Testing Attribute-Based Transactions in SOC

Laura Bocchi and Emilio Tuosto

Department of Computer Science, University of Leicester, UK

Abstract. We set the basis for a theory of testing for distributed transactions in service oriented systems where each service definition is decorated with a *transactional attribute* (inspired by the Java Transaction API). Transaction attributes discipline how services are executed with respect to the transactional scope of the invoking party.
We define a language of observers and show that, in general, the choice of different transactional attributes causes different system's behaviours wrt the testing equivalences induced by the observers.

1 Introduction

We give an observational theory for transactional behaviours in *Service-Oriented Computing* (SOC) based on the theory of testing [4]. Transaction in SOC, often referred to as *long-running*, feature a mechanism called *compensation* which is a weaker version[1] of the classic rollback mechanism of ACID transactions in database systems. In SOC, each activity of a transactional computation can be associated with a compensation *installed* as the activity is executed. The run-time failure of an activity is backwardly propagated and triggers the execution of the compensations installed for the activities completed earlier. Therefore, compensations have been studied in relation to mechanisms of failure propagation.

Notably, the key characteristics of SOC are loose-coupling and dynamism: services can be discovered at run-time relying only on their published interface, and upon service invocation the system dynamically reconfigures to include the newly created service instance. System reconfigurations should also consider transactional scopes (or scopes for short) as they play a fundamental role in failures propagation.

Consider the system $\langle \texttt{invoke}(s).P \rangle$ where the transaction, represented by the angled brackets, includes a process that invokes a service, which is described by the interface s, and then behaves like P. Suppose that there exists a provider that implements s as process Q. Should the system evolve so to include Q in the scope of the invoking process (i.e., $\langle P \mid Q \rangle$)? Should Q be running in a fresh scope (i.e., $\langle P \rangle \mid \langle Q \rangle$)? Or else, should Q be outside any scope (i.e., $\langle P \rangle \mid Q$)? Each alternative is valid and influences failure propagation and the behaviour of the system (as shown in § 4).

We design an observational theory that yields a formal framework for analysing the interplay between communication failures and the behaviour of a service-oriented system. We use *may*- and *must*-testing equivalences to compare transactional behaviours.

[1] ACID transactions are implemented by locking resources. Locks can be unfeasible if transactions are long lasting.

invoker **outside** a scope	invoker **inside** a scope	callee supports
(1) $\bullet \Longrightarrow \bullet\,\boxed{\circ}$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet\,\circ}$	`r` (**Requires**)
(2) $\bullet \Longrightarrow \bullet\,\boxed{\circ}$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet}\,\boxed{\circ}$	`rn` (**Requires New**)
(3) $\bullet \Longrightarrow \bullet\,\circ$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet}\,\circ$	`ns` (**Not Supported**)
(4) $\bullet \Longrightarrow \otimes$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet\,\circ}$	`m` (**Mandatory**)
(5) $\bullet \Longrightarrow \bullet\,\circ$	$\boxed{\bullet} \Longrightarrow \boxed{\otimes}$	`n` (**Never**)
(6) $\bullet \Longrightarrow \bullet\,\circ$	$\boxed{\bullet} \Longrightarrow \boxed{\bullet\,\circ}$	`s` (**Supported**)

Table 1. Informal semantics of EJB attributes. Boxes represent scopes, $\bullet$ represent callers, $\circ$ represent callees. Failed activities are denoted by $\otimes$. Each row shows the behaviour of one attribute; the first two columns show, respectively, invocations from outside and from within a scope.

A remarkable feature of our framework is that it allows to discipline the reconfiguration of transactional scopes, hence to predict and control the effects of failures in the reconfigured system.

We build up on ATc (after *Attribute-based Transactional calculus*) [1], a CCS-like process calculus designed to model dynamic SOC transactions featuring EJB transactional attributes [7, 6]; ATc and EJB attributes are summarised in § 2. § 3 yields the main contribution of the paper, namely the definition of a class of observers which induces suitable testing equivalences to compare ATc systems as shown in § 4.

