

A Java Inspired Semantics for Transactions in SOC[★]

Laura Bocchi and Emilio Tuosto

Department of Computer Science, University of Leicester, UK

Abstract. We propose a formal semantics for distributed transactions inspired by the attribute mechanisms of the Java Transaction API. Technically, we model services in a process calculus featuring transactional scope mechanisms borrowed from the so called *container-managed* transactions of Java. We equip our calculus with a type system for our calculus and show that, in well-typed systems, it guarantees absence of run-time errors due to misuse of transactional mechanisms.

1 Introduction

The *Service-Oriented Computing* (SOC) paradigm envisages distributed systems as loosely-coupled computational entities which dynamically discover each other and bind together. Although appealing, SOC has imposed to re-think, among other classic concepts, the notion of transaction. The long lasting and cross-domain nature of SOC makes typically unfeasible to adopt ACID transactions, which are implemented by locking the involved resources. The investigation of formal semantics of SOC transactions (often referred to as long-running transactions) has been a topic of focus in the last few years (see § 6 for a non-exhaustive overview). Central to this investigation is the notion of *compensation* (a weaker and “ad hoc” version of the classic rollback of database systems) which has mainly been studied in relation to mechanisms of failure propagation.

In this paper we address an orthogonal topic, namely the semantics of dynamic re-configuration of transactions in SOC which, to the best of our knowledge, has not been explicitly considered. In SOC, the configuration of a system can change at each service invocation to include a new instance of the service in the ongoing computation. There is still a lack of agreement on how the run-time reconfiguration should affect the relationships between existing and newly created transactional scopes. To illustrate the main problems, we consider the following example:

$$\langle \text{invoke } s.P \mid \langle C \rangle \rangle \quad \text{with } s \text{ implemented as } Q \quad (1)$$

where a process in a transactional scope (represented by the angled brackets) with compensation C invokes a service s and then behaves like P ; the invocation triggers a (possibly remote) instance Q of the service s . Should the system in (1) evolve to a transactional scope that includes Q (i.e., $\langle P \mid Q \mid \langle C \rangle \rangle$)? Should instead Q be running in a

[★] This work has been partially sponsored by the project Leverhulme Trust Award “Tracing Networks”. The authors also thank Hernan Melgratti for his valuable comments on a preliminary draft of this paper.

different scope (i.e., $\langle P \mid \langle C \rangle \rangle \mid \langle Q \rangle$)? Or should Q be executed outside any transactional scope (i.e., $\langle P \mid \langle C \rangle \rangle \mid Q$) or else raise an exception triggering the compensation C ? Notice that each alternative is valid and has an impact on failure propagation.

Enterprise Java Beans (EJB) promote *Container Managed Transactions* (CMT) as a mechanism to control dynamic reconfigurations. We take inspiration from the EJB mechanism and adapt it to SOC transactions. A *container* can be used to publish objects and can specify:

- the transactional modality of method calls (e.g., “*calling the method fooBar from outside a transactional scope throws an exception*”),
- how the scope of transactions dynamically reconfigure (e.g., “*fooBar is always executed in a newly created transactional scope*”).

A limitation of CMT is that it only permits to declare transactional modalities for the methods to be invoked and does not allow invokers to specify their own requirements on the needed transactional support. On the contrary service invocations are resolved at run-time and different providers may publish different implementations of a service. Hence, it is natural to give the invoker the opportunity to express some requirements on the transactional behaviour of the invoked services. For instance, in (1) the invocation to s may require that Q must be executed in the same transactional scope of P .

We do not aim to provide a semantics for CMT but rather investigate how CMT could be borrowed to address the issues described above for SOC transactions. We promote some CMT inspired primitives for SOC which allow invokers (and not just callees) to specify their own transactional requirements. Furthermore, we give a typing discipline to ensure that invocations do not yield run-time errors due to the incompatibility of the transactional modalities required by callers and those guaranteed by callees.

Our main contributions are

1. a semantics to specify dynamic reconfiguration of SOC transactions inspired by the CMT mechanisms of EJB; namely, we introduce a CCS-like process calculus called ATc (after *Attribute-based Transactional calculus*)
2. a type system that guarantees that no error will occur for a method invocation due to the incompatibility of the transactional scopes of caller and callee
3. a methodology for designing SOC transactions based on our typing discipline.

Synopsis. The transactional mechanisms of EJB are summarised in § 2. The syntax and semantics of ATc are introduced in § 3. The typing discipline of ATc is in § 4. In § 5 we give a gist of how our type system can be used to design systems correct wrt dynamic reconfigurations of transactions. Conclusions and related work are discussed in § 6.

2 EJB Transactional Attributes

Roughly, a *Java bean* can be thought of as an object amenable to be executed in a specialised run-time environment called *container* (see e.g., [19,18]). An EJB container supports typical functionalities to manage e.g. the life-cycle of a bean and to make components accessible to other components by binding it to a naming service¹.

¹ <http://docs.sun.com/app/docs/doc/819-3658/ablmw?a=view>

For the sake of this paper, we focus on the transactional mechanisms offered by EJB-containers. Specifically, we consider *Container Managed Transactions* (CMT) whereby a container associates each method of a bean with a *transactional attribute* specifying the modality of reconfiguring transactional scopes. We denote the set of EJB transactional attributes as

$$(EJB \text{ Transactional Attributes}) \quad \mathcal{A} \stackrel{def}{=} \{m, s, n, ns, r, rn\}$$

where, following the EJB terminology, *m* stands for *mandatory*, *s* for *supported*, *n* for *never*, *ns* for *not supported*, *r* for *requires*, and *rn* for *requires new*.

The intuitive semantics of EJB attributes \mathcal{A} (ranged over by a, a_1, a_2, \dots) is illustrated in Figure 1 where each row represents the behaviour of one transactional attribute and shows how the transactional scope (represented by a rectangular box) of the caller (represented by a filled circle) and callee (represented by an empty circle) behave upon invocation. The first two columns of Figure 1 represent, respectively, invocations from outside and from within a transactional scope. More precisely, (1) a callee supporting *r* is always executed in a transactional scope which happens to be the same as the caller’s if the latter is already running in a transactional scope; (2) a callee supporting *rn* is always executed in a new transactional scope; (3) a callee supporting *ns* is always executed outside a transactional scope; (4) the invocation to a method supporting *m* fails if the caller is not in a transactional scope (first column of the fourth row in Figure 1), otherwise the method is executed in the transactional scope of the caller; (5) the invocation to a method supporting *n* is successful only if the caller is outside a transactional scope, and it fails if the caller is running in a transactional scope (in this case an exception is triggered in the caller); (6) a method supporting *s* is executed inside (resp. outside) the caller’s scope if the caller is executing in (resp. outside) a scope.

