mid \overline{unbook} \mid \overline{refund}] & \xrightarrow{t} & (5)\\
0 \mid \langle 0 \mid \overline{unbook} \mid \overline{refund}\rangle & \xrightarrow{\overline{unbook}} & (6)\\
0 \mid \langle 0 \mid \overline{refund}\rangle & \xrightarrow{\overline{refund}} & (7)\\
0 \mid \langle 0\rangle & & (8)
\end{array}
$$

Figure 9: Parallel recovery process: sample execution.

and in line 2 the corresponding compensation item is installed. Similarly, in line 3 the hotel is paid, and in line 4 the corresponding compensation item is installed. In line 5 the transaction scope is aborted because of an external request. In lines 6-7 the two compensation items are executed.

The translation of the transaction above 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 10 shows a sample execution, corresponding to the one in Figure 9. In lines 1-2 the hotel is booked and paid, and compensation items are installed. In line 3 the transaction scope is aborted because of an external request. In lines 4-5 the static compensation $\overline{r}$ interacts with the stored compensation items enabling them. Finally, in lines 6-7 the two compensation items are executed. With respect to the original parallel recovery process, there are no explicit steps for compensation update, while some steps are needed to enable the compensation items before executing them.

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 [5] and asynchronous π-calculus [18], 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 or asynchronous π-calculus than to generalize an approach from CCS or asynchronous π-calculus to synchronous π-calculus.

The rest of this section is devoted to prove that $[\![\bullet]\!]_{p2s}$ is a good, bisimilarity preserving encoding (see Definitions 16 and 19). We describe in detail the case

$$(\nu\, r)\, t\, \left[\overline{book}.(\langle r.(\overline{unbook}|\overline{r})\rangle \mid \overline{pay}.\langle r.(\overline{refund}|\overline{r})\rangle), 0 \mid \overline{r}\right] \xrightarrow{\overline{book}} \quad (1)$$

$$(\nu\, r)\, t\, \left[\langle r.(\overline{unbook}|\overline{r})\rangle \mid \overline{pay}.\langle r.(\overline{refund}|\overline{r})\rangle, 0 \mid \overline{r}\right] \xrightarrow{\overline{pay}} \quad (2)$$

$$(\nu\, r)\, t\, \left[\langle r.(\overline{unbook}|\overline{r})\rangle \mid \langle r.(\overline{refund}|\overline{r})\rangle, 0 \mid \overline{r}\right] \xrightarrow{t} \quad (3)$$

$$(\nu\, r)\, \langle r.(\overline{unbook}|\overline{r})\rangle \mid \langle r.(\overline{refund}|\overline{r})\rangle \mid \langle \overline{r}\rangle \xrightarrow{\tau} \quad (4)$$

$$(\nu\, r)\, \langle r.(\overline{unbook}|\overline{r})\rangle \mid \langle (\overline{refund}|\overline{r})\rangle \xrightarrow{\tau} \quad (5)$$

$$(\nu\, r)\, \langle \overline{unbook}|\overline{r}\rangle \mid \langle \overline{refund}\rangle \xrightarrow{\overline{unbook}} \quad (6)$$

$$(\nu\, r)\, \langle \overline{r}\rangle \mid \langle \overline{refund}\rangle \xrightarrow{\overline{refund}} \quad (7)$$

$$(\nu\, r)\, \langle \overline{r}\rangle \mid \langle 0\rangle \quad (8)$$

Figure 10: Translation: sample execution.

of aborting semantics for transaction nesting, for dealing with preserving and discarding semantics minimum changes are required. In order to show that the encoding is correct we prove that it is bisimilarity preserving.

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. The difference can be spotted also with testing equivalence, using e.g. context $\overline{r}.\overline{\surd}$.

Alternatively, one may require that actions on fresh names introduced by the encoding, such as r here, are not observed by the behavioral equivalence. We have chosen the approach based on well formedness since it matches the intuition that compensation update is not meaningful outside a transaction scope.

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 will be shown by Lemma 7 and Lemma 9. 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 relationship holds only up to some transformations deleting the garbage produced by the encoding. We formalize these transformations using a *structural congruence* and an *auxiliary reduction relation*.

Definition 21 (Structural congruence). Structural congruence $\equiv$ on compensable processes is the minimum congruence closed under the rules in Fig. 11.

The first three rows in the definition of structural congruence contain standard rules from π-calculus: associativity, commutativity and neutral element

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid Q \equiv Q \mid P \qquad \mathbf{0} \mid P \equiv P$$

$$(\nu\, x)\, \mathbf{0} \equiv \mathbf{0} \qquad (\nu\, x)\ (\nu\, y)\ P \equiv (\nu\, y)\ (\nu\, x)\, P$$

$$P \mid (\nu\, x)\ Q \quad \equiv \quad (\nu\, x)\ (P \mid Q) \qquad \text{if } x \notin \mathsf{fn}(P)$$

$$t\,[(\nu\, x)\ P, Q] \quad \equiv \quad (\nu\, x)\ t\,[P, Q] \qquad \text{if } t \neq x,\ x \notin \mathsf{fn}(Q)$$

$$\langle (\nu\, x)\, P \rangle \equiv (\nu\, x)\, \langle P \rangle \qquad \langle\langle P \rangle\rangle \equiv \langle P \rangle$$

$$\langle P \mid Q \rangle \equiv \langle P \rangle \mid \langle Q \rangle \qquad \langle \mathbf{0} \rangle \equiv \mathbf{0} \qquad (\nu\, x)\, \overline{x} \equiv \mathbf{0}$$

Figure 11: Structural congruence relation.

for parallel composition, garbage collection of useless restrictions, swapping of restrictions and scope extrusion. The two following rules define scope extrusion for transaction scope and protected block. Then there are three rules defining basic properties of protected blocks. Finally, we consider the simple garbage collection rule $(\nu\, x)\, \overline{x} \equiv \mathbf{0}$, since it simplifies our proofs.

The following lemma shows that structural congruent processes have the same transitions.

Lemma 3. *Let P, Q be processes such that $P \equiv Q$. Then $P \xrightarrow{\alpha} P'$ iff $Q \xrightarrow{\alpha} Q'$*[6] *with $P' \equiv Q'$.*

Proof. By induction on the derivation of $P \equiv Q$, with a case analysis on the used axiom. □

The auxiliary reduction relation below catches the fact that stored compensation items have to be activated before becoming enabled.

Definition 22 (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, for each $i \in \{1, \ldots, n\}$, $r \notin \mathsf{fn}(Q_i)$.

The auxiliary reduction relation is included in $\Longrightarrow$.

Lemma 4. *Let P be a process. If $P \mapsto P'$ then $P \Longrightarrow P'$.*

Proof. Trivial. □

[6] The two labels denoted by α may actually differ on the order of extruded names, but, as discussed before, this difference can be disregarded.

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. However, differently from what happens for translations, if P evolves to P' then each possible translation of P evolves to a possible translation of P', and this allows to build coinductive proofs. Roughly, 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 23 (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 $\tilde{P}' \in \{\!|P|\!\}_{p2s}$.

Possible translations are well behaved w.r.t. name substitutions.

Lemma 5. *Let σ be a name substitution such that r (the fixed fresh name used in the possible translation) does not occur in σ. Then $\tilde{P} \in \{\!|P|\!\}_{p2s}$ iff $\tilde{P}\sigma \in \{\!|P\sigma|\!\}_{p2s}$.*

Proof. By induction on the derivation of $\tilde{P} \in \{\!|P|\!\}_{p2s}$. □

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

Lemma 6. *Let P be a parallel recovery process and $\tilde{P} \in \{\!|P|\!\}_{p2s}$ one of its possible translations. Then $\mathsf{extr_a}(\tilde{P}) \in \{\!|\mathsf{extr_a}(P)|\!\}_{p2s}$.*

Proof. The proof is composed of two parts: the first one is by structural induction on P, the second one considers closures under the structural congruence and the auxiliary reduction relation. For the first part the only non trivial case is $P = t[P', Q]$. In this case $\tilde{P} = (\nu\, r)\, t\left[\tilde{P}' \mid \prod_{i\in\{1\ldots,n\}} \langle r.(\tilde{Q}_i \mid \overline{r})\rangle, \tilde{Q}_0 \mid \overline{r}\right]$ for some

$\tilde{P}' \in \{\!|P'|\!\}_{p2s}$ and $\tilde{Q}_i \in \{\!|Q_i|\!\}_{p2s}$ for each $i \in \{0\ldots,n\}$ with $Q \equiv \prod_{i\in\{0\ldots,n\}} Q_i$. Also, $\mathsf{extr_a}(t[P',Q]) = \mathsf{extr_a}(P') \mid \langle Q\rangle$.

We have $\mathsf{extr_a}(\tilde{P}) = (\nu\, r)\, \mathsf{extr_a}(\tilde{P}' \mid \prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{Q}_i \mid \overline{r})\rangle) | \langle \tilde{Q}_0 | \overline{r}\rangle \equiv \mathsf{extr_a}(\tilde{P}') | (\nu\, r)\, (\prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{Q}_i \mid \overline{r})\rangle \mid \langle \overline{r}\rangle) \mid \langle \tilde{Q}_0\rangle \mapsto \mathsf{extr_a}(\tilde{P}') \mid (\nu\, r)\, (\prod_{i\in\{0,\ldots,n\}} \langle \tilde{Q}_i\rangle \mid \langle \overline{r}\rangle)$. Since r does not occur in $\tilde{Q}_i$ we have $\mathsf{extr_a}(\tilde{P}') \mid (\nu\, r)\, (\prod_{i\in\{0,\ldots,n\}} \langle \tilde{Q}_i\rangle \mid \langle \overline{r}\rangle) \equiv \mathsf{extr_a}(\tilde{P}') \mid \prod_{i\in\{0,\ldots,n\}} \langle \tilde{Q}_i\rangle$. This concludes the first part of the proof since by inductive hypothesis $\mathsf{extr_a}(\tilde{P}') \in \{\!|\mathsf{extr_a}(P')|\!\}_{p2s}$ and $\prod_{i\in\{0,\ldots,n\}} \langle \tilde{Q}_i\rangle \equiv \langle \tilde{Q}\rangle$ for some $\tilde{Q} \in \{\!|Q|\!\}_{p2s}$.