2 Background

The ATc calculus presented in [1] takes inspiration from the *Container Managed Transactions* (CMT) mechanism of Enterprise Java Beans (EJB). Hereafter, the terms *container* and *service provider* which refer to the environment where methods and services are executed, will be used interchangeably.

An ATc *container* associates each service interface to a *transactional attribute* (attribute, for short) which specifies (*i*) the 'reaction' of the system upon invocations (e.g., "*calling the service from outside a scope throws an exception*"), and (*ii*) how scopes dynamically reconfigure (e.g., *"the invoked service is always executed in a newly created scope*"). On the other hand, also the invoking party can specify which attribute must be supported by the invoked service. This is natural in SOC where, typically, the service properties are mutually negotiated between requester and provider.

The set of attributes is

$$\mathcal{A} \stackrel{def}{=} \{\mathtt{m}, \mathtt{s}, \mathtt{n}, \mathtt{ns}, \mathtt{r}, \mathtt{rn}\} \qquad (\textit{attributes})$$

The intuitive semantics of each $a \in \mathcal{A}$ (attributes range over $a, a_1, a_2, \ldots$) is in Table 1. An ATc process is a CCS-like process with three additional capabilities: *service invocations*, *transactional scoping*, and *compensation installation*. The set $\mathcal{P}$ of ATc *processes* is given by the following grammar

$$P, Q ::= \mathsf{0} \mid \nu x\, P \mid P \mid Q \mid !P \mid s \triangleright A.P \mid \langle P \rangle_Q \mid \pi \downarrow Q.P \mid \mathtt{err} \qquad (\textit{processes})$$

where $s, s', \ldots$ range over a set of *service* names $\mathcal{S}$ while $x, y, z, \ldots$ range over a *channel* names $\mathcal{N}$ (assumed to be both countably infinite and disjoint), u ranges over $\mathcal{S} \cup \mathcal{N}$, and π is either x or $\overline{x}$. We assume $x = \overline{\overline{x}}$. Restriction $\nu x\, P$ binds x in P; we denote the sets of *free* and *bound* channels of $P \in \mathcal{P}$ by $\mathsf{fc}(P)$ and $\mathsf{bc}(P)$. The standard process algebraic syntax is adopted for idle process, restriction, parallel composition, and replication. Process $s \,\&\vdash\, A.P$ invokes a service s required to support one of the attributes in $A \subseteq \mathcal{A}$; a scope $\langle P \rangle_Q$ consists of a running process P and a compensation Q (confined in the scope) to be executed only upon failure (scopes can be nested); $\pi \downarrow Q.P$ executes π and installs the compensation Q in the enclosing scope (if any), then behaves as P; finally, `err` represents a run-time failure (`err` cannot be used by programmers).

A *system*

$$\Gamma \vdash P \quad \text{with } \Gamma = \{\gamma_1, \ldots, \gamma_n\} \quad (\textit{systems}) \qquad\qquad \gamma : \mathcal{S} \to \mathcal{A} \times \mathcal{P} \quad (\textit{containers})$$

is a process P within an *environment* Γ, namely within a set of *containers*. A container is a finite partial map that assign an attribute and a "body" to service names. When defined, $\gamma(s) = (a, P)$ ensures that, if invoked in γ, s supports the attribute a and activates an end-point that executes as P. Environments may offer different implementations of s and support different attributes. Henceforth we write $P \in \Gamma(s, A)$ for $\exists \gamma \in \Gamma\ \exists a \in A\ :$ $\gamma(s) = (a, P)$ and $P \in \Gamma(s, a)$ for $P \in \Gamma(s, \{a\})$.