In this paper, we adapt the transactional model of EJB to the context of SOC, where each provider can be thought of as a container specifying a number of services together with their transactional attribute. A transactional attribute declares whether a published service must or must not be executed within a transactional scope and the modality of

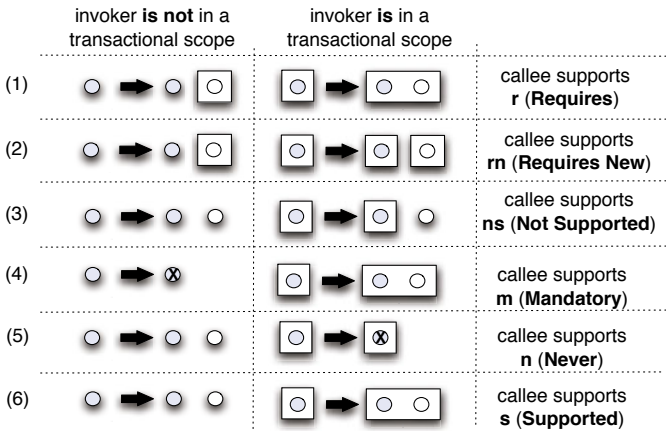


Fig. 1. EJB transactional attributes synopsis

dynamic reconfiguration of the transactional scope (e.g., whether a new scope has to be created, how the scope of the invoking party has to be extended, etc.). We formally model the behaviour illustrated in Figure 1 by embedding EJB attributes in a simple process calculus to give a general model for SOC². Hereafter, according to this interpretation, the terms *service provider* and *container* will be used interchangeably.

3 Attribute-Based Transaction Calculus (ATc)

The ATc calculus is built on top of two layers; *processes* (§ 3.1) and *systems* (§ 3.2). The former specify how communication takes place in presence of (nested) transactional scopes while the latter provide a formal framework for defining and invoking transactional services and the run-time reconfiguration of the transactional scopes.

3.1 ATc Processes

An ATc process is a CCS-like process with three additional capabilities: *service invocation*, *transactional scope*, and *compensation installation*. Let \mathcal{S} and \mathcal{N} be two countably infinite and disjoint sets of names for *service* and *channel*, respectively.

Definition 1. The set ATc processes \mathcal{P} is defined by following grammar:

$P, Q ::= 0$	<i>empty process</i>	$\pi ::= x$	<i>input</i>
$\nu x P$	<i>channel restriction</i>	\bar{x}	<i>output</i>
$P \mid Q$	<i>parallel</i>		
$!P$	<i>replication</i>		$A \subseteq \mathcal{A}$
$s \varepsilon A.P$	<i>service invocation</i>		s, s', \dots range over \mathcal{S}
$\langle P \mid \langle Q \rangle \rangle$	<i>transactional scope</i>		x, y, z, \dots range over \mathcal{N}
$\pi \llbracket Q \rrbracket . P$	<i>compensation installation</i>		u ranges over $\mathcal{S} \cup \mathcal{N}$

Restriction $\nu x P$ binds x in P and the sets of free and bound channels of $P \in \mathcal{P}$ are defined as usual and respectively denoted by $\text{fc}(P)$ and $\text{bc}(P)$. Finally, we assume $\pi = \bar{\pi}$.

The standard process algebraic syntax is adopted for idle process, restriction, parallel composition, and replication. Process $s \varepsilon A.P$ invokes a service s required to support a transactional attributes in $A \subseteq \mathcal{A}$; a transactional scope $\langle P \mid \langle Q \rangle \rangle$ consists of a running process P and a compensation Q (confined in the scope) executed only upon failure; $\pi \llbracket Q \rrbracket . P$ executes π and installs the compensation Q in the enclosing transactional scope then behaves as P . Service definition and invocation are dealt with in § 3.2.

Definition 2. The structural congruence $\equiv \subseteq \mathcal{P} \times \mathcal{P}$, is the smallest equivalence relation containing α -renaming, the monoidal axioms for \mid and 0 , and satisfying:

$$!P \mid P \equiv !P \quad \langle 0 \mid \langle Q \rangle \rangle \equiv 0 \equiv \langle 0 \rangle \quad \langle P \rangle \mid \langle Q \rangle \equiv \langle P \mid Q \rangle$$

$$\text{if } P \equiv Q \text{ then } \langle P \rangle \equiv \langle Q \rangle \text{ and } \langle P \rangle \equiv \langle Q \rangle$$

$$\nu x \langle P \rangle \equiv \langle \nu x P \rangle \quad \nu x \nu y P \equiv \nu y \nu x P \quad \nu x 0 \equiv 0 \quad \nu x (P \mid Q) \equiv (\nu x P) \mid Q, \text{ if } x \notin \text{fc}(Q)$$

Hereafter, $\pi.P$ stands for $\pi \llbracket 0 \rrbracket . P$ when $Q \equiv 0$ and trailing occurrences of 0 are omitted.

² We refer to the service-oriented paradigm in a technology-agnostic way, abstracting from its actual realisations (e.g., the Web Service Architecture).

In ATc, transactional scopes can be nested up to an arbitrary level. The fact that a process is inside a transactional scope does not alter its communication capabilities, since we assume that transactional scopes influence the behaviour of processes only in case of failure. To model the semantics of communications we use *contexts*³.

Definition 3. A context is a term generated by the following productions:

$$C[-] ::= - \mid 0 \mid \langle - \mid P \mid \langle Q \rangle \rangle \mid P \mid C[-] \mid C[-] \mid P$$

A context $C[-]$ is scope-avoiding if there are no $P, Q \in \mathcal{P}$ and context $C'[-]$ such that $C[-] = C'[\langle - \mid P \mid \langle Q \rangle \rangle]$.