For the second part, the case of closure under structural congruence is trivial. Closure under the auxiliary reduction relation is trivial too, since $\mathsf{extr_a}(\bullet)$ is the identity on both the sides of the rule. □

Lemma 7 and Lemma 9 below relate the behavior of a process to the one of its possible translations. Remember that we may write a compensation update $\lambda X.Q \mid X$ simply as Q.

Lemma 7. *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. *$\alpha \neq (\vec{z})Q$ and $\tilde{P} \overset{\alpha}{\Longrightarrow} \tilde{P}'$ with $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$;*
2. *$\alpha = (\vec{z})Q$ and $\tilde{P} \Longrightarrow \equiv (\nu\, \vec{z})\, (\tilde{P}' \mid \langle r.(\tilde{Q} \mid \overline{r})\rangle)$ where $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$ and $\tilde{Q} \in \{\!|Q|\!\}_{p2s}$.*

Proof. The proof is by structural induction on P, using a case analysis on the last applied rule. In the analysis of the possible translations we have no need to consider the closure under the structural congruence and the auxiliary reduction relation since they can be taken into account using Lemma 3 and Lemma 4.

- Case $\mathbf{0}$) Trivial.
- Case $a(\vec{x}).R$) The only applicable rule is rule (P-In), allowing $a(\vec{x}).R \xrightarrow{a(\vec{v})} R\{\vec{v}/\vec{x}\}$. From the definition of possible translations $\tilde{P}$ has the form $a(\vec{x}).\tilde{R}$ where $\tilde{R} \in \{\!|R|\!\}_{p2s}$. Applying rule (P-In) to the possible translation $\tilde{P}$ we get $a(\vec{x}).\tilde{R} \xrightarrow{a(\vec{v})} \tilde{R}\{\vec{v}/\vec{x}\}$ as desired (using Lemma 5).
- Case $\overline{a}\langle\vec{v}\rangle.R$) Simpler than the previous one.
- Case $\mathsf{inst}\lfloor Q\rfloor.R$) The only applicable rule is (L-Inst), which allows to derive $\mathsf{inst}\lfloor Q\rfloor.R \xrightarrow{Q} R$. From the definition of possible translations $\tilde{P}$ has the form $\tilde{R} \mid \langle r.(\tilde{Q} \mid \overline{r})\rangle$ where $\tilde{R} \in \{\!|R|\!\}_{p2s}$ and $\tilde{Q} \in \{\!|Q|\!\}_{p2s}$. This is already the desired form.
- Case $\sum_{i\in I} \pi_i.P_i$) The only applicable rule is (L-Choice), which allows to derive $\sum_{i\in I} \pi_i.P_i \xrightarrow{\alpha} P'_j$ if $\pi_j.P_j \xrightarrow{\alpha} P'_j$. From the definition of possible translations $\tilde{P}$ has the form $\sum_{i\in I} \pi_i.\tilde{P}_i$ where $\tilde{P}_i \in \{\!|P_i|\!\}_{p2s}$. Note that α

cannot have the form $(\vec{z})Q$. By inductive hypothesis $\pi_j.\tilde{P}_j \stackrel{\alpha}{\Longrightarrow} \tilde{P}'_j$ with $\tilde{P}'_j \in \{\!|P'_j|\!\}_{p2s}$. Applying rule (L-Choice) to the possible translation we get $\sum_{i\in I} \pi_i.\tilde{P}_i \stackrel{\alpha}{\Longrightarrow} \tilde{P}'_j$ as desired.

- Case $!\pi_i.R$) Similar to the case above.

- Case $R \mid S$) We have a case analysis on the applied rule.

  Let us consider rule (L-Close). From the premises we know $R \xrightarrow{x(\vec{v})} R'$ and $S \xrightarrow{(\vec{z})\overline{x}\langle\vec{v}\rangle} S'$. From the definition of possible translations, $\tilde{P}$ has the form $\tilde{R} \mid \tilde{S}$ where $\tilde{R} \in \{\!|R|\!\}_{p2s}$ and $\tilde{S} \in \{\!|S|\!\}_{p2s}$. By inductive hypothesis $R \stackrel{x(\vec{v})}{\Longrightarrow} R'$ and $S \stackrel{(\vec{z})\overline{x}\langle\vec{v}\rangle}{\Longrightarrow} S'$. Steps with label τ can be propagated to $\tilde{R} \mid \tilde{S}$ using rule (L-Par). Then the input and the output actions can be combined into a τ step using rule (L-Close). Steps with label τ after the synchronization can again be propagated using rule (L-Par), followed by rule (L-Res) to take into account the restrictions possibly introduced by rule (L-Close).

  If the applied rule is (L-Par) then the thesis follows again by inductive hypothesis. If the label has the form $(\vec{z})Q$ then structural congruence can be used to pull out restrictions corresponding to names in $\vec{z}$.

- Case $(\nu\, x)\, R$) We have again a case analysis on the used rule. If the used rule is (L-Res) then inductive hypothesis can be applied, using structural congruence to pull out restrictions corresponding to names in $\vec{z}$ if the label is $(\vec{z})Q$. The case of rule (L-Open) follows by inductive hypothesis, using rule (L-Res) to propagate τ actions.

  Let us consider rule (L-Open-inst). We know that $R \xrightarrow{(\vec{w})Q} R'$. From inductive hypothesis given $\tilde{R} \in \{\!|R|\!\}_{p2s}$ we have $\tilde{R} \Longrightarrow (\nu\, \vec{w})\, (\tilde{R}' | \langle r.(\tilde{Q}|\overline{r})\rangle)$ where $\tilde{R}' \in \{\!|R'|\!\}_{p2s}$ and $\tilde{Q} \in \{\!|Q|\!\}_{p2s}$. We can apply rule (L-Res) to all transitions to get $(\nu\, x)\, \tilde{R} \Longrightarrow (\nu\, x)\, (\nu\, \vec{w})\, (\tilde{R}' \mid \langle r.(\tilde{Q} \mid \overline{r})\rangle)$ as desired (up to structural congruence).

- Case $t[R, S]$) Consider a decomposition $S \equiv \prod_{i\in\{0,\ldots,n\}} S_i$ and a corresponding possible translation
  $(\nu\, r)\, t\left[\tilde{R} \mid \prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r})\rangle, \tilde{S}_0 \mid \overline{r}\right]$
  of $t[R, S]$, where $\tilde{R} \in \{\!|R|\!\}_{p2s}$ and $\tilde{S}_i \in \{\!|S_i|\!\}_{p2s}$ for each $i \in \{0, \ldots, n\}$.

  Let us consider rule (L-Recover-out). We know that $t[R, S] \xrightarrow{t} \mathsf{extr_a}(R)|\langle S\rangle$ and we have to show a corresponding transition from the possible translation. We can derive
  $(\nu\, r)\, t\left[\tilde{R} \mid \prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r})\rangle, \tilde{S}_0 \mid \overline{r}\right] \xrightarrow{t}$
  $\quad (\nu\, r)\, \mathsf{extr_a}(\tilde{R} \mid \prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r})\rangle) \mid \langle \tilde{S}_0 \mid \overline{r}\rangle$
  From the definition of function $\mathsf{extr_a}(\bullet)$ this is equal to:
  $(\nu\, r)\, \mathsf{extr_a}(\tilde{R}) \mid \prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r})\rangle \mid \langle \tilde{S}_0 \mid \overline{r}\rangle$

Using structural congruence and the auxiliary reduction relation this reduces to:
$(\nu\, r)\, \mathsf{extr_a}(\tilde{R}) \mid \prod_{i\in\{0,\ldots,n\}} \langle \tilde{S}_i \rangle \mid \langle \overline{r} \rangle \equiv \mathsf{extr_a}(\tilde{R}) \mid \langle \tilde{S} \rangle$
The thesis follows from Lemma 6 and from the definition of possible translation.

Let us consider rule (L-Recover-in). We have $t[R, S] \xrightarrow{\tau} \mathsf{extr_a}(R') \mid \langle S \rangle$ and we have to show a corresponding transition from the possible translation. We know that $R \xrightarrow{\overline{t}} R'$. By inductive hypothesis $\tilde{R} \overset{\overline{t}}{\Longrightarrow} \tilde{R}'$ where $\tilde{R}' \in \{\!|R'|\!\}_{p2s}$. Thus we have $(\nu\, r)\, t \left[\tilde{R} \mid \prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r}) \rangle, \tilde{S}_0 \mid \overline{r}\right] \Longrightarrow$ $(\nu\, r)\, \mathsf{extr_a}(\tilde{R}' \mid \prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r}) \rangle) \mid \langle \tilde{S}_0 \mid \overline{r} \rangle$ where we applied rules (L-Par), (L-Scope-out) and (L-Res) for τ actions before action $\overline{t}$, rules (L-Par), (L-Recover-in) and (L-Res) for $\overline{t}$, and rules (L-Par) and (L-Res) for τ actions after action $\overline{t}$. From the definition of function $\mathsf{extr_a}(\bullet)$ this is equal to

$$
\begin{aligned}
&(\nu\, r)\, \mathsf{extr_a}(\tilde{R}') \mid \textstyle\prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r}) \rangle \mid \langle \tilde{S}_0 \mid \overline{r} \rangle \equiv \\
&\quad \mathsf{extr_a}(\tilde{R}') \mid (\nu\, r)\, (\textstyle\prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r}) \rangle \mid \langle \overline{r} \rangle) \mid \langle \tilde{S}_0 \rangle \mapsto \\
&\quad \mathsf{extr_a}(\tilde{R}') \mid (\nu\, r)\, (\textstyle\prod_{i\in\{0,\ldots,n\}} \langle \tilde{S}_i \rangle \mid \langle \overline{r} \rangle) \equiv \\
&\quad \mathsf{extr_a}(\tilde{R}') \mid \langle \tilde{S} \rangle \mid (\nu\, r)\, \langle \overline{r} \rangle \equiv \mathsf{extr_a}(\tilde{R}') \mid \langle \tilde{S} \rangle
\end{aligned}
$$

The thesis follows from Lemma 6 and from the definition of possible translation.