The semantics of communications is given in terms of *contexts*; $\mathbf{C}[\natural]$ is *scope-avoiding* (s-a, for short) if there are no $P, Q \in \mathcal{P}$ and $\mathbf{C}'[\natural]$ s.t. $\mathbf{C}[\natural] = \mathbf{C}'[\langle \natural \mid P \rangle_Q]$.

$$\mathbf{C}[\natural] ::= \natural \;\big|\; \langle \mathbf{C}[\natural] \mid P \rangle_Q \;\big|\; P \mid \mathbf{C}[\natural] \;\big|\; \mathbf{C}[\natural] \mid P \qquad\qquad (\textit{contexts})$$

The *reduction relation of ATc processes* (i.e., $\rightarrow$) is the smallest relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ closed under the following axioms and rules:

$$\mathbf{C}[\langle \pi \downarrow Q.P \rangle_R] \mid \mathbf{C}'[\langle \bar{\pi} \downarrow Q'.P' \rangle_{R'}] \;\rightarrow\; \mathbf{C}[\langle P \rangle_{R|Q}] \mid \mathbf{C}'[\langle P' \rangle_{R'|Q'}] \qquad \texttt{(p1)}$$

$$\mathbf{C}[\langle \pi \downarrow Q.P \rangle_R] \mid \mathbf{C}'[\bar{\pi} \downarrow Q'.P'] \;\rightarrow\; \mathbf{C}[\langle P \rangle_{R|Q}] \mid \mathbf{C}'[P'], \qquad \text{if } \mathbf{C}'[\natural] \text{ is s-a} \qquad \texttt{(p2)}$$

$$\mathbf{C}[\pi \downarrow Q.P] \mid \mathbf{C}'[\bar{\pi} \downarrow Q'.P'] \;\rightarrow\; \mathbf{C}[P] \mid \mathbf{C}'[P'], \qquad \text{if } \mathbf{C}[\natural] \text{ and } \mathbf{C}'[\natural] \text{ are s-a} \qquad \texttt{(p3)}$$

$$\frac{P \rightarrow P'}{P \mid R \rightarrow P' \mid R} \qquad \frac{P \rightarrow P'}{\nu x\, P \rightarrow \nu x\, P'} \qquad \frac{P \equiv P' \rightarrow Q' \equiv Q}{P \rightarrow Q} \qquad \texttt{(p4÷ p6)}$$

The $\rightarrow$ relation is defined up-to a standard structural congruence relation $\equiv$ (which is extended to contexts). In `(p1÷ p3)`, sender and receiver synchronise regardless the relative nesting of their scopes. Upon synchronisation, compensations are installed in parallel to the other compensations of the enclosing scope; if $\mathbf{C}[\natural]$ is s.a. then compensations are discarded.

The *reduction relation of ATc systems* (i.e., $\rightsquigarrow$) is defined below, assuming $\mathbf{C}[\boxminus] \neq 0$.

$$\frac{P \to P'}{\Gamma \vdash P \rightsquigarrow \Gamma \vdash P'} \qquad \frac{\mathtt{m} \in A \qquad \mathbf{C}[\boxminus] \text{ is s-a}}{\Gamma \vdash \mathbf{C}[s \,\&\!\!\leftarrow A.P] \rightsquigarrow \Gamma \vdash \mathbf{C}[\mathtt{err}]} \tag{s1/s2}$$

$$\frac{R \in \Gamma(s, \{\mathtt{s}, \mathtt{n}, \mathtt{ns}\} \cap A) \qquad \mathbf{C}[\boxminus] \text{ is s-a}}{\Gamma \vdash \mathbf{C}[s \,\&\!\!\leftarrow A.P] \rightsquigarrow \Gamma \vdash \mathbf{C}[P] \mid R} \quad \frac{R \in \Gamma(s, \{\mathtt{r}, \mathtt{rn}\} \cap A) \qquad \mathbf{C}[\boxminus] \text{ is s-a}}{\Gamma \vdash \mathbf{C}[s \,\&\!\!\leftarrow A.P] \rightsquigarrow \Gamma \vdash \mathbf{C}[P] \mid \langle R \rangle} \tag{s3/s4}$$

$$\frac{P = \mathbf{C}[\langle s \,\&\!\!\leftarrow A.P_1 \mid P_2 \rangle_Q] \qquad \mathsf{bc}(P) \cap \mathsf{fc}(R) = \emptyset \quad R \in \Gamma(s, \{\mathtt{m}, \mathtt{s}, \mathtt{r}\} \cap A)}{\Gamma \vdash P \rightsquigarrow \Gamma \vdash \mathbf{C}[\langle P_1 \mid P_2 \mid R \rangle_Q]} \tag{s5}$$