Definition 3 does not consider $\nu x C[-]$ to avoid name capture while prefix contexts $\alpha.C[-]$ (where α is either of the prefixes of ATc) are ruled out as they prevent inner reductions. The semantics of ATc is defined by means of two reduction relations, one (Definition 4) for process communication and the other (Definition 6) for service invocations (and, correspondingly, reconfigurations of transactional scopes).

Definition 4. The reduction relation of ATc processes is the smallest relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ closed under the following axioms and rules:

$$C[\langle \pi[\langle Q \rangle].P \mid \langle R \rangle \rangle] \mid C'[\langle \bar{\pi}[\langle Q' \rangle].P' \mid \langle R' \rangle \rangle] \rightarrow C[\langle P \mid \langle R \mid Q \rangle \rangle] \mid C'[\langle P' \mid \langle R' \mid Q' \rangle \rangle]$$

$$C[\langle \pi[\langle Q \rangle].P \mid \langle R \rangle \rangle] \mid C'[\langle \bar{\pi}[\langle Q' \rangle].P' \rangle] \rightarrow C[\langle P \mid \langle R \mid Q \rangle \rangle] \mid C'[P'], \quad \text{if } C'[-] \text{ is scope-avoiding}$$

$$C[\pi[\langle Q \rangle].P] \mid C'[\bar{\pi}[\langle Q' \rangle].P'] \rightarrow C[P] \mid C'[P'], \quad \text{if } C[-] \text{ and } C'[-] \text{ are scope-avoiding}$$

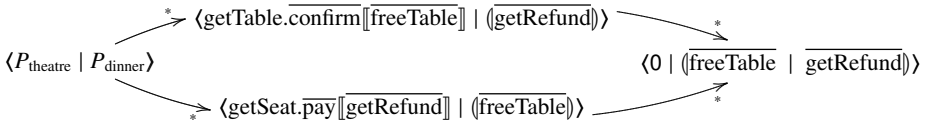
$$\frac{P \rightarrow P'}{P \mid R \rightarrow P' \mid R} \qquad \frac{P \rightarrow P'}{\nu x P \rightarrow \nu x P'} \qquad \frac{P \equiv P' \rightarrow Q' \equiv Q}{P \rightarrow Q}$$

Notice that sender and receiver synchronise regardless the relative nesting of transactional scopes. As in [12], when communication actions are executed compensations are installed in parallel to the other compensations of the enclosing transactional scope if any, otherwise they are discarded. In case of failure, only the actions executed before the failure are compensated, as illustrated by Example 1.

Example 1. Consider the transactional scope $P_{\text{bookNight}} = \langle P_{\text{theatre}} \mid P_{\text{dinner}} \rangle$ where:

$$P_{\text{theatre}} = \overline{\text{askSeat}}.\overline{\text{getSeat}}.\overline{\text{pay}}[\overline{\text{getRefund}}] \qquad P_{\text{dinner}} = \overline{\text{askTable}}.\overline{\text{getTable}}.\overline{\text{confirm}}[\overline{\text{freeTable}}]$$

Action $\overline{\text{getRefund}}$ compensates $\overline{\text{pay}}$ and action $\overline{\text{freeTable}}$ compensates $\overline{\text{confirm}}$. The process dynamically installs the compensations of its actions. The two executions show that different compensations may be executed in case of failure.



◇

³ Other and more standard techniques could have been used (e.g., LTS); however, contexts enable us to easily define the semantics of communication and service invocation of ATc.

3.2 ATc Systems

The semantics of transactional scoping of service invocations is given at the level of *systems* (Definition 5). Systems can be thought of as an abstraction for EJB and consist of processes wrapped by *containers* defined as a partial finite maps $\gamma : \mathcal{S} \rightarrow \mathcal{A} \times \mathcal{P}$; containers assign a transactional attribute and a process (the “body”) to service names. When defined, $\gamma(s) = (a, P)$ ensures that, if invoked in γ , the service s supports the attribute a and activates an end-point that executes as P .

Definition 5. A system in ATc is a pair $\Gamma \vdash P$ where the environment Γ is a set of containers and is derived by the productions in Definition 1 augmented with $P ::= \text{err}$ to represent erroneous processes. Also, the following axioms

$$\text{!err} \equiv \text{err} \quad \forall x \text{ err} \equiv \text{err} \quad \langle \text{err} \mid \langle Q \rangle \rangle \equiv \text{err}$$

extend the congruence relation to erroneous processes.

Given $A \subseteq \mathcal{A}$, $P \in \Gamma(s, A)$ shortens $\exists \gamma \in \Gamma \exists a \in A : \gamma(s) = (a, P)$ and $P \in \Gamma(s, \{a\})$ is abbreviated as $P \in \Gamma(s, a)$. Hereafter, we use P, Q to range over both \mathcal{P} and erroneous processes. We rule out terms where compensations contain err ; basically, err represent a run-time error and cannot be used by the programmer. A service invocation is *transactional* (resp. *non-transactional*) if it is (resp. not) executed a transaction scope.

Definition 6 formalises the informal presentation in Figure 1 of the CMT mechanisms which are rendered in SOC by allowing environments Γ to offer different implementations of the same service possibly with different attributes. This results in a non-deterministic semantics where one of several possible reductions is chosen.

Definition 6. The reduction relation of ATc systems is the smallest relation \rightsquigarrow closed under the following rule and axioms of Figure 2 where $C[_] \neq 0$ and $C[_]$ is scope avoiding in $(\text{ntx1} \div 3)$.

Axioms $(\text{ntx1} \div 3)$ rule non-transactional invocations; (ntx1) states that an invocation results in an error when a service supporting attribute m is required⁴; when a non-transactional invocation is made to a service supporting either s , or n , or ns , by (ntx2) , the end-point of the service is executed in parallel with the continuation of the caller; finally by (ntx3) , the end-point of a service supporting r or rn will be executed in a new scope (initially with idle compensation).

Axioms $(\text{tx1} \div 4)$ determine how transactional invocations modify the scope; by (tx1) , the end-point of the service is executed in the same scope of the caller when the requested attribute is m , s , or r ; instead by (tx2) , transactional invocations to a service supporting n yields a failure which triggers the compensation of the caller; by (tx3) a transactional invocation requesting ns will let the service end-point to run outside the

⁴ Axiom (ntx1) may seem odd as it introduces an error even if Γ may offer a service supporting other attributes in A . An actual implementation may in fact select more suitable services with an appropriate negotiation in the search phase. Here, more simply, we define the conditions to correctly use attributes avoiding errors in *any* possible environment; therefore (ntx1) models the worst case scenario. As shows in § 4, in well-typed processes, invocations requiring m never occur in scope-avoiding contexts.