Let us consider the case of rule (L-Scope-close). We have $t[R, S] \xrightarrow{\tau} (\nu\, \vec{z})\, t[R', (Q \mid X)\{^S/_X\}] = (\nu\, \vec{z})\, t[R', Q \mid S]$ and we have to show a corresponding transition from the possible translation. We know that $R \xrightarrow{(\vec{z})Q} R'$. By inductive hypothesis $\tilde{R} \Longrightarrow (\nu\, \vec{z})\, \tilde{R}' \mid \langle r.(\tilde{Q} \mid \overline{r}) \rangle$ with $\tilde{R}' \in \{\!|R'|\!\}_{p2s}$ and $\tilde{Q} \in \{\!|Q|\!\}_{p2s}$. Thus we can derive

$$
\begin{aligned}
&(\nu\, r)\, t \left[\tilde{R} \mid \textstyle\prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r}) \rangle, \tilde{S}_0 \mid \overline{r}\right] \Longrightarrow \\
&\quad (\nu\, r)\, t \left[(\nu\, \vec{z})\, (\tilde{R}' \mid \langle r.(\tilde{Q} \mid \overline{r}) \rangle) \mid \textstyle\prod_{i\in\{1,\ldots,n\}} \langle r.(\tilde{S}_i \mid \overline{r}) \rangle, \tilde{S}_0 \mid \overline{r}\right]
\end{aligned}
$$

where we applied rules (L-Par), (L-res) and (L-Scope-out).

By choosing $\tilde{S}_{n+1} = \tilde{Q}$ the process above can be written as
$(\nu\, r)\, t \left[(\nu\, \vec{z})\, (\tilde{R}' \mid \prod_{i\in\{1,\ldots,n+1\}} \langle r.(\tilde{S}_i \mid \overline{r}) \rangle), \tilde{S}_0 \mid \overline{r}\right]$.

This is structural congruent to
$(\nu\, r)\, (\nu\, \vec{z})\, t \left[\tilde{R}' \mid \prod_{i\in\{1,\ldots,n+1\}} \langle r.(\tilde{S}_i \mid \overline{r}) \rangle, \tilde{S}_0 \mid \overline{r}\right]$
as desired. Note, in fact, that $\prod_{i\in\{0,\ldots,n+1\}} \tilde{S}_i$ is a possible decomposition of $\tilde{Q} \mid \tilde{S}$.

The case of rule (L-Scope-out) is simpler than the previous one.

- Case $\langle R \rangle$) By inductive hypothesis. □

We have discussed how possible translations can mimic the behavior of the original process, in order to discuss the other direction of the relation we need a result about the auxiliary reduction relation $\mapsto$.

Lemma 8. *Let P be a process. If $P \xrightarrow{\alpha} P'$ and $P \mapsto Q$ then*

- *either $Q \xrightarrow{\alpha} Q'$ and $P' \mapsto Q'$;*
- *or $\alpha = \tau$ and $P' \equiv Q$;*
- *or $\alpha = \tau$ and $P' \mapsto Q$.*

Proof. We have $P = C[R]$ where R is the left-hand side of the axiom defining $\mapsto$, i.e. $C[R] \mapsto C[R']$. Note that it is not possible for the transition with label α to involve both process R and context $C[\bullet]$, since R has no possible interaction with external processes. Thus the transition with label α may be either of the form $C[R] \xrightarrow{\alpha} C'[R]$ for some C' (even kill preserves R, since R is composed by protected blocks), or of the form $C[R] \xrightarrow{\alpha} C[R']$ for some R'.

In the first case $C'[R] \mapsto C'[R']$ and $C[R'] \xrightarrow{\alpha} C'[R']$ as desired (first item of the statement).

In the second case the only possibiliy is that α is a τ step for enabling one of the compensation items. If it is the last remaining compensation item then $P' \equiv Q$ (second item in the statement). Otherwise also $P' \mapsto Q$ (third item in the statement) by enabling the remaining compensation items. □

Lemma 9. *Let P be a parallel recovery process such that* $\mathsf{noComp}(P)$ *and let $\tilde{P} \in \{\!|P|\!\}_{p2s}$ be 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 $\tilde{P} \in \{\!|P|\!\}_{p2s}$. For the first three cases, the case analysis is similar to the one in Lemma 7. We consider as an example just the case where $P = t[R, Q]$. In this case $\tilde{P} = (\nu\, r)\, t\left[\tilde{R} \mid \prod_{i\in\{1\ldots,n\}} \langle r.(\tilde{Q}_i \mid \overline{r})\rangle, \tilde{Q}_0 \mid \overline{r}\right]$. Let us consider the transition

$$(\nu\, r)\, t\left[\tilde{R} \mid \prod_{i\in\{1\ldots,n\}} \langle r.(\tilde{Q}_i \mid \overline{r})\rangle, \tilde{Q}_0 \mid \overline{r}\right] \xrightarrow{\alpha}$$
$$(\nu\, r)\, t\left[\tilde{R}' \mid \prod_{i\in\{1\ldots,n\}} \langle r.(\tilde{Q}_i \mid \overline{r})\rangle, \tilde{Q}_0 \mid \overline{r}\right]$$

This transition is derived from $\tilde{R} \xrightarrow{\alpha} \tilde{R}'$ by applying rules (L-Par), (L-Scope-out) and (L-Res). By inductive hypothesis $R \xRightarrow{\alpha} R'$ with $\tilde{R}' \in \{\!|R'|\!\}_{p2s}$. Then $t[R, Q] \xRightarrow{\alpha} t[R', Q]$. The thesis follows from the definition of possible translation. The other cases are similar.

The case for structural congruence follows from Lemma 3.

Let us consider the case for the auxiliary reduction relation. There exists $\tilde{Q}$ such that $\tilde{P} \mapsto \tilde{Q}$ and $\tilde{Q} \in \{\!|P|\!\}_{p2s}$. We know that $\tilde{P} \xrightarrow{\alpha} \tilde{P}'$. We can now apply Lemma 8. Let us consider the possible cases.

If we have that $\tilde{Q} \xrightarrow{\alpha} \tilde{Q}'$ and $\tilde{P}' \mapsto \tilde{Q}'$ then by applying inductive hypothesis to $\tilde{Q} \xrightarrow{\alpha} \tilde{Q}'$ we have $P \xRightarrow{\alpha} P'$ with $\tilde{Q}' \in \{\!|P'|\!\}_{p2s}$. Thanks to the definition of auxiliary reduction relation also $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$ as desired.

If $\alpha = \tau$ and $\tilde{P}' \equiv \tilde{Q}$ then also $\tilde{P}' \in \{\!|P|\!\}_{p2s}$ and the empty computation from P satisfies the thesis.

If $\alpha = \tau$ and $\tilde{P}' \mapsto \tilde{Q}$ then also $\tilde{P}' \in \{\!|P|\!\}_{p2s}$ and again the empty computation from P satisfies the thesis. □

The following lemmas prove properties of well formed processes and processes with well formed compensations necessary for proving that the encoding $[\![\bullet]\!]_{p2s}$ is bisimilarity preserving.

Lemma 10. *Let P be a well formed process. Then P has well formed compensations.*

PROOF. By induction on the definition of well formed process. □

Lemma 11. *Let P be a process with well formed compensations. If $P \xrightarrow{(\vec{z})Q} P'$ then Q is well formed.*

PROOF. By induction on the definition of process with well formed compensations. □

Lemma 12. *Let P be a process with well formed compensations. Then $\mathsf{extr_a}(P)$ is well formed.*

PROOF. By induction on the definition of process with well formed compensations. □

Lemma 13. *Let P be a well formed process. If $P \xrightarrow{\alpha} P'$ then α is not of the form $(\vec{z})\lambda X.R$.*

PROOF. By induction on the definition of well formed process. □

Lemma 14. *Let P be a process. If $P \xrightarrow{\alpha} P'$ then:*

- *if P is well formed then P' is well formed;*
- *if P has well formed compensations then P' has well formed compensations.*

PROOF. By structural induction on P. The most difficult case is $P = t[R, Q]$. Let us consider the applicable rules.

For rule (L-RECOVER-OUT) $t[R, Q] \xrightarrow{t} \mathsf{extr_a}(R) \,|\, \langle Q \rangle$. Assume P well formed. Then Q is well formed and R has well formed compensations. Thanks to Lemma 12 $\mathsf{extr_a}(R)$ is well formed. Then also $\mathsf{extr_a}(R) \mid \langle Q \rangle$ is well formed as required. Now assume P has well formed compensations. Then R has well formed compensations and Q is well formed. Thanks to Lemma 12 $\mathsf{extr_a}(R)$ is well formed and thanks to Lemma 10 it has also well formed compensations. Since Q is well formed then $\langle Q \rangle$ has well formed compensations. The thesis follows. The case of rule (L-RECOVER-IN) is similar, but it also exploits inductive hypothesis.

Let us consider the case of rule (L-SCOPE-CLOSE). We have $t[R, Q] \xrightarrow{\tau} (\nu\, \vec{z})\, t[R', S\{^{Q}/_{X}\}]$ with hypothesis $R \xrightarrow{(\vec{z})\lambda X.S} R'$. Assume $t[R, Q]$ well formed. Then R has well formed compensations and Q is well formed. Since $R \xrightarrow{(\vec{z})\lambda X.S}$

R' then by inductive hypothesis also R' has well formed compensations. Thanks to Lemma 11 also S is well formed. Then $S\{^Q/_X\}$ is well formed too. The thesis follows from the definition of well formedness. Assume now that $t[R, Q]$ has well formed compensations. Then R has well formed compensations and Q is well formed. By inductive hypothesis also R' has well formed compensations. Also, S is well formed thanks to Lemma 11. Then $S\{^Q/_X\}$ is well formed. The thesis follows from the definition of process with well formed compensations. The case of rule (L-Scope-out) is simpler. □

Lemma 15. *Let P be a well formed process and $\tilde{P} \in \{\!|P|\!\}_{p2s}$. If $\tilde{P} \xrightarrow{\alpha} \tilde{P}'$ then $\alpha \neq r$.*

Proof. The proof is by induction on the derivation of $\mathsf{wf}(P)$. □

Next lemma proves that a process P can always evolve to some process P' such that $\mathsf{noComp}(P')$ by performing all the enabled compensation updates.

Lemma 16. *Let P be a parallel recovery process. Then $P \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} P'$ with $\mathsf{noComp}(P')$ and with each α_i of the form τ or $(\vec{w})Q$. Also, if the label is τ, then the transition has been derived using axiom (*L-Inst*) as only premise.*

Proof. By structural induction on P. □

The following lemma proves that compensation update corresponds to an empty computation in the translation.