$$\frac{\mathtt{n} \in A}{\Gamma \vdash \mathbf{C}[\langle s \,\&\!\!\leftarrow A.P_1 \mid P_2 \rangle_Q] \rightsquigarrow \Gamma \vdash \mathbf{C}[Q]} \tag{s6}$$

$$\frac{\mathtt{rn} \in A \wedge R \in \Gamma(s, \mathtt{rn})}{\Gamma \vdash \mathbf{C}[\langle s \,\&\!\!\leftarrow A.P_1 \mid P_2 \rangle_Q] \rightsquigarrow \Gamma \vdash \mathbf{C}[\langle P_1 \mid P_2 \rangle_Q] \mid \langle R \rangle} \tag{s7}$$

The rules above correspond to the informal presentation in Table 1: `(s2÷ s4)` model the first column and `(s5÷ s7)` model the second one. Failures trigger the compensation when occurring inside a scope `(s6)` and lead to an error otherwise `(s2)`.

ATc systems do not model communication failures[2] and do not provide an explicit notion of commit for transactions. These aspects are modelled in § 3.

3 Observers for ATc

In this section we provide a theory of testing by defining a notion of *observers* suitable for ATcthat interact with systems and possibly cause communication failures. Two systems are equivalent if they cannot be distinguished by an observers (they "pass the same tests").

In § 3.1 we define observers and observed systems, in § 3.2. we give an observational semantics of ATc, in § 4 we show some motivating examples.

3.1 Observed Systems

The class of observers defined in this section is used to model communication failures and define successful computations. An *observer* is derived by the following grammar:

$$O ::= 0 \;\big|\; \checkmark \;\big|\; \pi.O \;\big|\; \lightning\pi.O \;\big|\; O + O \;\big|\; \mathbf{rec}\, X.O \;\big|\; X \qquad (\textit{observers})$$

The *structural congruence for observers* is the smallest equivalence relation closed under the monoidal axioms of + and it is denoted as $\equiv_o$.

We consider sequential observers. Failing and successful tests are represented by 0 and $\checkmark$, respectively; prefix $\pi.O$ allows observers to communicate with the system, while prefix $\lightning\pi.O$ causes the failure of π in the system and continues as O; observers can be composed with the (external) choice operator + and recursively defined as **rec** $X.O$ (where the occurrences of X in O are supposed guarded by prefixes). An observer is a process that can interact with a system over its (free) channels and trigger failures in

[2] The relation $\rightsquigarrow$ only considers errors due to misuse of attributes.

the communications (e.g., to check that failures are correctly handled). Since observers cannot be composed in parallel, they do not communicate among themselves. This, and the absence of name passing in ATc, allow us to avoid using name restriction in observers. Moreover, observers do not run in transactional scopes and they are not allowed to invoke services; they are used to model communication failures so to scrutinize the transactional behaviour of ATc systems.

Let systems be ranged over by $S, S', \ldots$; the set States of *observed systems* is the set of pairs made of a system S and an observer O, written as $S \parallel O$.

The *reduction relation of ATc observed systems* (i.e., $\rightsquigarrow$) is the smallest relation satisfying the following axioms (where $\mathbf{C}[\natural]$ is s-a in `(os1/os2)`):

$$\Gamma \vdash \mathbf{C}[\pi \downarrow Q.P] \parallel \overline{\pi}.O \rightsquigarrow \Gamma \vdash \mathbf{C}[P] \parallel O \qquad \Gamma \vdash \mathbf{C}[\langle \pi \downarrow Q.P \rangle_R] \parallel \overline{\pi}.O \rightsquigarrow \Gamma \vdash \mathbf{C}[\langle P \rangle_{Q|R}] \parallel O \quad \texttt{(os1/os2)}$$

$$\Gamma \vdash \mathbf{C}[\pi.P] \parallel \lightning \pi.O \rightsquigarrow \Gamma \vdash \mathbf{C}[\texttt{err}] \parallel O \qquad \Gamma \vdash \mathbf{C}[\langle \pi.P \mid R \rangle_Q] \parallel \lightning \pi.O \rightsquigarrow \Gamma \vdash \mathbf{C}[Q] \parallel O \quad \texttt{(os3/os4)}$$