(ntx1)	$\Gamma \vdash C[s \varepsilon A.P] \rightsquigarrow \Gamma \vdash C[\text{err}]$	$m \in A$
(ntx2)	$\Gamma \vdash C[s \varepsilon A.P] \rightsquigarrow \Gamma \vdash C[P] \mid R$	$R \in \Gamma(s, \{s, n, ns\} \cap A)$
(ntx3)	$\Gamma \vdash C[s \varepsilon A.P] \rightsquigarrow \Gamma \vdash C[P] \mid \langle R \rangle$	$R \in \Gamma(s, \{r, rn\} \cap A)$
(tx1)	$\frac{P = C[\langle s \varepsilon A.P_1 \mid P_2 \mid \langle Q \rangle \rangle] \quad \text{bc}(P) \cap \text{fc}(R) = \emptyset}{\Gamma \vdash P \rightsquigarrow \Gamma \vdash C[\langle P_1 \mid P_2 \mid R \mid \langle Q \rangle \rangle]}$	$R \in \Gamma(s, \{m, s, r\} \cap A)$
(tx2)	$\Gamma \vdash C[\langle s \varepsilon A.P_1 \mid P_2 \mid \langle Q \rangle \rangle] \rightsquigarrow \Gamma \vdash C[Q]$	$n \in A$
(tx3)	$\Gamma \vdash C[\langle s \varepsilon A.P_1 \mid P_2 \mid \langle Q \rangle \rangle] \rightsquigarrow \Gamma \vdash C[\langle P_1 \mid P_2 \mid \langle Q \rangle \rangle] \mid R$	$ns \in A \wedge R \in \Gamma(s, ns)$
(tx4)	$\Gamma \vdash C[\langle s \varepsilon A.P_1 \mid P_2 \mid \langle Q \rangle \rangle] \rightsquigarrow \Gamma \vdash C[\langle P_1 \mid P_2 \mid \langle Q \rangle \rangle] \mid \langle R \rangle$	$rn \in A \wedge R \in \Gamma(s, rn)$
(s-p)	$\frac{P \rightarrow P'}{\Gamma \vdash P \rightsquigarrow \Gamma \vdash P'}$	

Fig. 2. Semantics of ATc

caller's scope; finally, (tx4) states that a transactional invocation requesting rn will let the service end-point to run in a new scope with idle compensation.

Rule (s-p) lifts process reduction relation to systems.

Communication failures occurring within transactional scopes trigger compensations while those occurring outside result in an error. Formally, this can be achieved by adding to Definition 6 the axioms

$$C[\langle \pi \llbracket Q \rrbracket . P \mid \langle R \rangle \rangle] \rightarrow C[Q] \quad \text{and} \quad C[\pi \llbracket Q \rrbracket . P] \rightarrow C[\text{err}], \text{ if } C[_] \text{ is scope avoiding} \quad (2)$$

For simplicity, we gloss over this point in order to focus on failures due to misuse of transactional attributes and scope reconfigurations. We are currently working on a semantics of communication failures for ATc systems briefly outlined in § 6. The semantics of failures is based on the notion of testing equivalence [10] (see [5] for an extended report of this paper including further details).

3.3 Some Examples of Failing Invocations

The following examples motivate the need of a disciplined use of transactional attributes. The typing system presented in § 4 ensures that a well-typed process will incur in errors due to the fact that the attributes required by an invoker do not match those guaranteed by the service.

Example 2. Let $P_{\text{bookTheatre}} = \langle s_{\text{tickets}} \varepsilon \{m\}.P_{\text{theatre}} \mid \langle s_{\text{compensate}} \varepsilon \{m\} \rangle \rangle$ be a process that invokes s_{tickets} and behaves as $P_{\text{theatre}} = \overline{\text{askSeat}}.\text{getSeat}.\overline{\text{pay}}[\overline{\text{getRefund}}]$. If a communication of P_{theatre} fails (i.e., the left-most axiom in (2) is applied), then the compensation is executed outside a transactional scope. Therefore, the non-transactional invocation to $s_{\text{compensate}}$ will result in an error. \diamond

Example 3. Let P_{theatre} as in Example 2 and consider

$$P_{\text{bookTheatre}} = s_{\text{tickets}} \varepsilon \{m, s, n, ns, r, rn\}.P_{\text{theatre}} \quad P_{\text{tickets}} = \text{askSeats}.\overline{\text{getSeats}}.s_{\text{bank}} \varepsilon \{m\}$$

The non-transactional invocation s_{tickets} in a Γ for which $P_{\text{tickets}} \in \Gamma(s_{\text{tickets}}, s)$ causes P_{tickets} to run outside a transactional scope; hence, invoking s_{bank} leads to an error. \diamond

A provider must guarantee that none of its services yield errors; namely, the execution of (the body of) a service in any context resulting from its supported attributes should be safe. For instance, since s_{tickets} in Example 3 supports s , the execution P_{tickets} should be safe regardless it will run inside or outside a transactional scope. In fact, whether or not P_{tickets} will be running in a scope depends on the caller.

4 A Type System for Transactional Services

This section yields a type system for ATc that can determine if a system may fail for a service invocation due to misuse of the transactional attributes. We give an algebra of types (§ 4.1), then define a type system for ATc (§ 4.2), and finally we give a suitable notion of well-typedness for ATc systems (§ 4.3) which is preserved by the reduction relation (Theorem 1) and ensures error-freedom (Corollary 1). All the proofs are reported in [5].

4.1 Types for ATc

Our types record which transactional attributes may be required/supported in service invocations of processes. Basically, for each possible invocation, a type specifies if it is transactional or not and which transactional attributes are declared for the invocation.