Lemma 17. *Let P be a parallel recovery process. If $P \xrightarrow{\tau} P'$ and the transition has been derived using axiom (*L-Inst*) as only premise then $\{\!|P|\!\}_{p2s} = \{\!|P'|\!\}_{p2s}$.*

Proof. By induction on the derivation of $P \xrightarrow{\tau} P'$, using a stronger inductive hypothesis, requiring also that if $P \xrightarrow{(\vec{z})Q} P'$ then $\tilde{P} \in \{\!|P|\!\}_{p2s}$ iff $\tilde{P} \equiv (\nu\, \vec{z})\, (\tilde{P}' \mid \langle r.(\tilde{Q} \mid \overline{r})\rangle)$ with $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$ and $\tilde{Q} \in \{\!|Q|\!\}_{p2s}$. □

We can finally prove our main result, showing that $[\![\bullet]\!]_{p2s}$ is bisimilarity preserving.

Theorem 1. *Let P be a well formed parallel recovery 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 condition on extraction function is satisfied thanks to Lemma 6. The extracted process is well formed thanks to Lemma 10 and Lemma 12. Assume a transition $P \xrightarrow{\alpha} P'$. Thanks to Lemma 13, α has not the form $(\tilde{z})\lambda X.R$. Thus thanks to Lemma 7 $\tilde{P} \stackrel{\alpha}{\Longrightarrow} \tilde{P}'$ with $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$. Thanks to Lemma 14 also P' is well formed, thus $(P', \tilde{P}')$ is in $\mathcal{R}$.

Now, assume a transition $\tilde{P} \xrightarrow{\alpha} \tilde{P}'$. Thanks to Lemma 16 $P \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} P_n$ with $\mathsf{noComp}(P_n)$ and with each α_i of the form τ or $(\vec{w})Q$. The proof is by

induction on n. For the base case $n = 0$ from Lemma 9 (since $\alpha \neq r$ thanks to Lemma 15) $P \stackrel{\alpha}{\Longrightarrow} P'$ with $\tilde{P}' \in \{\!|P'|\!\}_{p2s}$. Since P' is well formed thanks to Lemma 14, the thesis follows. Let us consider the inductive case. We know that $P \stackrel{\alpha_1}{\longrightarrow} P_1$. However, thanks to Lemma 13, α_1 should be τ and should be derived using axiom (L-Inst) as only premise (from Lemma 16). Thanks to Lemma 17 $\tilde{P} \in \{\!|P_1|\!\}_{p2s}$. From inductive hypothesis $P_1 \stackrel{\alpha}{\Longrightarrow} P_1'$ with $\tilde{P}' \in \{\!|P_1'|\!\}_{p2s}$. Thus also $P \stackrel{\alpha}{\Longrightarrow} P_1'$. Thanks to Lemma 14 also P_1' is well formed. The thesis follows. □

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

Proof. It is easy to see that $[\![\bullet]\!]_{p2s}$ is compositional. Thanks to Theorem 1 and Proposition 2 it is also correct. Finally, it is divergence reflecting since the only additional transitions are the consumptions of the outputs $\overline{r}$, which are a finite number in the starting process, and which are created in a finite number at each step. Also, they are never created by the consumption steps themselves. □

The results in this section have been proved for the aborting semantics for transaction nesting, however the same proofs can be used for the preserving and the discarding semantics.

This result proves that parallel recovery can be implemented on top of static recovery, thus there is no need to implement it in the core language, but it can be provided as a derived construct.

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 results 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 compensation items 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 performing a sequence of inputs on names a, b, t, b' but no computation with a sequence of inputs on names 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 2 exploits similar arguments. The proof is based on the fact that the order of installation of compensation items 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 24 (Enabling contexts). *Enabling contexts* $E[\bullet_1]$ are unary contexts generated by:
$E[\bullet_1] ::= \bullet_1 \;\mid\; P|E[\bullet_1] \;\mid\; E[\bullet_1]|P \;\mid\; (\nu\, x)\, E[\bullet_1] \;\mid\; t[E[\bullet_1], Q] \;\mid\; \langle E[\bullet_1]\rangle$

In the proof of the separation result, we will use as auxiliary equivalence *may testing* [29].

Definition 25 (May 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 may O iff $P|O \overset{\surd}{\Longrightarrow}$. Two processes P and Q are *may testing equivalent*, written $P \simeq_{may} Q$, if, for each observer O, P may O iff Q may O.

The following proposition, taken from [35], shows that should testing implies may testing.

Proposition 3. *Let P and Q be processes. If $P \simeq_{shd} Q$ then $P \simeq_{may} Q$.*

Proof. The proof in [35, Proposition 11] can be applied to compensable processes. □

Next definition introduces two measures on the complexity of processes.

Definition 26. 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 (two transaction scopes are nested if one is inside the body of the other).

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

Lemma 18. *Let P be a process. If $P \xrightarrow{\alpha} P'$ then $\mathrm{mcd}(P') \leq \mathrm{mcd}(P)$ and $\mathrm{mtd}(P') \leq \mathrm{mtd}(P)$.*

Proof. The proof is by induction on the derivation of $P \xrightarrow{\alpha} P'$. □

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

Lemma 19. *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 $P = E[Q_1|Q_2]$ 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 detail here the most interesting cases, the others are trivial or follow by inductive hypothesis.

$P = \sum_{i \in I} \pi_i.P_i$) Let m be the size of I. This is also the maximum number n of enabled actions in P. Since by hypothesis $n > c + t \geq m + t \geq m = n$ we have a contradiction. Thus this case cannot happen.

$P = t[R, S]$) There are two cases: either $t = a_j$ for some j or not. In the first case, R must be able to perform all the other actions. Note that $\mathrm{mtd}(R) = \mathrm{mtd}(P) - 1$. Thus we can apply the inductive hypothesis to R and the remaining actions. The other case is simpler.

$P = R \mid S$) If there are j and k such that $R \xrightarrow{a_j} R'$ and $S \xrightarrow{a_k} S'$ then $E[\bullet] = \bullet$ satisfies the thesis. Otherwise, either R or S must be able to perform all the actions, and the thesis follows by inductive hypothesis. □

We can finally prove the desired separation result.

Theorem 2. *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 $P_i = a_i.\mathsf{inst}\lfloor \lambda Y_i.b_i.\mathbf{0}\rfloor.\mathbf{0}$. 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 translation $[\![P]\!]_{g2s}$ should be of the form $C_t[\prod_{i\in\{1,\ldots,n\}}[\![P_i]\!]_{g2s}, [\![\mathbf{0}]\!]_{g2s}]$. We will denote it as $\tilde{P}$. P is well formed, thus $P \simeq_{shd} \tilde{P}$ and, from Proposition 3, also $P \simeq_{may} \tilde{P}$. 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 may 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 τ action should compromise the possibility of performing a_j for each j, thanks to the definition of should testing equivalence.

We show now by contradiction that $\tilde{P} \Longrightarrow Q$ for some Q such that $Q \xrightarrow{a_i} Q'_i$ for each $i \in \{1, \ldots, n\}$. We assume that such a Q does not exist and build an infinite computation composed by transitions with label τ, contradicting divergence reflection. Since we assume Q does not exist, in particular, for some a_i there is no transition $\tilde{P} \xrightarrow{a_i}$. Thus since $\tilde{P} \stackrel{a_i}{\Longrightarrow}$ we have $\tilde{P} \xrightarrow{\tau}^{+} Q_1 \xrightarrow{a_i}$ where $\xrightarrow{\tau}^{+}$ denotes a non empty sequence of transitions with label τ. 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 of the form $Q_1 \xrightarrow{a_j}$. Thus we can further extend the computation. By iterating the procedure we get an infinite sequence of transitions with label τ. 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 18 the same relation holds also for Q. Thus we can apply Lemma 19 to prove that $Q = E[Q_1|Q_2]$ with $Q_1 \xrightarrow{a_j} Q'_1$ and $Q_2 \xrightarrow{a_k} Q'_2$ for some enabling context $E[\bullet]$. We have $E[Q_1|Q_2] \xrightarrow{a_j} E[Q'_1|Q_2] \xrightarrow{a_k} E[Q'_1|Q'_2]$ and $E[Q_1|Q_2] \xrightarrow{a_k} E[Q_1|Q'_2] \xrightarrow{a_j} E[Q'_1|Q'_2]$. 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 $\tilde{P}$ may 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 aborting, preserving and discarding semantics for transaction nesting.

Remark 3. We have presented this separation result in the framework of synchronous π-calculus. The same result however can be proved for CCS [5], extended with the primitives for transactions and compensations. In fact the used processes and observers are all CCS processes.

We can adapt the proof above to the case of asynchronous compensable processes. Remember that in asynchronous compensable processes there is no continuation after the output prefix. In particular, there is no compensation update after outputs.

The main difficulty is that only asynchronous contexts can be used, and this makes much more difficult to check the order in which input actions are executed. First, we show that there is no good encoding preserving *weak asynchronous bisimilarity* [36], which is defined in our context as follows.

Definition 27 (Weak asynchronous bisimulation). A *weak asynchronous bisimulation* is a symmetric binary relation $\mathcal{R}$ such that $P\mathcal{R}Q$ implies:

- if $P \xrightarrow{\alpha} P'$ with $\alpha \neq (\vec{z})\lambda X.R$, $\alpha \neq a(\vec{v})$ and $\mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$, then there is Q' such that $Q \stackrel{\alpha}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$;
- if $P \xrightarrow{(\vec{z})\lambda X.R} P'$ and $\{\vec{z}\} \cap \mathsf{fn}(Q) = \emptyset$ then there are S, Q' such that $Q \stackrel{(\vec{z})\lambda X.S}{\Longrightarrow} Q'$, $P'\mathcal{R}Q'$ and $R\{^T/_X\}\mathcal{R}S\{^T/_X\}$ for all processes T with no free variables;
- if $P \xrightarrow{a(\vec{v})} P'$, then
  1. either there is Q' such that $Q \stackrel{a(\vec{v})}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$
  2. or there is Q' such that $Q \stackrel{\tau}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q' \mid \overline{a}\langle\vec{v}\rangle$.
- $\mathsf{extr}(P)\mathcal{R}\,\mathsf{extr}(Q)$.

The extraction function $\mathsf{extr}(\bullet)$ in the last condition should be instantiated to $\mathsf{extr_a}(\bullet)$, $\mathsf{extr_p}(\bullet)$ or $\mathsf{extr_d}(\bullet)$ according to the semantics for transaction nesting used in the LTS.

Definition 28. *Weak asynchronous bisimilarity* $\approx_a$ is the largest weak asynchronous bisimulation.