$$S \parallel \checkmark \rightsquigarrow S \parallel \checkmark \qquad \frac{O \equiv_o O_1 \quad S \parallel O_1 \rightsquigarrow S' \parallel O_2 \quad O_2 \equiv_o O'}{S \parallel O \rightsquigarrow S' \parallel O'} \quad \texttt{(os5/os6)}$$

$$\frac{S \rightsquigarrow S'}{S \parallel O \rightsquigarrow S' \parallel O} \qquad \frac{S \parallel O \rightsquigarrow S' \parallel O'}{S \parallel O + O'' \rightsquigarrow S' \parallel O'} \quad \texttt{(os7/os8)}$$

Rules `(os1/os2)` model a communication step involving the system and the observer. Communication failures occurring outside a scope yield an error `(os3)`; failures occurring inside a scope trigger the compensations associated with the enclosing scope `(os4)`. Rule `(os5)` signals when a test is passed, and `(os6)` is the usual rule for congruence. Rule `(os7)` models a step due to transitions of the system that do not involve the observer. The interactions of the system with non-deterministic observers are defined by rule `(os8)`; notice that, by `(os5)`, if $O = \checkmark$ and $O' = \mathsf{0}$, then O'' is discarded.

Example 1. Consider a scenario where process P acts as a proxy of a shared resource for a client (which are not explicitly represented):

$$R = \overline{\text{lock}} \downarrow \overline{\text{unlock}}.(\text{quit}.\overline{\text{unlock}}).$$

R interacts with the resource to acquire a lock. This action is associated to compensation $\overline{\text{unlock}}$ whose aim is to release the resource if an error interrupts the normal execution flow. The client is granted to use of the resource until she sends message quit. Finally the resource is released ($\overline{\text{unlock}}$). Consider the observer

$$O = \text{lock}.(\lightning\overline{\text{quit}}.\text{unlock}.\checkmark)$$

that checks if the resource, after having been acquired, is released in case of failure of the clients' request to quit (action $\lightning\overline{\text{quit}}$). Notably, for any Γ, the observed system $\Gamma \vdash R \parallel O$ does not pass the test since the compensation is discarded by rule `(os1)` and O never reaches state $\checkmark$. Observed system $\Gamma \vdash \langle R \rangle \parallel O$ instead is satisfactory since the compensation, installed by `(os2)`, can release the resource. $\diamond$

The set Comp of *computations* (ranged over by c) is the set of (possibly infinite) sequences of states $S_0 \parallel O_0, \cdots, S_n \parallel O_n, \cdots$ such that $S_i \parallel O_i \rightsquigarrow S_{i+1} \parallel O_{i+1}$ for each i.

3.2 Testing Equivalences for ATc

The basic elements of the testing theory are the notions of *successful* and *non-divergent* computation. Intuitively, a computation is successful if the test is passed (i.e., the corresponding observer halts with ✓). Non-divergent computations are successful computations that reach ✓ before the occurrence of an error. We now cast the basic notions of the testing theory to ATc observed systems.

Definition 1. *Let* $O \searrow \checkmark$ *stand for* $O = \checkmark + O'$ *for some observer* O'.

- $S \parallel O \in \mathsf{States}$ *is* successful *if* $O \searrow \checkmark$*;*
- $\Gamma \vdash P \parallel O \in \mathsf{States}$ *is* diverging *if* $P = \mathbf{C}[\mathtt{err}]$ *for a context* $\mathbf{C}[\square]$*;*
- $c \in \mathsf{Comp}$ *is* successful *if it contains a successful state,* unsuccessful *otherwise;*
- $c = S_0 \parallel O_0, S_1 \parallel O_1, \ldots, S_n \parallel O_n, \ldots$ diverges *if either* c *is unsuccessful or there is* $i \geq 0$ *such that* $S_i \parallel O_i$ *is diverging and* $O_j \not\searrow \checkmark$ *for* $j < i$.

