A Process Calculus Analysis of Compensations

Luís Caires Carla Ferreira Hugo Vieira

CITI / Departamento de Informática, FCT Universidade Nova de Lisboa, Portugal

Abstract. Conversations in service-oriented computation are frequently long running. In such a setting, traditional ACID properties of transactions cannot be reasonably implemented, and compensation mechanisms seem to provide convenient techniques to, at least, approximate them. In this paper, we investigate the representation and analysis of structured compensating transactions within a process calculus model, by embedding in the Conversation Calculus certain structured compensation programming abstractions inspired by the ones proposed by Butler, Ferreira, and Hoare. We prove the correctness of the embedding after developing a general notion of stateful model for structured compensations and related results, and showing that the embedding induces such a model.

1 Introduction

Conversations in service-oriented computing scenarios are frequently long running. In such a setting, traditional ACID properties of transactions cannot be reasonably implemented, and compensation mechanisms seem to provide convenient techniques to, at least, approximate them. Although well known in the context of transaction processing systems for quite a long time (see e.g., [9]), the use of compensation as a mechanism to undo the effect of long running transactions, and thus recover some properties of confined ACID transactions, is now usually assumed to be the recovery mechanism of choice for aborted transactions in distributed services.

In this paper, we investigate the representation and analysis of structured compensating transactions in the Conversation Calculus [17, 16], a session based nominal process calculus for service-oriented computing. In particular, we show how the basic abstractions present in the CC are enough to express general structured compensating transactions, and illustrate how a process calculus model may be used to reason about the correctness of such programming abstractions. In order to carry out our study in an abstract, implementation independent setting, we introduce and analyze a general model of stateful compensating transactions. We take as starting point the core language for structured compensations introduced in [7], the compensating CSP calculus (cCSP), but reinterpret the basic constructions in a way more suitable for reasoning about stateful computation models, where programs (e.g, tasks, activities, elements of transactions) may have visible side effects on global resources. In our model, the most elementary program is an *atomic action*. An atomic action enjoys the following atomicity property: it either executes successfully to completion, or it aborts. In the case of abortion, a basic action is required not to perform any relevant observable behavior, except signaling abortion by throwing an exception. Structured compensable transactions are defined from basic compensable transactions.

The basic compensable transaction is a pair $P \div Q$ where P and Q are atomic actions. The action Q is intended to undo the effect of the P action, leading to a state that should be in some precise sense "similar" to the state right before P was executed; in general such state cannot be recovered precisely. Complex structured compensable transactions may then be defined by composition under various control operators: sequential composition $T; R$, parallel composition $T \mid R$, choice $T \oplus R$ (or conditional **if** c **then** T **else** R), and exception handling $T \rhd R$. This last operator is not present in [7] and turns out useful to deal with failure detection inside compensating transactions. An arbitrary structured compensable transaction T may then be encapsulated as a basic action, by means of the operator $\langle T \rangle$, enjoying the fundamental properties of a basic action described above, in particular, the atomicity property.

The first part of the paper develops a general notion of stateful model for structured compensations, providing a precise foundation for the forthcoming semantic analysis. Our notion of compensating model (Definition 3.1) is fairly abstract, parametric on the intended notion of "similarity" between states, and independent of the concrete underlying operational model in which compensating transactions would be embedded. However, it already allows us to state precise conditions on basic actions enough to derive general reversibility, cancellation and atomicity results (Theorems 3.6, 3.7 and 3.9), that may then be reused in each particular application, as would be needed to reason about compensating transactions in some process calculus model.

In the second part of the paper, we present and prove correct (Theorem 4.5) an embedding of our language for compensating transactions in the Conversation Calculus. The encoding builds on the Conversation Calculus exception handling primitives, but highlights the essential difference between exceptions and compensations: obviously these are quite different and even independent concepts. Exceptions are a mechanism to signal abnormal conditions during program execution, while compensations are commands intended to undo the effects of previously successfully completed tasks during a transaction. The synergy between exceptions and compensations is also usefully exploited in the compensable exception handler construct $T \rhd R$ available in compensable programs. We will also illustrate how our encoding may be used to specify and reason about specifications of service-oriented systems using compensating transactions, including distributed ones, in the Compensation Calculus and related models. To the best of our knowledge, this work is the first addressing the semantic analysis of structured compensating transactions, within a concurrent process calculus framework.