We can exploit asynchronous bisimilarity as a form of observation of process behavior. We call an encoding *asynchronous bisimilarity preserving* if it is $\approx_a$-preserving. Requiring asynchronous bisimilarity preservation is not only more natural for an asynchronous calculus than requiring bisimilarity preservation, but it also makes the result stronger since bisimilarity preservation implies asynchronous bisimilarity preservation.

We can now prove a first separation result for the asynchronous case.

Theorem 3. *There is no good, asynchronous bisimilarity preserving encoding* $[\![\bullet]\!]_{g2s-a}$ *of asynchronous compensable processes into asynchronous static recovery processes.*

Proof. The proof adapts the technique used in the proof of Theorem 2. Suppose by contradiction that such an encoding exists. As in Theorem 2, for each i let $P_i = a_i.\mathsf{inst}\lfloor\lambda Y_i.b_i.\mathbf{0}\rfloor.\mathbf{0}$. 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 translation $[\![P]\!]_{g2s-a}$ should be of the form $C_t[\prod_{i\in\{1,\ldots,n\}}[\![P_i]\!]_{g2s-a}, [\![\mathbf{0}]\!]_{g2s-a}]$, which we will denote as $\tilde{P}$. Thanks to preservation of asynchronous bisimilarity $P \approx_a \tilde{P}$. First, $P \xrightarrow{a_i} P_i$ for each i, thus $\tilde{P} \stackrel{a_i}{\Longrightarrow} \tilde{P}_i$ for each i. Note, in fact, that $\tilde{P}$ cannot match the input action with a τ action, since otherwise we would need $P_i \approx_a \tilde{P}_i \mid \overline{a_i}$. This is however impossible since P_i would not be able to match the execution of the output $\overline{a_i}$.

Following the proof of Theorem 2, one can show that $\tilde{P} \Longrightarrow Q$ for some Q such that $Q \xrightarrow{a_i} Q'_i$ for each $i \in \{1,\ldots,n\}$. The proof technique is essentially the same, just note that all the processes obtained from $\tilde{P}$ via τ transitions should be bisimilar to P, since P cannot perform τ transitions.

Let us consider this process Q, which is again bisimilar to P. We can apply now Lemma 19 to prove that $Q = E[Q_1|Q_2]$ with $Q_1 \xrightarrow{a_j} Q'_1$ and $Q_2 \xrightarrow{a_k} Q'_2$ for some enabling context $E[\bullet]$. We have $E[Q_1|Q_2] \xrightarrow{a_j} E[Q'_1|Q_2] \xrightarrow{a_k} E[Q'_1|Q'_2]$ and $E[Q_1|Q_2] \xrightarrow{a_k} E[Q_1|Q'_2] \xrightarrow{a_j} E[Q'_1|Q'_2]$. The final process $E[Q'_1|Q'_2]$ is the same in both the cases.

Let us consider how P can match those transitions. The only possibility is to match

$$E[Q_1|Q_2] \xrightarrow{a_j} E[Q'_1|Q_2] \xrightarrow{a_k} E[Q'_1|Q'_2]$$

with

$$t[\prod_{i\in\{1,\ldots,n\}} P_i, \mathbf{0}] \xrightarrow{a_j}\xrightarrow{\tau}\xrightarrow{a_k}\xrightarrow{\tau} t[\prod_{i\in\{1,\ldots,n\}\setminus\{j,k\}} P_i, b_k.b_j.\mathbf{0}] = P_{j,k}$$

and

$$E[Q_1|Q_2] \xrightarrow{a_k} E[Q_1|Q'_2] \xrightarrow{a_j} E[Q'_1|Q'_2]$$

with

$$t[\prod_{i\in\{1,\ldots,n\}} P_i, \mathbf{0}] \xrightarrow{a_k}\xrightarrow{\tau}\xrightarrow{a_j}\xrightarrow{\tau} t[\prod_{i\in\{1,\ldots,n\}\setminus\{j,k\}} P_i, b_j.b_k.\mathbf{0}] = P_{k,j}$$

Note, in fact, that the first τ action corresponding to compensation update has to be executed so to make the following input enabled. We can also assume that the second compensation update is executed, since its execution will be needed anyway before executing any other observable action.

Thus we have $P_{j,k} \approx_a E[Q'_1|Q'_2] \approx_a P_{k,j}$, and, by transitivity, $P_{j,k} \approx_a P_{k,j}$. This is a contradiction, since $P_{j,k}$ has a computation with labels t, b_k which cannot be matched by $P_{k,j}$. Thus the encoding cannot exist. □

We now prove an even stronger result, based on testing. However, we have not been able to prove the result below using should testing as for the synchronous case, but we need a stronger form of observation, able to relate two special actions (while should testing considers just one). We call *guarded should testing* this form of testing equivalence, and define it below.

Definition 29 (Guarded should testing). Let P and O be processes and $\surd_g$ and $\surd$ two special names occurring in O but not in P. We call O a *guarded observer*. P guarded should O iff for each P' such that $P \mid O \stackrel{\overline{\surd_g}}{\Longrightarrow} P'$ we have $P' \stackrel{\overline{\surd}}{\Longrightarrow}$. Two processes P and Q are *guarded should testing equivalent*, written $P \simeq_{gshd} Q$, if, for each observer O, P guarded should O iff Q guarded should O.

Guarded should testing is stronger than should testing, but weaker than asynchronous weak bisimilarity.

Lemma 20. *Let P and Q be processes. $P \simeq_{gshd} Q$ implies $P \simeq_{shd} Q$.*

Proof. Assume $P \simeq_{gshd} Q$. We have to prove that $P \simeq_{shd} Q$. Let us consider an observer O. P should O iff P guarded should $\overline{\surd_g} \mid O$, and the same for Q. The thesis follows. □

Lemma 21. *Let P and Q be asynchronous processes. $P \approx_a Q$ implies $P \simeq_{gshd} Q$.*

Proof. The proof is similar to the one of Proposition 2. □

We call an encoding *guarded should testing preserving* if it is $\simeq_{gshd}$-preserving. The theorem below exploits this notion to prove a result stronger than the one in Theorem 3.

Theorem 4. *There is no good, guarded should testing preserving encoding $[\![\bullet]\!]_{g2s-a}$ of asynchronous compensable processes into asynchronous static recovery processes.*

Proof. Suppose by contradiction that such an encoding exists. For each i let $P_i = a_i(x).(\overline{x} \mid \mathsf{inst}\lfloor \lambda Y_i.b_i.(\overline{c}_i \mid Y_i)\rfloor.\mathbf{0})$. Let us 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 translation $[\![P]\!]_{g2s-a}$ should be of the form $C_t[\prod_{i\in\{1,\ldots,n\}} [\![P_i]\!]_{g2s-a}, [\![\mathbf{0}]\!]_{g2s-a}]$, which we will denote as $\tilde{P}$. Note that P is well formed, thus $P \simeq_{gshd} \tilde{P}$. From Lemma 20 $P \simeq_{shd} \tilde{P}$, and from Proposition 3 $P \simeq_{may} \tilde{P}$.

Let us consider the guarded observers $O_{j,k} = (\nu\, t_1, t_2, e)\, \overline{t_1} \mid t_1[\overline{a_j}\langle e\rangle, \overline{t_2} \mid t_2[\overline{a}_k\langle e\rangle, \overline{t}]] | e.e.\overline{\surd_g} | \overline{b}_k | c_k.\overline{b_j} | c_j.\overline{\surd}$ and $O'_{j,k} = (\nu\, t_1, t_2, e)\, \overline{t_1} | t_1[\overline{a_j}\langle e\rangle, \overline{t_2} | t_2[\overline{a}_k\langle e\rangle, \overline{t}]] | e.e.\overline{\surd_g} \mid \overline{b}_j \mid c_j.\overline{b_k} \mid c_k.\overline{\surd}$

The two guarded observers always perform an output on t, possibly preceded by an output on a_j or a_k, or an output on a_j followed by an output on a_k. It is not possible instead to have an output on a_k followed by an output on a_j.

By interacting with any of the guarded observers $O_{j,k}$ and $O'_{k,j}$, the tested process may install the following compensations: $\mathbf{0}$, $b_j.(\overline{c}_j \mid \mathbf{0})$, $b_k.(\overline{c}_k \mid \mathbf{0})$ and $b_k.(\overline{c}_k \mid b_j.(\overline{c}_j \mid \mathbf{0}))$ (corresponding to the cases above). The second part of the testing process $O_{j,k}$ can interact with the last compensation producing $\overline{\surd}$. The last compensation is installed iff both the inputs on a_j and a_k are executed, and this is the only case where two outputs on e become available, thus the only case in which $\overline{\surd_g}$ can be performed. Thus P guarded should $O_{j,k}$. Instead, $O'_{k,j}$ can produce $\overline{\surd_g}$, but never $\overline{\surd}$. Thus P may not $O'_{k,j}$ (note that any guarded observer is also an observer).

Given $O_j = (\nu\, e)\, \overline{a_j}\langle e\rangle \mid e.\overline{ok}$, 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 $\tilde{P} \stackrel{a_j(e)}{\Longrightarrow}$. Also, no action τ should compromise the possibility of performing a_j for each j, since we are using should testing equivalence.

One can show by contradiction that $\tilde{P} \Longrightarrow Q$ for some Q such that $Q \stackrel{a_i(e)}{\longrightarrow} Q'_i$ for each $i \in \{1, \ldots, n\}$. The proof follows the steps of the one in Theorem 2.

Observe that thanks to condition 2 of compositional encoding all $[\![P_i]\!]_{g2s-a}$ 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 18 the same relation holds also for Q. Thus we can apply Lemma 19 to prove that $Q = E[Q_1|Q_2]$ with $Q_1 \stackrel{a_j(e)}{\longrightarrow} Q'_1$ and $Q_2 \stackrel{a_k(e)}{\longrightarrow} Q'_2$ for some enabling context $E[\bullet]$. We have $E[Q_1|Q_2] \stackrel{a_j(e)}{\longrightarrow} E[Q'_1|Q_2] \stackrel{a_k(e)}{\longrightarrow} E[Q'_1|Q'_2]$ and $E[Q_1|Q_2] \stackrel{a_k(e)}{\longrightarrow} E[Q_1|Q'_2] \stackrel{a_j(e)}{\longrightarrow} E[Q'_1|Q'_2]$. The final process $E[Q'_1|Q'_2]$ is the same in both the cases.