As customary in testing theory, the possible outcomes of computations are defined in terms of *result sets*, namely (non-empty) subsets of $\{\top, \bot\}$ where $\bot$ and $\top$ denote divergence and non-divergence, respectively.

Definition 2. *The* result set of $S \parallel O \in \mathsf{States}$, $\mathfrak{R}(S \parallel O) \subseteq \{\top, \bot\}$*, is defined by*

- $\top \in \mathfrak{R}(S \parallel O) \iff$ *there is a successful* $c \in \mathsf{Comp}$ *that starts from* $S \parallel O$*,*
- $\bot \in \mathfrak{R}(S \parallel O) \iff$ *there is* $c \in \mathsf{Comp}$ *starting from* $S \parallel O$ *such that* c *is diverging.*

As in [4], we consider *may*- and *must*-preorders and the corresponding induced equivalences.

Definition 3. *Given a system* S *and an observer* O*, we say that*

$$S \ \textsc{may}\ O \iff \top \in \mathfrak{R}(S \parallel O) \qquad \textit{and} \qquad S \ \textsc{must}\ O \iff \{\top\} = \mathfrak{R}(S \parallel O)$$

We define the preorders $\sqsubseteq_{\mathbf{m}}$ *(may preorder) and* $\sqsubseteq_{\mathbf{M}}$ *(must preorder) on systems:*

- $S \sqsubseteq_{\mathbf{m}} S' \iff (S \ \textsc{may}\ O \implies S' \ \textsc{may}\ O)$*, for all observers*
- $S \sqsubseteq_{\mathbf{M}} S' \iff (S \ \textsc{must}\ O \implies S' \ \textsc{must}\ O)$*, for all observers.*

The two equivalences $\simeq_{\mathbf{m}}$ *and* $\simeq_{\mathbf{M}}$ *corresponding to* $\sqsubseteq_{\mathbf{m}}$ *and* $\sqsubseteq_{\mathbf{M}}$ *are defined as expected:* $\simeq_{\mathbf{m}} = \sqsubseteq_{\mathbf{m}} \cap \sqsubseteq_{\mathbf{m}}^{-1}$ *and* $\simeq_{\mathbf{M}} = \sqsubseteq_{\mathbf{M}} \cap \sqsubseteq_{\mathbf{M}}^{-1}$.

Recall that (*i*) may-testing enforces some *fairness* ensuring that divergence is not "catastrophic" provided that there is a chance of success and (*ii*) that must-testing corresponds to liveness as it requires all possible computations to be successful.

4 Testing Theory for ATc at Work

The following examples show how attributes influence the reconfiguration of transactional scopes and how this is captured by our testing framework.

Example 2. Consider the service s with body R defined in Example 1. Let Γ be an environment such that $R \in \Gamma(s, \mathtt{r})$ and $R \in \Gamma(s, \mathtt{rn})$, namely in Γ there are (at least) two providers for s with the same body R but supporting different attributes. Consider the two possible clients, both invoking s and then releasing the resource:

$$P_1 = \langle s \,\overline{\&}\, \{\mathtt{r}\}.\overline{\mathrm{quit}}\rangle \qquad \text{and} \qquad P_2 = \langle s \,\overline{\&}\, \{\mathtt{rn}\}.\overline{\mathrm{quit}}\rangle.$$

The different attributes associated to s generate two different behaviours from P_1 and P_2 upon invocation (i.e., activation of endpoint $R = \overline{\mathrm{lock}} \downarrow \overline{\mathrm{unlock}}.\mathrm{quit}.\overline{\mathrm{unlock}}$):

$$\begin{aligned} S_1 = \Gamma \vdash \langle \overline{\mathrm{quit}} \mid \overline{\mathrm{lock}} \downarrow \overline{\mathrm{unlock}}.\mathrm{quit}.\overline{\mathrm{unlock}}\rangle \qquad & \text{by rule } (\mathtt{s5}) \\ S_2 = \Gamma \vdash \langle \overline{\mathrm{quit}}\rangle \mid \langle \overline{\mathrm{lock}} \downarrow \overline{\mathrm{unlock}}.\mathrm{quit}.\overline{\mathrm{unlock}}\rangle \qquad & \text{by rule } (\mathtt{s7}). \end{aligned}$$