In Section 2 we review the syntax and semantics of the Conversation Calculus. In Section 3, the general framework of compensating model is defined and analyzed. In Section 4, we present, prove, and exemplify our embedding of compensating transactions in the Conversation Calculus.

2 The Conversation Calculus

The Conversation Calculus (CC) [17, 16] is a process model for service-oriented computing that builds on the concepts of process delegation, loose-coupling of subsystems, and, crucially, conversation contexts. A conversation context is a medium where several partners may interact by exchanging messages, possibly concurrently. It can be

$$
\begin{array}{llll}
a, b, c, \ldots & \in \Lambda & \textit{(Names)} & d ::= \downarrow \mid \uparrow \quad \textit{(Directions)} \\
x, y, z, \ldots & \in \mathcal{V} & \textit{(Variables)} & \alpha ::= l^d!(n) \quad \textit{(Output)} \\
n, m, o \ldots & \in \Lambda \cup \mathcal{V} & & \quad\mid\quad l^d?(x) \quad \textit{(Input)} \\
l, s \ldots & \in \mathcal{L} & \textit{(Labels)} & \\
\mathcal{X}, \mathcal{Y}, \ldots & \in \chi & \textit{(Process Vars)} &
\end{array}
$$

$$
\begin{array}{lll}
P, Q ::= \mathbf{0} & \textit{(Inaction)} & \mid n \blacktriangleleft [P] \quad \textit{(Conversation Access)} \\
\quad\mid\quad P \mid Q & \textit{(Parallel Composition)} & \mid \Sigma_{i \in I}\ \alpha_i.P_i \quad \textit{(Prefix Guarded Choice)} \\
\quad\mid\quad (\boldsymbol{\nu} a)P & \textit{(Name Restriction)} & \mid \mathtt{try}\ P\ \mathtt{catch}\ Q \quad \textit{(Exception Block)} \\
\quad\mid\quad \mathtt{rec}\, \mathcal{X}.P & \textit{(Recursion)} & \mid \mathtt{throw}.P \quad \textit{(Exception Throw)} \\
\quad\mid\quad \mathcal{X} & \textit{(Variable)} &
\end{array}
$$

Fig. 1. The Conversation Calculus.

distributed in many pieces, and processes in any piece may seamlessly talk to processes in the same or any other piece of the same conversation context. Conversation context identities can be passed around, allowing participants to dynamically join and leave conversations, while being able to coordinate the interactions from a local viewpoint. Mechanisms for handling exceptional behavior seem to be essential in distributed computation in general, and in service-oriented computing in particular. Thus, the CC also includes basic exception handling primitives. In this section, we present the syntax of the Conversation Calculus (CC), and recall its operational semantics.

The CC extends the π-calculus [14] static fragment with the conversation construct $n \blacktriangleleft [P]$, and replaces channel based communication with context-sensitive message based communication. We use here a monadic version, and omit the context-awareness primitive of [17], which is irrelevant for the purpose of this paper. The syntax of the calculus is defined in Figure 1. We assume given an infinite set of names Λ, an infinite set of variables $\mathcal{V}$, an infinite set of labels $\mathcal{L}$, and an infinite set of process variables χ. The static fragment is defined by the inaction $\mathbf{0}$, parallel composition $P \mid Q$, name restriction $(\boldsymbol{\nu} a)P$ and recursion $\mathtt{rec}\, \mathcal{X}.P$. The conversation access construct $n \blacktriangleleft [P]$, allows a process to initiate interactions, as specified by P, in the conversation n.