Consider now the interactions of $\tilde{P}$ with the guarded observers $O_{j,k}$ and $O'_{k,j}$. For the first one we have:

$$\tilde{P} \mid O_{j,k} \Longrightarrow E[Q_1 \mid Q_2] \mid O_{j,k} \Longrightarrow E[Q'_1 \mid Q'_2] \mid \langle t\rangle \mid e.e.\overline{\surd_g} \mid \overline{b}_k \mid c_k.\overline{b_j} \mid c_j.\overline{\surd}$$

For the second one we have:

$$\tilde{P} \mid O'_{k,j} \Longrightarrow E[Q_1 \mid Q_2] \mid O_{j,k} \Longrightarrow E[Q'_1 \mid Q'_2] \mid \langle t\rangle \mid e.e.\overline{\surd_g} \mid \overline{b}_k \mid c_k.\overline{b_j} \mid c_j.\overline{\surd}$$

The final process is the same in both the cases.

Consider now observer $O''_{j,k} = (\nu\, e)\, \overline{a_j}\langle e\rangle \mid \overline{a_k}\langle e\rangle \mid e.e.\overline{\surd}$. P should $O''_{j,k}$ for each j, k, thus the same should hold for $\tilde{P}$. This means that in the case above $E[Q'_1 \mid Q'_2] \mid \langle t\rangle \mid e.e.\overline{\surd_g} \mid \overline{b}_k \mid c_k.\overline{b_j} \mid c_j.\overline{\surd} \Longrightarrow Q' \mid \langle t\rangle \mid \overline{ok_g} \mid \overline{b}_k \mid c_k.\overline{b_j} \mid c_j.\overline{\surd} \stackrel{\overline{\surd_g}}{\longrightarrow}$

$Q' \,|\, \overline{b}_k \,|\, c_k.\overline{b_j} \,|\, c_j.\overline{\surd}$. Since $\tilde{P}$ guarded should $O_{j,k}$ we have $Q' \,|\, \overline{b}_k \,|\, c_k.\overline{b_j} \,|\, c_j.\overline{\surd} \stackrel{\overline{\surd}}{\Longrightarrow}$. However from $\tilde{P}$ may not $O'_{k,j}$ we have that such a computation cannot exist.

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

We have not been able to prove the result above requiring $\simeq_{shd}$-preservation instead of $\simeq_{gshd}$-preservation. This would be a stronger result.

Many approaches in the literature, such as [26], use as observers in the target language the translation of the observers in the source language. This is needed if the source and the target calculus are unrelated: in our case we can use the same observers since the target calculus is a subcalculus of the source one. We can restate our results using the approach in [26], but we need some more conditions ensuring that the encoding preserves the behavior of the observers.

The theorems above concern general dynamic recovery, however similar results 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. Compensable processes however are based on π-calculus, thus they exploit prefixing instead of sequential composition. Separation results corresponding to Theorem 2 and Theorem 3 can be easily proved considering just a very constrained form of backward recovery, where P is a single prefix, instead of general dynamic recovery.

Corollary 2. *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 2. □

Corollary 3. *There is no good, asynchronous bisimilarity preserving encoding $[\![\bullet]\!]_{b2s-a}$ of asynchronous backward recovery processes into asynchronous 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 3. □

The compensation update used in the proof of Theorem 4 is slightly more complex than $\lambda Y_i.b_i.Y_i$. It contains, in fact, also an output message. However, since the calculus is asynchronous, such a compensation update would be allowed by backward recovery. For instance, using sequential composition such an observer can be written as $\lambda Y_i.b_i;\overline{c}_i;Y_i$ where the fact that the output is asynchronous is modeled by using a two-steps semantics for output as done, for instance, in [37].

Corollary 4. *There is no good, guarded should testing preserving encoding $[\![\bullet]\!]_{g2s-a}$ of asynchronous backward recovery processes into asynchronous static recovery processes.*

Proof. The proof is the same of Theorem 4, observing that the used compensation updates are valid backward recovery compensation updates. □

Combining the results of Section 4 with the results in this section we also deduce that general dynamic recovery and backward recovery are both more expressive than parallel recovery. We state here the result that corresponds to Theorem 2. All the results in this section can be generalized in a similar way.

Corollary 5. *There is no good encoding* $[\![\bullet]\!]_{g2p}$ *of compensable processes into parallel recovery processes.*

Proof. The proof is by contradiction. Assume that such an encoding exists. Since the composition of two good encodings is a good encoding, one could compose $[\![\bullet]\!]_{g2p}$ and the encoding $[\![\bullet]\!]_{p2s}$ from parallel recovery processes to static recovery processes described in Section 4 to have an encoding from compensable processes into static recovery processes. However such an encoding cannot exist thanks to Theorem 2. This is a contradiction. □

All the results of this section hold under aborting, preserving and discarding semantics for transaction nesting.

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 [6, 7, 8, 34, 12, 13, 21], which are obtained by adding primitives for compensation handling on top of concurrent calculi such as π-calculus [4] or Join [38]. These calculi differ on many design choices. The main differences are summarized in Table 1 and their impact on our results discussed below.

The meaning of the different columns is as follows:

communication mechanism: which is the communication mechanism used in the calculus: the one of CCS [5], of π-calculus [4], of asynchronous π-calculus [18], of Join calculus [38], or communication based on correlation sets;

compensation definition: whether compensations are defined using static recovery, parallel recovery or general dynamic recovery;

nested transactions: which is the semantics used for nested transactions: they can be aborted, preserved, discarded, or their termination can be waited for;

protection operator: whether the calculus provides or not a protection operator; "implem." here means that the calculus has no native protection operator, but it can be implemented using the available operators;

	communic. mechanism	compens. definition	nested transactions	protection operator	encoding applicable	separation applicable
webπ [8]	asynch. π	static	preserved	implem.	no	Th. 6
webπ_∞ [34]	asynch. π	static	preserved	implem.	yes	Th. 6
dcπ [12]	asynch. π	parallel	aborted	yes	yes	Th. 3, 4
πt [6]	asynch. π	static	waited	no	no	Th. 3
ATc [13]	π	parallel	discarded	no	yes [a]	Th. 2
TransCCS [21]	CCS	static	discarded	no	yes [a]	Th. 2
c-join [7]	join	static	aborted	no	yes [a]	no
SOCK [17]	correlation	dynamic	aborted	implem.	yes	no
COWS [39]	correlation	static	aborted	yes	yes	no
Jolie [40]	correlation	dynamic	aborted	implem.	yes	no
WS-BPEL [3]	correlation	static	aborted	implem.	yes	no

[a] If a protection operator is added.

Table 1: Features of interaction-based calculi and languages.

encoding applicable: whether the encoding defined in Section 4 can be applied to the calculus or not; for calculi featuring parallel recovery or general dynamic recovery this means whether or not the parallel recovery subcalculus can be encoded in the static recovery subcalculus; for calculi featuring static recovery it means whether or not a parallel recovery mechanism can be encoded in the calculus;

separation applicable: whether the separation results from Section 5 can be applied or not. In case they are applicable we write the reference to the theorem that best applies. Theorem 6 is presented later on in this section. As for the applicability of the encoding, the applicability of separation results is referred to the static and general dynamic fragments/extensions of the calculus.

We now describe in detail the relationships between compensable processes and the calculi in the table. We start with webπ_∞, since this is one of the most related calculi to ours, and we show in detail how the results can be applied to it. This can be used as a guideline for understanding how to apply those results to other calculi. For this reason, the comparisons with other calculi are less detailed.

6.1. Application to webπ_∞

Webπ_∞ [34] is a calculus with static recovery based on asynchronous π-calculus [18]. It is the untimed fragment of webπ [8], which is described later.

In this section we study whether the encoding in Section 4 and the separation results in Section 5 can be applied to webπ_∞. Since webπ_∞ is based on static recovery, the two results show whether or not suitable extensions of webπ_∞ with parallel and general dynamic recovery can be encoded in the basic calculus. We

will show that both the encoding in Section 4 and the separation result in Theorem 3 can be applied to webπ_∞ with minor adaptations. Before doing this, we compare the design choices of webπ_∞ with the ones of static recovery processes.

Webπ_∞ is strongly related to asynchronous static recovery processes with preserving semantics for transaction nesting. Actually, the syntax of the two calculi is the same but for some notational differences (also, webπ_∞ has no protection, but this can be easily encoded as discussed below), but the two semantics differ in a few points, discussed below:

1. output messages in webπ_∞ are not deleted in case of abort of the enclosing transaction scope: the same behavior can be obtained in static recovery processes by enclosing each output message inside a protected block, i.e., by defining the syntax for output messages as $\langle \overline{a}\langle \vec{v}\rangle\rangle$.
2. in webπ_∞ messages and subtransactions can escape from transaction scopes using structural congruence: actually, this has no influence on the semantics since the external transaction scope has no influence on subtransactions neither on messages. For subtransactions, this depends on the fact that webπ_∞ has preserving semantics. For messages, this depends on the fact that messages are protected, as discussed in the previous point.
3. protected blocks are not available in webπ_∞: webπ_∞ has preserving semantics for transaction nesting, thus transaction scopes can be used to mimic protected blocks. In particular, given a fresh name t, $(\nu\, t)\, t[P, \mathbf{0}]$ has the same behavior of $\langle P\rangle$. Note that in webπ_∞, when a transaction scope $t[P, Q]$ is aborted (e.g., from the outside), it becomes, assuming that P contains no messages or subtransactions, $(\nu\, t)\, t[Q, \mathbf{0}]$, which corresponds exactly to $\langle Q\rangle$.
4. a transaction scope $t[P, Q]$ can only be aborted if P contains at least one input, *i.e.*, once P completes the transaction scope cannot be aborted any more. To match this behavior one would need to change the semantics of static recovery processes, by adding a condition checking that there are inputs inside the body of the transaction scope as a premise to both rule (L-Recover-out) and rule (L-Recover-out). This can be done using a predicate $\mathsf{inp}(P)$ defined as for webπ_∞.

As we have seen, the only two differences which have a relevant impact on the semantics of the calculus are 1 and 4. Luckily, those two differences have no impact on the encoding of parallel recovery into static recovery. We detail below how the encoding in Definition 20 can be reformulated using webπ_∞ semantics (we stick however to the notation of compensable processes). The only notable difference is that protection is simulated by transaction scopes with bound names, according to item 3 in the list above.