Remarkably, R runs in the same transactional scope of the invoker in S_1 (due to the attribute $\mathtt{r}$), while it runs in a different scope in S_2 (due to the attribute $\mathtt{rn}$). Now take observer $O = \mathrm{lock}.\mathrm{unlock}.\checkmark + \lightning\overline{\mathrm{quit}}.\mathrm{unlock}.\checkmark$ that checks that the resource is unlocked both in case of normal execution and failure.

Running S_1 in parallel with O, and S_2 in parallel with O would results, after the synchronisation on channel lock, respectively in the system

$$S_1' = \Gamma \vdash \langle \overline{\mathrm{quit}} \mid \mathrm{quit}.\overline{\mathrm{unlock}}\rangle_{\overline{\mathrm{unlock}}} \qquad \text{and} \qquad S_2' = \Gamma \vdash \langle \overline{\mathrm{quit}}\rangle \mid \langle \mathrm{quit}.\overline{\mathrm{unlock}}\rangle_{\overline{\mathrm{unlock}}}$$

running in parallel with the continuation $\mathrm{unlock}.\checkmark + \lightning\overline{\mathrm{quit}}.\mathrm{unlock}.\checkmark$ of O. ⋄

In Example 2 both S_1 MAY O and S_2 MAY O hold true. In fact, there is at least a successful computation in both scenarios, namely the one in which the client manages to send $\overline{\mathrm{quit}}$ so that there is no failure. In this case of normal execution both systems pass the test. On the other hand, an observer can tell apart systems S_1 and S_2 if it causes the failure of $\overline{\mathrm{quit}}$. In fact, S_1' the failure would trigger the compensation $\overline{\mathrm{unlock}}$ whereas in system S_2' the observer would remain blocked after the failure since the compensation is installed in a different scope. It is immediate from the definitions in § 3.2 that S_1 MUST O holds and $\bot \in \mathfrak{R}(S_2 \parallel O)$, therefore S_2 MUST O does not hold.

Example 2 shows that, by specifying different transactional attributes we obtain different reconfiguration semantics (i.e., the scopes of the resulting systems are differently configured) which may lead to different behaviours when failures have to be handled and propagated. In general the behaviour of a system changes depending on how its processes are nested in transactional scopes, as shown in Example 3.

Example 3. Given any environment Γ, it is possible to find $P, R, Q \in \mathcal{P}$ such that (omitting Γ for simplicity):

$\langle P \mid R\rangle_Q \not\simeq_{\mathbf{M}} \langle P\rangle_Q \mid \langle R\rangle$	*e.g.*, $\langle \overline{a} \mid \overline{b}\rangle_{\overline{c}} \not\simeq_{\mathbf{M}} \langle \overline{a}\rangle_{\overline{c}} \mid \langle \overline{b}\rangle$ with $O = \lightning b.c.\checkmark$
$\langle P\rangle_Q \mid \langle R\rangle \not\simeq_{\mathbf{M}} \langle P \mid R\rangle_Q$	*e.g.*, $\langle \overline{a}\rangle_{\overline{c}} \mid \langle \overline{b}\rangle \not\simeq_{\mathbf{M}} \langle \overline{a} \mid \overline{b}\rangle_{\overline{c}}$ with $O = \lightning a.b.\checkmark$
$\langle P\rangle_Q \mid \langle R\rangle_Q \not\simeq_{\mathbf{M}} \langle P \mid R\rangle_Q$	*e.g.*, $\langle \overline{a}\rangle_{\overline{c}} \mid \langle \overline{b}\rangle_{\overline{c}} \not\simeq_{\mathbf{M}} \langle \overline{a} \mid \overline{b}\rangle_{\overline{c}}$ with $O = \lightning a.b.\checkmark$
$\langle P \mid R\rangle_Q \not\simeq_{\mathbf{M}} \langle P\rangle_Q \mid R$	*e.g.*, $\langle \overline{a} \mid \overline{b}\rangle_{\overline{c}} \not\simeq_{\mathbf{M}} \langle \overline{a}\rangle_{\overline{c}} \mid \overline{b}$ with $O = \lightning b.c.\checkmark$
$\langle P\rangle_Q \mid R \not\simeq_{\mathbf{M}} \langle P \mid R\rangle_Q$	*e.g.*, $\langle \overline{a}\rangle_{\overline{c}} \mid \overline{b} \not\simeq_{\mathbf{M}} \langle \overline{a} \mid \overline{b}\rangle_{\overline{c}}$ with $O = \lightning a.b.\checkmark$
$\langle P \mid R\rangle_Q \not\simeq_{\mathbf{M}} \langle P\rangle_Q \mid \langle R\rangle_Q$	*e.g.*, $\langle \overline{a} \downarrow \overline{e}.\overline{d} \mid d.\overline{b}\rangle_{\overline{c}} \not\simeq_{\mathbf{M}} \langle \overline{a} \downarrow \overline{e}.\overline{d}\rangle_{\overline{c}} \mid \langle d.\overline{b}\rangle_{\overline{c}}$ with $O = \lightning b.e.\checkmark$.