Definition 7. Let $I \subseteq \{i, o\} \times \mathcal{A}$ where labels i and o are the transactional modalities used to keep track of transactional and non-transactional invocations, respectively. Types are defined as

$$(\text{Types}) \quad t ::= 0 \mid (I, t, t)$$

Let $P \triangleright t$ state that $P \in \mathcal{P}$ has type t . If $P \triangleright 0$ then P does not make any invocations; if $P \triangleright (I, t_c, t_u)$,

- I records the transactional modality/attribute pairs of the service invocations of P ;
- t_c collects the transactional modality/attribute pairs relative to the service invocations in the compensations of the transactional scopes of P ;
- t_u yields modality/attribute pairs for the invocations in the compensation installation prefixes⁵ of P ;

Example 4. Consider $P_2 = s \varepsilon A.y[[P_1]]$ with $P_1 \triangleright t_1$. As more clear in § 4.2, $P_2 \triangleright t_2 = (\{o\} \times A, 0, t_1)$. In fact, the invocation in P_2 is non-transactional and the third component of t_2 is t_1 as P_1 is used to compensate prefix y . \diamond

Types of processes become more complex in presence of nested scopes.

⁵ By Definition 6 compensations vanish for synchronisations outside transactional scopes.

Example 5. Take the process $P_3 = \langle P_2 \mid \langle [s' \varepsilon A'] \rangle \mid \langle \langle P_2 \mid \langle [s' \varepsilon A''] \rangle \rangle \rangle$, where P_2 is defined in Example 4. The type of P_3 is

$$t_3 = (\{i\} \times (A \cup A''), (\{0\} \times A', \mathbf{0}, \mathbf{0}), \mathbf{0})$$

In fact, the invocations in P_2 and in the nested compensation in the rightmost scope of P_3 will be transactional; therefore the first component of t_3 is $\{i\} \times (A \cup A'')$. Moreover, the leftmost scope of P_3 may possibly have a non-transactional invocation (thereby the second component of t_3). \diamond

The next example illustrates the installation of a non-trivial compensation.

Example 6. The type of $P_4 = \bar{z}[[s_1 \varepsilon A_1]].\langle \bar{z}[[s_1 \varepsilon A_1]] \mid \langle [s_2 \varepsilon A_2.z[[s_3 \varepsilon A_3]]] \rangle \rangle$ is

$$t_4 = (\mathbf{0}, (\{0\} \times (A_1 \cup A_2), \mathbf{0}, \{0\} \times A_3), \{0\} \times A_1)$$

In fact, P_4 does not invoke services but installs compensations that do so. Observe that the third component of t_4 corresponds to the first installation of P_4 , while the second component of t_4 is the type of the scope occurring in P_4 . \diamond

It is convenient to treat types as binary trees whose nodes are labelled with subsets of $\{i, 0\} \times \mathcal{A}$. More precisely, the type (I, t_c, t_u) can be represented as a tree where the root is labelled I , t_c is the left child, and t_u is the right child ($\mathbf{0}$ is the empty tree which is conventionally labelled with the empty set). The operators $_^\varepsilon$, $_^\downarrow$, and $_^\uparrow$ are used to “traverse” types and $_ \oplus _$ to “sum” them as per the following definitions:

$$\begin{aligned} \mathbf{0}^\varepsilon &= \emptyset, & \mathbf{0}^\downarrow &= \mathbf{0}^\uparrow = \mathbf{0} & (I, t_c, t_u)^\varepsilon &= I, & (I, t_c, t_u)^\downarrow &= t_c, & (I, t_c, t_u)^\uparrow &= t_u \\ \mathbf{0} \oplus t &= t, & (I, t_c, t_u) \oplus (I', t'_c, t'_u) &= (I \cup I', t_c \oplus t'_c, t_u \oplus t'_u) \end{aligned}$$

We assume that $_ \oplus _$ has lower precedence than unary operators.

Propositions 1 and 2 will be tacitly used in the proofs of the lemmas and Theorem 1.

Proposition 1. *The operator $_ \oplus _$ is idempotent, associative and commutative.*

Proposition 2. *Operators $_^\varepsilon$, $_^\downarrow$, and $_^\uparrow$ distribute over $_ \oplus _$ and $(t_1 \oplus t_2)^\varepsilon = t_1^\varepsilon \cup t_2^\varepsilon$.*

4.2 Typing ATc

This section introduces a typing system for ATc. We recall that the ATc programmer has to write non-erroneous processes for which we give the following typing rules.

Definition 8. *The typing rules for non-erroneous processes (cf. Definition 5) are*

$$\begin{aligned} (\text{idle}) \frac{}{0 \triangleright 0} \quad (\text{res}) \frac{P \triangleright t}{\nu x P \triangleright t} \quad (\text{par}) \frac{P \triangleright t \quad P' \triangleright t'}{P \mid P' \triangleright t \oplus t'} \quad (\text{repl}) \frac{P \triangleright t}{!P \triangleright t} \\ (\text{inv}) \frac{P \triangleright t_p \quad I = \{0\} \times A}{s \varepsilon A.P \triangleright (I \cup t_c^\varepsilon, t_p^\downarrow, t_p^\uparrow)} \quad (\text{comp}) \frac{P \triangleright t_p \quad Q \triangleright t_q}{\pi[[Q]].P \triangleright (t_p^\varepsilon, t_p^\downarrow, t_q \oplus t_p^\uparrow)} \\ (\text{scope}_1) \frac{P \triangleright (I, t_c, t_u) \quad Q \triangleright t_q}{\langle P \mid \langle Q \rangle \rangle \triangleright ((I \cup t_c^\varepsilon)[[0 \mapsto i]], t_u \oplus t_c^\downarrow \oplus t_c^\uparrow \oplus t_q, \mathbf{0})} \quad (\text{scope}_2) \frac{P \triangleright 0}{\langle P \mid \langle Q \rangle \rangle \triangleright 0} \end{aligned}$$

where, for $I \subset \{i, 0\} \times \mathcal{A}$, $I[[0 \mapsto i]] \stackrel{\text{def}}{=} \{(i, a) : (0, a) \in I\} \cup (I \cap \{i\} \times \mathcal{A})$.

The first five rules are straightforward. Rule (comp) states that the type of the installation of a compensation Q records the invocations in Q as possible invocations of P by adding them to the third component of the type of $\pi\llbracket Q \rrbracket.P$. The last two rules regulate the typing of transactional scopes. By rule (scope₁), when P is in a transactional scope the invocations done by the compensations installed by P (recorded in t_u) become possible; therefore they are removed from the third component and added to the second component with the compensations nested in P (recorded in t_c^\downarrow and $t_c^?$) and to those of Q (recorded in t_q). Also, t_c^ε records the invocation of the compensation of P when P is itself defined s a transactional scope (e.g., $P = \langle Q \mid \langle C \rangle \rangle$); in this case the compensations of P will be surely executed inside a transactional scope thus they are included in the first component with the substitution $\llbracket \circ \mapsto i \rrbracket$. A transactional scope whose process does not invoke/install anything is simply typed as $\mathbf{0}$ by rule (scope₂).