Definition 30 (From webπ_∞ with parallel recovery to webπ_∞).
Let r, s be fixed fresh names. The encoding $[\![\bullet]\!]_{wp2w}$ from webπ_∞ with parallel

recovery processes to webπ_∞ processes is defined as:

$$\begin{array}{rcl} [\![t[P,Q]]\!]_{wp2w} & = & (\nu\, r)\, t\, [[\![P]\!]_{wp2w}, [\![Q]\!]_{wp2w} \mid \overline{r}] \\ [\![\mathsf{inst}\lfloor \lambda X.Q \mid X \rfloor.P]\!]_{wp2w} & = & [\![P]\!]_{wp2w} \mid (\nu\, s)\, s\, [r.([\![Q]\!]_{wp2w} \mid \overline{r}), \mathbf{0}] \end{array}$$

All the remaining operators are homomorphically mapped to themselves.

To prove that $[\![\bullet]\!]_{wp2w}$ is a good encoding one can follow the approach described in Section 4. The results in Section 4 are stated on a synchronous calculus, while webπ_∞ is asynchronous. However, the encoding of an asynchronous process is still an asynchronous process, thus the result can be applied in an asynchronous setting too (see Remark 1). We prove the correctness theorem below. The notion of bisimilarity preservation suitable for webπ_∞ is asynchronous, since webπ_∞ is asynchronous, but requires also to preserve the $\mathsf{inp}(\bullet)$ predicate. This is needed to have a compositional equivalence, as argued in [34]. We call such a notion *weak asynchronous webπ_∞ bisimilarity*. However, the encoding preserves any reasonable weak equivalence, including weak synchronous bisimilarity and weak asynchronous bisimilarity.

Theorem 5. *$[\![\bullet]\!]_{wp2w}$ is a good, weak asynchronous webπ_∞ bisimilarity preserving encoding.*

Proof. Following Definition 16, $[\![\bullet]\!]_{wp2w}$ is a good encoding if it is compositional, correct, and divergence reflecting. The proofs for compositionality and divergence reflection are exactly as in Section 4. For proving that the encoding is correct we prove that it preserves weak asynchronous webπ_∞ bisimilarity (which implies should testing equivalence for asynchronous observers). The proof follows the same lines of the one in Section 4. The main differences are that:

- in all processes, protected blocks are replaced by transaction scopes with bound names;
- proofs related to the behavior of the extraction function (for instance, Lemma 6) are easier, since webπ_∞ is based on the preserving semantics;
- for checking bisimilarity one has to check that the $\mathsf{inp}(\bullet)$ predicate is preserved; however all such checks are easy. □

We now prove that webπ_∞ extended with general recovery operators cannot be encoded in webπ_∞, transporting to webπ_∞ the results in Section 5. Since webπ_∞ is an asynchronous calculus, the results of Theorem 3 and of Theorem 4 are of interest.

Adapting the first one to webπ_∞ requires no effort. However, as for the correctness of the encoding, the suitable notion of equivalence to be preserved is weak asynchronous webπ_∞ bisimilarity. Since this notion is stronger than weak asynchronous bisimilarity, the proof has no need to be changed to accommodate the new notion.

Theorem 6. *There is no good, weak asynchronous webπ_∞ bisimilarity preserving encoding $[\![\bullet]\!]_{gw2w}$ of webπ_∞ with general recovery processes into webπ_∞ processes.*

PROOF. The proof is essentially equal to the proof of Theorem 3. □

For Theorem 4, the situation is rather different. The proof of Theorem 4 strongly relies on output messages being killed by transaction abort, but this requirement is not satisfied in webπ_∞. Thus the proof strategy cannot be applied.

6.2. Relations with other calculi

We discuss here the relations between compensable processes and other calculi/languages in the literature, focusing on whether and how our results can be applied to them.

Webπ. Webπ [8] is the extension of webπ_∞ with time: webπ transactions can abort either because of an incoming error notification or because the time allowed for the transaction expired. Thus webπ processes are time sensitive, and the encoding does not work any more: a timed context can notice that the original process and its translation take different amounts of time to execute. It is not possible to statically keep this difference into account, since it depends on the number of compensation updates. As far as impossibility results are concerned, the approach used for webπ_∞ can be applied here, adapting Theorem 6. We guess that the additional observational power provided by time sensitivity can allow to prove stronger results and/or to simplify the proofs. However, we are not interested here in studying the expressive power of timed calculi, thus we leave this issue for future work.

Dcπ. The Dynamic Compensation calculus [12] is a calculus with parallel recovery based on asynchronous π-calculus. 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 $a(\vec{x})\%Q.P$ that, after receiving values $\vec{v}$ on channel a, continues as $P\{\vec{v}/\vec{x}\}$ and adds $Q\{\vec{v}/\vec{x}\}$ in parallel to the current compensation. The same behavior can be obtained in parallel recovery processes by writing $a(\vec{x}).\,\mathsf{inst}\lfloor\lambda X.Q \mid X\rfloor.P$. In fact, even if in compensable processes the input and the compensation update are two separate steps, the overall result is the same, and atomicity is enforced by priority of compensation update. Thus dcπ can be seen as the asynchronous fragment of parallel recovery processes with aborting semantics for transaction nesting where compensation update can occur only after input prefix. The encoding in Section 4 can be easily adapted to dcπ, allowing to encode dcπ into its static recovery fragment. Similarly Theorem 3 and Theorem 4 can be applied to dcπ, showing that the extension of dcπ with general dynamic recovery cannot be encoded in dcπ. Note that to define such an extension one would need to change also the runtime syntax of dcπ.

πt-calculus. The πt-calculus [6] is based on asynchronous π-calculus. When a process aborts, abortion or completion of parallel processes (including parallel transactions) is waited for. Then, compensations of successfully terminated parallel transactions are executed (in parallel), followed by the failure manager of the enclosing transaction, which is in charge of managing the abort. Both compensations and failure managers are defined statically. It is difficult to adapt the encoding in Section 4 to πt-calculus. One will need to add protected blocks to the calculus, and to change the semantics of abort/commit. In fact, abort/commit in πt-calculus waits for the termination of all parallel processes inside the same transaction scope, and processes produced by the encoding include protected blocks which cannot terminate before the compensation of the transaction is executed. Concerning separation results, Theorem 3 can be adapted to prove that adding general dynamic recovery to πt-calculus (allowing to dynamically update failure managers or compensations) would increase its expressive power. As for webπ_∞, Theorem 4 cannot be adapted, since transaction abort in πt-calculus never deletes output messages.

ATc. The Attribute-based Transactional calculus [13] is an extension of π-calculus [4] to model the dynamic reconfiguration of transactions described by Enterprise Java Beans [41]. As for dcπ, in ATc prefixes include a compensation update (both input and output prefixes, since ATc is synchronous), and the new compensation item is added in parallel to the current compensation. ATc is based on the discarding semantics for transaction nesting. Our results apply quite straightforwardly to ATc. For instance, the encoding in Section 4 can be applied to map ATc into its static fragment, provided that a protection operator is added to the calculus. Also, Theorem 2 can be used to show that adding general dynamic recovery to the calculus would increase its expressive power.

TransCCS. TransCCS [21] is a calculus for modeling communicating transactions. Part of the context can be imported into the body of the transaction scope (in parallel), provided that it is also copied into the compensation (in parallel), so to restore the environment state in case of failure. This is different from our parallel recovery, since this is automatic and the added code comes from outside the transaction scope. Thus, allowing the process into the transaction scope to update the compensation code (using parallel recovery or general dynamic recovery) is an orthogonal feature. Our results apply to such an extension. Remember, in fact, that the encoding can be adapted to CCS-based languages (see Remark 1). Similarly, Theorem 2 can be adapted (see Remark 3).

C-join. C-join [7] is a calculus with static recovery built on top of Join calculus [38]. However here transaction scopes can be dynamically merged, and their compensations are composed in parallel, thus obtaining some form of parallel recovery. The encoding in Section 4 is not directly applicable since C-join has no protected block operator, but it becomes applicable

as soon as such an operator is introduced. As far as separation results are concerned, join patterns are more powerful than π-calculus communication, and we conjecture that they can be used to implement general dynamic recovery. In fact, the separation results are based on the fact that compensation update and the input prefix enabling it have to be executed atomically, and it is difficult to satisfy such an atomicity constraint. However, one can use a join pattern with two arguments to model an input and at the same time capture a mutual exclusion token. The capture of the mutual exclusion token (which would be required to perform any activity related to error handling) would ensure that compensation update is completed before any other error handling activity could be done.

Service oriented calculi. Many service oriented calculi have been recently proposed [42, 43, 44, 39, 45, 46]. Since long running transactions are an important aspect of service oriented computing, many of these calculi include primitives for error and compensation handling. We discuss here the ones more related to our approach.

SOCK [43] is a calculus allowing to compose service invocations and definitions using primitives from sequential languages and concurrent calculi. It has been extended with primitives for general dynamic recovery in [17]. The encoding in Section 4 can be applied to SOCK by exploiting signals (SOCK synchronization mechanism) to enable compensation items. Actually, in SOCK, the protected block is just used in the definition of the semantics, but it can be implemented too. In fact, $(\nu\, t)\, t[\overline{t}, Q]$, which has a straightforward translation into SOCK, reduces to $\langle Q \rangle$ (both on its own and in case of an external abort before the synchronization on t). 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 global state allows to track the order in which parallel actions are executed. However, exploiting this information for selecting the order of execution of compensation items is not trivial, and we do not know whether this can be done in a compositional way.

The discussion made for SOCK above applies also to Jolie [40, 47], a full-fledged language for service oriented computing based on it.

COWS [39] communication is in the style of fusion calculus [48]. COWS has a kill primitive and a protected block. This allows to program static recovery as shown in [39]. 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. As for SOCK, it is not clear whether such a feature allows to program general dynamic recovery.

Other service oriented calculi either have no error handling mechanisms [44], or they include only basic mechanisms for exception handling [45] or notification of session failure [42, 46].

Compensable flow calculi. Compensable flow calculi, such as SAGAs calculi [11] or StAC [10], concentrate on how to derive compensations for complex transactions from compensations for basic activities. These calculi use backward recovery for sequential activities and parallel recovery for parallel ones. However, these calculi are at a more abstract level, describing which compensation items are executed and when, but abstracting away from most of the details: compensation policies are hard-wired in the calculus semantics, not specified by the programmer. For this reason, more than analyzing which mechanisms they can express, it is interesting to analyze whether they can be implemented on top of more basic calculi. Such a problem has been studied for instance in [9, 49]. Comparisons among different compensable flow calculi, concentrating on classifying the different hard-wired policies, have been presented in [50, 51].