On the left-hand side of each case above we present a counter-example for that case, where the observer is satisfied for the first process and not for the second one. In words, transactional scopes do not commute with or distribute over parallel composition. ⋄

5 Concluding Remarks and Related Work

Building on ATc [1], we define a theory of testing to study reconfigurable SOC transactions in presence of failures. The proposed framework captures the interplay between the semantics of processes and their compensations, and the dynamic reconfiguration of transactional scopes due to the run-time invocation of new services.

Transactional attributes of EJB have been adapted to SOC transitions in [1] where ATc has been introduced. The primitives of ATc allow one to *determine* and *control* the dynamic reconfiguration of distributed transactions so to have consistent and predictable failure propagation. Also, in [1] it has been given a type system for ATc that guarantees absence of failures due to misuse of transactional attributes.

A comparison of the linguistic features of ATc wrt other calculi featuring distributed transactions has been given in [1]; StAC [3] and CJoin [2] possibly are the closest calculi to ATc as they feature arbitrarily nested transactions and separate process communication from error/compensation. CJoin offers a mechanism to merge different scopes but it is not offering the flexibility of the transactional attributes of ATc. To the best of our knowledge, none of the calculi proposed in literature has given a testing semantics (in [5] testing equivalence is given for the Join calculus but not adapted to Cjoin).

One of the limitations of our approach is the lack of link mobility à la π-calculus; the extension of our approach to a name passing calculus is left as future work. Other interesting extensions would be to allow the communication of attributes and a primitive enabling a service s to make a parametrized invocation of a service s' using the same attribute supported by s (recall that attributes are when services are published in containers). Also, the interplay of attributes with the behaviour of observed systems deserves further investigation as in some contexts it could be possible to inter-change the attributes obtaining the same observed behaviour.

References

1. L. Bocchi and E. Tuosto. A Java inspired semantics for transactions in SOC. In *TGC 2010*, LNCS. Springer-Verlag, 2010. To appear.
2. R. Bruni, H. Melgratti, and U. Montanari. Nested commits for mobile calculi: extending Join. In J.-J. Lévy, E. Mayr, and J. Mitchell, editors, *IFIP TCS 2004*, pages 563–576. Kluwer, 2004.
3. M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In R. De Nicola, G. Ferrari, and G. Meredith, editors, *Coordination 2004*, volume 2949 of *LNCS*, pages 87–104. Springer-Verlag, 2004.
4. R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Comput. Sci.*, 34(1–2):83–133, Nov. 1984.
5. C. Laneve. May and must testing in the Join-Calculus. Technical Report UBLCS-96-4, Department of Computer Science University of Bologna, 1996.
6. D. Panda, R. Rahman, and D. Lane. *EJB 3 in action*. Manning, 2007.
7. Sun Microsystems. Enterprise JavaBeans (EJB) technology, 2009. `http://java.sun.com/products/ejb/`.