Example 7. Consider the process $P = \pi_1\llbracket Q \rrbracket$ where

$$Q = \pi_2\llbracket R \rrbracket \quad \text{and} \quad R = s_1 \varepsilon A_1.\pi_3\llbracket s_2 \varepsilon A_2 \rrbracket$$

The typing of P is $t = (\mathbf{0}, \mathbf{0}, (\mathbf{0}, \mathbf{0}, (I_1, \mathbf{0}, I_2)))$ as proved by the type inference below.

$$\begin{array}{c} \text{(inv)} \frac{I_2 = \{\circ\} \times A_2 \quad \mathbf{0} \triangleright \mathbf{0}}{s_2 \varepsilon A_2 \triangleright (I_2, \mathbf{0}, \mathbf{0}) \quad \mathbf{0} \triangleright \mathbf{0}} \text{(Comp)} \\ \text{(inv)} \frac{\pi_3\llbracket s_2 \varepsilon A_2 \rrbracket \triangleright (\mathbf{0}, \mathbf{0}, I_2) \quad I_1 = \{\circ\} \times A_1}{s_1 \varepsilon A_1.\pi_3\llbracket s_2 \varepsilon A_2 \rrbracket \triangleright (I_1, \mathbf{0}, I_2) \quad \mathbf{0} \triangleright \mathbf{0}} \text{(Comp)} \\ \frac{\pi_2\llbracket R \rrbracket \triangleright (\mathbf{0}, \mathbf{0}, (I_1, \mathbf{0}, I_2)) \quad \mathbf{0} \triangleright \mathbf{0}}{\pi_1\llbracket Q \rrbracket \triangleright (\mathbf{0}, \mathbf{0}, (\mathbf{0}, \mathbf{0}, (I_1, \mathbf{0}, I_2)))} \text{(Comp)} \quad \diamond \end{array}$$

Proposition 3. For each non-erroneous $P \in \mathcal{P}$ there is a unique type t such that $P \triangleright t$.

Proposition 4. For any non-erroneous $P, Q \in \mathcal{P}$, if $P \equiv Q$ and $P \triangleright t$ then $Q \triangleright t$.

Definition 9. Let t be a type. The flat type \widehat{t} of t is defined as follows:

$$\widehat{\mathbf{0}} = \emptyset \quad \widehat{t} = t^\varepsilon \cup \text{FLATTEN}(t^\downarrow), \text{ if } t \neq \mathbf{0}$$

$$\text{FLATTEN}(\mathbf{0}) = \emptyset \quad \text{FLATTEN}(t) = t^\varepsilon \cup \text{FLATTEN}(t^\downarrow) \cup \text{FLATTEN}(t^\uparrow), \text{ if } t \neq \mathbf{0}$$

Notice that $\widehat{t_1 \oplus t_2} = \widehat{t_1} \cup \widehat{t_2}$. In the interpretation of t as a tree, the flat type of t is the union of the set labelling all the nodes of t , excluding those of the subtree t^\uparrow which corresponds to dead code (cf. Example 8); in other words, either the typed process is outside a scope (in which case its pending compensations can be ignored) or the typed process is inside a scope (hence t^\uparrow is empty because of rule (Scope₁)).

4.3 Well-Typedness in ATc

The definition of well-typedness requires some care. In ATc, invocations to services can be statically typed as transactional or not. However, there is a different notion of well-typedness to adopt for services.

If P is not published as a service then it is possible to determine the nature of the service invocations of P by inspecting its code. Therefore, it suffices to specify, for each service invocation, the attributes for which no run-time errors are possible. This enables us to adopt the following definition.

Definition 10. Let $P \in \mathcal{P}$ such that $P \triangleright t$. The process P is well-typed iff $(v, m) \notin \widehat{t}$.

Example 8. Process P in Example 7 is (trivially) well-typed since $\widehat{t} = \emptyset$. In fact, the only service invocations of P are in the compensations to install (that are dead code since P is not included in any transactional scope). \diamond

Correctness depends on the (correctness of the) services invoked by a process. Remarkably, the fact that the invoked service is well-typed could be guaranteed by the service provider (as part of the service interface) and required as an obligation by the service requester in the service discovery phase. Namely, negotiation of transactional attributes should be part of the “contract” between requester and provider. The study of the mechanisms used to require/negotiate/certify transactional aspects of published services is out of the our scopes. However, our type system provides an effective framework to certify compatibility of transactional aspects between services and invokers.

Ensuring correctness for services is a bit more complex. Whether or not the invocations in the body of (the end-point of) a service, say s , are transactional depends on which attribute s supports and if the invocation to s happened from within or outside a transactional scope. Therefore, well-typedness of services takes into account both cases.

Definition 11. Let γ be a container and s be a service such that $\gamma(s) = (a, P)$ for some $a \in \mathcal{A}$ and $P \in \mathcal{P}$. Service s is well-typed in γ , if both (3) and (4) below hold.

$$\langle P \rangle \triangleright t \wedge a \in \{r, rn, m, s\} \implies (v, m) \notin \widehat{t} \quad (3)$$

$$P \triangleright t \wedge a \in \{s, n, ns\} \implies (v, m) \notin \widehat{t} \quad (4)$$

An environment Γ is well-typed iff all the services in the domain of any $\gamma \in \Gamma$ are well-typed.

We only consider the errors generated by the invocation of a service when attributes and transactional scopes mismatch. Errors due to other causes (e.g., failure of a communication channel) have been modelled in [5] by introducing *observers*, namely processes which can interfere in communications.