WS-BPEL. WS-BPEL [3] is the de-facto standard for web services composition, proposed by Oasis. Compensations are statically defined, and they are composed using backward recovery for sequential subtransactions, and parallel recovery for parallel ones. The analysis of the applicability of our results to WS-BPEL follows the same reasonings discussed for SOCK. Summarizing, the encoding can be applied, while it is not clear how to adapt the proofs of the separation results. Note that from BPEL documentation [3] it is not clear whether compensation execution is protected from external faults as in SOCK, thus allowing to implement protected blocks. However this is what happens in most of the implementations, e.g. in ActiveBPEL [52], and in the formal semantics described in [53].

6.3. Conclusion and future work

We have presented different results concerning the expressive power of primitives for compensation handling: an encoding of parallel recovery into static recovery, and separation results showing that general dynamic recovery and backward recovery cannot be encoded using static recovery. The general approach that we support consists in analyzing the problem in a simple formal setting, and then transporting the results to more complex calculi and languages, where the feature under analysis will interact with other aspects. We have presented our main results in sections 4 and 5, and discussed their applicability to other calculi in the first part of this section. The proof techniques used in Section 5 are original, different e.g. from the ones considered in [24, 27].

As we already discussed throughout the paper, many open issues concerning the expressive power of mechanisms for long running transactions still remain. In fact this topic, while relevant, has been neglected until now. We already discussed in the first part of the section the most related works in the literature. Other papers, such as [54, 55], 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 the aborting semantics for transaction nesting can be implemented

using the preserving one, and to the encodability of BPEL-style recovery into static recovery. We conjecture that this last encoding is possible, thus BPEL-style recovery could be defined as a macro on top of static recovery. After these 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 results. It has been shown in [56] that such a kind of synchronization increases the expressive power of CCS, but its impact on our results has to be analyzed. Similarly, the impact of priority on our results needs investigation. The effect of priority on the expressive power of π-calculus has been studied for instance in [57], and its effect on behavioral equivalence has been studied also for service oriented calculi [58].

Thus, there are still many missing pieces in the puzzle. Filling them would allow to have a deep understanding of the relationships among the different design choices involved in the definition of mechanisms for long running transactions, and of their relationships with other aspects of the underlying language. This would allow to put some order on the current flurry of proposals, and to choose the ones that best fit each purpose based on technical reasons.

Acknowledgments.. We thank Claudio Antares Mezzina and Gianluigi Zavattaro for useful suggestions and comments.

References

[1] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services—Concepts, Architectures and Applications. Springer (2003)

[2] Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4) (1983) 287–317

[3] 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`.

[4] Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Inf. Comput. **100**(1) (1992) 1–40,41–77

[5] Milner, R.: A Calculus of Communicating Systems. Number 92 in LNCS. Springer (1982)

[6] Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In: Proc. of FMOODS'03. Number 2884 in LNCS, Springer (2003) 124–138

[7] Bruni, R., Melgratti, H., Montanari, U.: Nested commits for mobile calculi: Extending join. In: Proc. of IFIP TCS'04, Kluwer (2004) 563–576

[8] Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Proc. of FoSSaCS'05. Number 3441 in LNCS, Springer (2005) 282–298

[9] Caires, L., Ferreira, C., Vieira, H.: A process calculus analysis of compensations. In: Proc. of TGC'08. Number 5474 in LNCS, Springer (2008) 87–103

[10] Butler, M.J., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In: Proc. of COORDINATION'04. Number 2949 in LNCS, Springer (2004) 87–104

[11] Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: Proc. of POPL '05, ACM Press (2005) 209–220

[12] Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Proc. of TGC'08. Number 5474 in LNCS, Springer (2008) 201–215

[13] Bocchi, L., Tuosto, E.: A java inspired semantics for transactions in SOC. In: Proc. of TGC 2010. Number 6084 in LNCS, Springer (2010) 120–134

[14] Butler, M.J., Hoare, C., Ferreira, C.: A trace semantics for long-running transactions. In: 25 Years CSP. Number 3525 in LNCS. Springer (2004) 133–150

[15] Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., Salem, K.: Coordinating multi-transaction activities. Technical Report CS-TR-2412, University of Maryland, Dept. of Computer Science (1990)

[16] Weimer, W., Necula, G.: Finding and preventing run-time error handling mistakes. In: Proc. of OOPSLA'04, ACM Press (2004) 419–431

[17] Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: Proc. of ACSD'08, IEEE Computer Society Press (2008) 190–199

[18] Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: Proc. of ECOOP'91. Number 512 in LNCS. Springer (1991) 1–20

[19] Lanese, I., Vaz, C., Ferreira, C.: On the expressive power of primitives for compensation handling. In: Proc. of ESOP 2010. Number 6012 in LNCS, Springer (2010) 366–386

[20] Milner, R.: The polyadic pi-calculus: a tutorial. Technical Report ECS-LFCS-91-180, School of Informatics at the University of Edinburgh (1991)

[21] de Vries, E., Koutavas, V., Hennessy, M.: Communicating transactions. In: Proc. of CONCUR 2010. Number 6269 in LNCS, Springer (2010) 569–583

[22] Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundamenta Informaticae **95(1)** (2009) 73–102

[23] Parrow, J.: Expressiveness of process algebras. In: Proc. of the LIX Colloquium on Emerging Trends in Concurrency Theory. Number 209 in ENTCS, Elsevier (2008) 173–186

[24] Palamidessi, C.: Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In: Proc. of POPL'97. (1997) 256–265

[25] Nestmann, U.: What is a "good" encoding of guarded choice? Inf. Comput. **156**(1-2) (2000) 287–319

[26] Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. In: Proc. of CONCUR'08. Number 5201 in LNCS, Springer (2008) 492–507

[27] Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. Inf. Comput. **208**(9) (2010) 1031–1053

[28] Rensink, A., Vogler, W.: Fair testing. Inf. Comput. **205**(2) (2007) 125–198

[29] De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. **34** (1984) 83–133

[30] Milner, R.: Communication and Concurrency. Prentice Hall (1989)

[31] de Vries, E., Koutavas, V., Hennessy, M.: Liveness of communicating transactions. In: Proc. of APLAS 2010. Number 6461 in LNCS, Springer (2010) 392–407

[32] Sangiorgi, D., Walker, D.: Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)

[33] Thomsen, B.: Calculi for Higher Order Communicating Systems. PhD thesis, Imperial College (1990)

[34] Mazzara, M., Lanese, I.: Towards a unifying theory for web services composition. In: Proc. of WS-FM'06. Number 4184 in LNCS, Springer (2006) 257–272

[35] Fournet, C., Gonthier, G.: A hierarchy of equivalences for asynchronous calculi. Journal of Logic and Algebraic Programming **63**(1) (2005) 131–173 Special issue on The pi-calculus.

[36] Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. Theor. Comput. Sci. **195**(2) (1998) 291–324

[37] Lanese, I., Guidi, C., Montesi, F., Zavattaro, G.: Bridging the gap between interaction- and process-oriented choreographies. In: Proc. of SEFM'08, IEEE Computer Society Press (2008) 323–332

[38] Fournet, C., Gonthier, G.: The reflexive CHAM and the join-calculus. In: Proc. of POPL'96, ACM Press (1996) 372–385

[39] Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: Proc. of ESOP'07. Number 4421 in LNCS, Springer (2007) 33–47

[40] Montesi, F., Guidi, C., Zavattaro, G.: Composing services with JOLIE. In: Proc. of ECOWS'07, IEEE Computer Society Press (2007) 13–22

[41] Sun Mycrosystems: Enterprise javabeans (ejb) technology. `http://java.sun.com/products/ejb/` (2009)

[42] Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., Zavattaro, G.: SCC: a Service Centered Calculus. In: Proc. of WS-FM'06. Number 4184 in LNCS, Springer (2006) 38–57

[43] Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: Proc. of ICSOC'06. Number 4294 in LNCS, Springer (2006) 327–338

[44] Lanese, I., Martins, F., Vasconcelos, V., Ravara, A.: Disciplining orchestration and conversation in service-oriented computing. In: Proc. of SEFM'07, IEEE Computer Society Press (2007) 305–314

[45] Vieira, H., Caires, L., Seco, J.: The conversation calculus: A model of service-oriented computation. In: Proc. of ESOP'08. Number 4960 in LNCS, Springer (2008) 269–283

[46] Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: Proc. of FMOODS'08. Number 5051 in LNCS, Springer (2008) 19–38

[47] Jolie team: Jolie website. `http://www.jolie-lang.org/`

[48] Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: Proc. of LICS'98. (1998) 176–185

[49] Lanese, I., Zavattaro, G.: Programming Sagas in SOCK. In: Proc. of SEFM'09, IEEE Computer Society Press (2009) 189–198

[50] Bruni, R., Butler, M.J., Ferreira, C., Hoare, C.A.R., Melgratti, H.C., Montanari, U.: Comparing two approaches to compensable flow composition. In: Proc . of CONCUR'05. Number 3653 in LNCS, Springer (2005) 383–397

[51] Lanese, I.: Static vs dynamic sagas. In: Proc. of ICE 2010. Number 38 in EPTCS (2010) 51–65

[52] Active Endpoints: Activebpel 4.1. `http://www.active-endpoints.com` (2007)

[53] Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL. In: Proc. of COORDINATION'08. Number 5052 in LNCS, Springer (2008) 199–215

[54] Aceto, L., Fokkink, W., Ingólfsdóttir, A., Nain, S.: Bisimilarity is not finitely based over bpa with interrupt. Theor. Comput. Sci. **366**(1-2) (2006) 60–81

[55] Bravetti, M., Zavattaro, G.: On the expressive power of process interruption and compensation. Math. Stru. Comp. Sci. **19**(3) (2009) 565–599

[56] Laneve, C., Vitale, A.: The expressive power of synchronizations. In: Proc. of LICS 2010, IEEE Computer Society Press (2010) 382–391

[57] Versari, C., Busi, N., Gorrieri, R.: An expressiveness study of priority in process calculi. Math. Stru. Comp. Sci. **19**(6) (2009) 1161–1189

[58] Pugliese, R., Tiezzi, F., Yoshida, N.: On observing dynamic prioritised actions in SOC. In: Proc. of ICALP'09 (2). Number 5556 in LNCS, Springer (2009) 558–570