Example 9. Let the process P in Example 7 be the body of a service s supporting $s \in \mathcal{A}$. Both the well-typedness of P and of $\langle P \rangle$ must be checked. As argued in Example 8, P is well-typed while for $\langle P \rangle$ we just need to apply rule (Scope1) as follows:

$$\frac{\pi_1 \llbracket Q \rrbracket \triangleright (0, 0, (0, 0, (I_1, 0, I_2))) \quad 0 \triangleright 0}{\langle \pi_1 \llbracket Q \rrbracket \mid \langle 0 \rangle \rangle \triangleright (0, (0, 0, (I_1, 0, I_2)), 0)} \text{ (Scope1)}$$

Clearly, well-typedness of $\langle P \rangle$ depends on whether $(v, m) \in I_1 \cup I_2$ or not. \diamond

Theorem 1. *Let $P \in \mathcal{P}$ be well-typed. For every well-typed environment Γ , if $\Gamma \vdash P \rightsquigarrow \Gamma \vdash Q$ then Q is well-typed.*

A straightforward corollary of Theorem 1 is

Corollary 1. *If Γ and $P \in \mathcal{P}$ are well-typed and $\Gamma \vdash P \rightsquigarrow \Gamma \vdash Q$ then Q is a non-erroneous process.*

Our notion of well-typedness is stricter than necessary. In fact, a weaker notion can be adopted by taking a definition of flat type where the labels of some of the ‘right children’ of types are not considered. Though yielding less restrictive types, this would make the theory more complex, therefore we opted for simplicity rather than generality.

5 ATc Type System at Work

The type system in § 4 checks that any possible invocation to a service requires a safe set of attributes so to avoid errors due to misuse of transactional scopes and attributes.

The design of SOC transactions could be easier if we knew, for each service invocation in a process, the maximal set of attributes that satisfies the typing. As a matter of fact, specifying a larger set of attributes in a service invocation increases the chances of finding a suitable service supporting one of the attributes. The trade-off is however that a too large set of attributes may cause a run-time error due to a service instance running in a wrongly nested transactional scopes.

Arguably, non well-typed processes can be turned into well-typed ones by changing the attributes of some invocations. We show through an example a method for designing a well-typed process based on an alternative usage of the typing system in § 4. First, consider the types obtained as in Definition 7 but for set \mathcal{A} which is replaced by an infinite countable set \mathcal{E} of symbolic identifiers. A *symbolic* process corresponding to $P \in \mathcal{P}$ is a term $\mathit{sym}(P)$ obtained by replacing each set of attributes with a distinct formal identifier in \mathcal{E} meant to be substituted by a subset of \mathcal{A} .

Example 10. A symbolic process corresponding to $P_{\text{bookTheatre}}$ in Example 2 is

$$\mathit{sym}(P_{\text{bookTheatre}}) = \langle s_{\text{tickets}} \varepsilon X_1 \overline{\text{askSeat.getSeat.pay}}[\overline{\text{getRefund}}] \mid (s_{\text{compensate}} \varepsilon X_2) \rangle$$

(the sets of attributes in $P_{\text{bookTheatre}}$ are replaced by X_1 and X_2). \diamond

A *maximal process* is a well-typed process for which augmenting any of the sets of attributes of its invocations yields the same process or a non-well typed process. Given a well-typed $P \in \mathcal{P}$, $\max(P)$ is the maximal process corresponding to P . (Notice that if P does not make any invocation then $P = \max(P)$.)

The typing system of Definition 8 is adapted to symbolic processes by replacing rule (inv) with

$$(\text{invSym}) \frac{P \triangleright t_p \quad I = \{\emptyset\} \times X}{s \varepsilon X.P \triangleright (I \cup t_p \varepsilon, t_p^\downarrow, t_p^?)}, \text{ where } X \text{ not occurs in } P$$

Example 11. By straightforward application of the typing system for symbolic processes $\text{sym}(P_{\text{bookTheatre}}) \triangleright t_{\text{sym}}$, where $t_{\text{sym}} = (\{i\} \times X_1, (\{0\} \times X_2, 0, 0), 0, 0)$. Hence, the flattened type of $\text{sym}(P_{\text{bookTheatre}})$ is $\widehat{t}_{\text{sym}} = \{(i, X_1), (0, X_2)\}$. \diamond

Finally, the maximal process is obtained by replacing each formal identifies with a suitable set of attributes. For example,

$$\max(P_{\text{bookTheatre}}) = \langle s_{\text{tickets}} \varepsilon \overline{\mathcal{A}.\text{askSeat.getSeat.pay}}[\overline{\text{getRefund}}] \mid (s_{\text{compensate}} \varepsilon \mathcal{A} \setminus \{m\}) \rangle$$

is obtained by replacing X_1 with \mathcal{A} and X_2 with $\mathcal{A} \setminus \{m\}$ in $\text{sym}(P_{\text{bookTheatre}})$. In fact, the invocation to s_{tickets} (i.e., the one associated with X_1) can possibly contain all attributes since they are transactional while the other invocation (i.e., the one to $s_{\text{compensate}}$ associated with X_2) can contain all attributes except m .

In general, one is interested only in some policies for transactional scopes and will typically choose, for each invocation in a process P , a subset of the attributes of the corresponding invocation in $\max(P)$.

6 Concluding Remarks and Related Work

An original contribution of this paper is the definition of mechanisms to *determine* and *control* the dynamic reconfiguration of distributed transactions. Namely, we embed a few primitives for managing the dynamic reconfiguration of transactional scopes in ATc to generalise the transactional mechanisms of EJB to SOC so to have consistent and predictable failure propagation. We give a type system that guarantees absence of failures due to misuse of transactional attributes. Since both dynamic reconfiguration and LRT are a key aspects in SOC, it is crucial to provide a formal account of their interrelationships and to understand and control the mechanisms of failure. The aim of this paper is to address the lack of agreement on the semantics of dynamic reconfigurations of transactional scopes in SOC. In fact, service invocations cause systems reconfiguration as they may dynamically introduce new transactional scopes or rearrange the old ones. Such problem is amplified when services support and rely on different kinds of transactional behaviour.

Languages for service orchestration (e.g., WS-BPEL [16]) providing support for distributed transactions have been modelled extending some process calculi like those in [3,11,13,14] with primitives that allow a party to define the scopes, failure handlers, and compensation mechanisms (see [20] for an overview and a comparison of such approaches). StAC [8] and CJoin [6] are process calculi which model arbitrarily nested transactions and focus on the separation of process management with error/compensation. The latter offers a mechanism to merge different scopes but it is not offering the flexibility of the transactional attributes of ATc. At the best of our knowledge, none of the proposed framework has been given a type system as the one proposed here (a formal comparison of different approaches for compensations in flow composition languages can be found in [7]). The existing literature addresses only part of the dynamic aspects involved in error management. For example, [12] proposes a model for dynamic installations of compensation processes, however, dynamic reconfigurations of transactional scopes have not been considered.

We are currently extending ATc with a theory of testing [10] where observers can cause communication failures. The aim is to test the correctness of the system behaviour, including failure handling and compensations. On this basis it is possible to define a notion of equivalence for ATc systems. The intuition is that two systems are equivalent if they satisfy the same set of tests; some preliminary results are summarised below (the interested reader is referred to [5] for a detailed presentation). The theory of testing of ATc shows that under some conditions some transactional attributes are equivalent. Namely, it is possible to replace a transactional attribute with an equivalent one without altering the behaviour of the system. Notice that this also allows one to specify a larger set of transactional attributes for service invocations. For example, $\langle s \varepsilon A.P \mid \langle Q \rangle \rangle$ maintain the same behaviour if A is any of the subsets of $\{r, m, s\}$ since the invocation of s happens inside a transaction.

A limitation of our approach is the lack of link mobility à la π -calculus; extending ATc with name passing is left as future work. We argue that the type discipline proposed here can be simply adapted to a name passing version of ATc. In fact, our type system is orthogonal to the communication mechanisms. On the contrary, the testing theory of ATc will be greatly affected by the introduction of name passing features. Allowing attributes to be communicated is another interesting extension of ATc also, a primitive enabling a service s to make a parametrised invocation to a service s' using the same attribute supported by s (attributes are set when services are published in containers). Such extensions increase expressiveness but require more sophisticated type disciplines.

An orthogonal topic is the modelling of protocols for deciding the outcome of distributed transactions (e.g., the work in [1]). Some standards like Business Transaction Protocol (BTP) [15] and Web Service Transaction (WS-Tx [17]) have been proposed for LRTs. Such protocols involve a more general scenario than the classic *atomic commit*: the global consensus is no longer necessary and is substituted by weaker constraints. In [2,4] BTP cohesion along with the properties ensured by the “weakened” constraints have been studied via a formalisation in the asynchronous π -calculus (see [9] for an overview on the *cohesion*-base approach of BTP). The present paper provides a high level semantics of failure propagation, compensation and scope reconfiguration, while abstracting from protocols necessary to implement them. Consider, for example, the process $\langle s \varepsilon \{r\}.P \mid \langle Q \rangle \rangle$ invoking a service s whose body is $x[[P'], Q']$. Since service s supports the attribute r , its body is executed inside the same scope (if any) of the caller, according to Definition 6.

$$\Gamma \vdash \langle s \varepsilon \{r\}.P \mid \langle Q \rangle \rangle \rightsquigarrow^* \Gamma \vdash \langle P \mid P' \mid \langle Q \mid Q' \rangle \rangle$$

The same above includes compensations of different possibly cross-domain and distributed processes. Noteworthy, the mechanism that trigger Q and Q' are not trivial. The higher level perspective we adopted has the advantage of providing a concise but rigorous understanding of dynamic scope reconfigurations. We leave the investigation of the underneath coordination protocols, which would provide a skeleton for the implementation of the higher level mechanisms, as a future work. (We remark that this issue is common to any theory of distributed transactions.)

References

1. Berger, M., Honda, K.: The two-phase commitment protocol in an extended pi-calculus. *Electr. Notes Theor. Comput. Sci.* 39(1) (2000)
2. Bocchi, L.: Compositional nested long running transactions. In: Wermelinger, M., Margaria-Steffen, T. (eds.) *FASE 2004*. LNCS, vol. 2984, pp. 194–208. Springer, Heidelberg (2004)
3. Bocchi, L., Laneve, C., Zavattaro, G.: A calculus for long-running transactions. In: Najm, E., Nestmann, U., Stevens, P. (eds.) *FMOODS 2003*. LNCS, vol. 2884, pp. 124–138. Springer, Heidelberg (2003)
4. Bocchi, L., Lucchi, R.: Atomic commit and negotiation in service oriented computing. In: Ciancarini, P., Wiklicky, H. (eds.) *COORDINATION 2006*. LNCS, vol. 4038, pp. 16–27. Springer, Heidelberg (2006)
5. Bocchi, L., Tuosto, E.: A Java Inspired Semantics for Transactions in SOC, extended report (2009), <http://www.cs.le.ac.uk/people/lb148/javatransactions.html>
6. Bruni, R., Melgratti, H.C., Montanari, U.: Nested commits for mobile calculi: extending Join. In: Lévy, J.-J., Mayr, E., Mitchell, J. (eds.) *IFIP TCS 2004*, pp. 563–576. Kluwer, Dordrecht (2004)
7. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In: *POPL*, pp. 209–220. ACM, New York (2005)
8. Butler, M., Ferreira, C.: An operational semantics for StAC, a language for modelling long-running business transactions. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) *COORDINATION 2004*. LNCS, vol. 2949, pp. 87–104. Springer, Heidelberg (2004)
9. Dalal, S., Temel, S., Little, M., Potts, M., Webber, J.: Coordinating business transactions on the web. *IEEE Internet Computing* 7(1), 30–39 (2003)
10. De Nicola, R., Hennessy, M.C.B.: Testing equivalences for processes. *Theoretical Comput. Sci.* 34(1-2), 83–133 (1984)
11. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the interplay between fault handling and request-response service invocations. In: *ACSD*, pp. 190–198. IEEE, Los Alamitos (2008)
12. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. *Fundam. Inf.* 95(1), 73–102 (2009)
13. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) *FOSSACS 2005*. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
14. Mazzara, M., Lanese, I.: Towards a unifying theory for web services composition. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) *WS-FM 2006*. LNCS, vol. 4184, pp. 257–272. Springer, Heidelberg (2006)
15. Business Transaction Protocol (BTP) (2002)
16. Web Services Business Process Execution Language (WS-BPEL). Technical report (2007)
17. Web Services Transaction (WS-TX) (2009)
18. Panda, D., Rahman, R., Lane, D.: *EJB 3 in action*. Manning (2007)
19. Sun Microsystems. Enterprise JavaBeans (EJB) technology (2009), <http://java.sun.com/products/ejb/>.
20. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Kaklamanis, C., Nielson, F. (eds.) *TGC 2008*. LNCS, vol. 5474, pp. 201–215. Springer, Heidelberg (2008)