Communication is expressed by the guarded choice construct $\Sigma_{i \in I}\ \alpha_i.P_i$, meaning that the process may select some initial action α_i and then progress as P_i. Communication actions are of two forms: $l^d!(n)$ for sending messages and $l^d?(x)$ for receiving messages. Thus, message communication is defined by the label l and the direction d. There are two message directions: $\downarrow$ (read "here") meaning that the interaction should take place in the current conversation or $\uparrow$ (read "up") meaning that the interaction should take place in the enclosing (caller) conversation. N.B.: to lighten notation we omit the $\downarrow$ in the $\downarrow$-directed messages without any ambiguity. Notice that message labels (from $l \in \mathcal{L}$) are not names but free identifiers (cf. record labels or XML tags), and therefore not subject to fresh generation, restriction or binding.

The CC includes two exception related primitives; we adapt the classical $\mathtt{try} - \mathtt{catch} -$ and $\mathtt{throw} -$ to a concurrent setting. The primitive to signal exceptional behavior is $\mathtt{throw}.\mathit{Excp}$. This construct is used to throw an exception with continuation

$$l^d!(a).P \xrightarrow{l^d!(a)} P \ (out) \qquad l^d?(x).P \xrightarrow{l^d?(a)} P\{x/a\} \ (inp) \qquad \frac{P\{\mathcal{X}/\mathtt{rec}\,\mathcal{X}.P\} \xrightarrow{\lambda} Q}{\mathtt{rec}\,\mathcal{X}.P \xrightarrow{\lambda} Q}(rec)$$

$$\frac{\alpha_j.P_j \xrightarrow{\lambda} Q \quad j \in I}{\Sigma_{i\in I}\ \alpha_i.P_i \xrightarrow{\lambda} Q}(sum) \qquad \frac{P \xrightarrow{(\nu a)\overline{\lambda}} P' \quad Q \xrightarrow{\lambda} Q'}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} a)(P' \mid Q')}(clo) \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}(com)$$

$$\frac{P \xrightarrow{\lambda} Q \quad a = out(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{(\nu a)\lambda} Q}(opn) \qquad \frac{P \xrightarrow{\lambda} Q \quad a \notin \mathit{fn}(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{\lambda} (\boldsymbol{\nu} a)Q}(res) \qquad \frac{P \xrightarrow{\lambda} Q}{P \mid R \xrightarrow{\lambda} Q \mid R}(par)$$

Fig. 2. Basic operators (π-calculus).

$$\frac{P \xrightarrow{\lambda^{\uparrow}} Q}{c \blacktriangleleft [P] \xrightarrow{\lambda^{\downarrow}} c \blacktriangleleft [Q]}(her) \qquad \frac{P \xrightarrow{\lambda^{\downarrow}} Q}{c \blacktriangleleft [P] \xrightarrow{c\cdot\lambda^{\downarrow}} c \blacktriangleleft [Q]}(loc)$$

$$\frac{P \xrightarrow{a\ \lambda^{\downarrow}} Q}{c \blacktriangleleft [P] \xrightarrow{a\ \lambda^{\downarrow}} c \blacktriangleleft [Q]}(thr) \qquad \frac{P \xrightarrow{\tau} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]}(tau)$$

Fig. 3. Conversation operators.

$$\mathtt{throw}.P \xrightarrow{\mathtt{throw}} P \ \ (i) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{P \mid Q \xrightarrow{\mathtt{throw}} R}(ii) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{n \blacktriangleleft [P] \xrightarrow{\mathtt{throw}} R}(iii)$$

$$\frac{P \xrightarrow{\lambda} Q \qquad \lambda \neq \mathtt{throw}}{\mathtt{try}\ P\ \mathtt{catch}\ R \xrightarrow{\lambda} \mathtt{try}\ Q\ \mathtt{catch}\ R}(iv) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{\mathtt{try}\ P\ \mathtt{catch}\ Q \xrightarrow{\tau} Q \mid R}(v)$$

Fig. 4. Exception handling operators.

Excp. It has the effect of forcing the termination of all other processes running in all enclosing contexts, up to the point where a $\mathtt{try} - \mathtt{catch}$ block is found (if any). The continuation Excp will be activated when (and if) the exception is caught by such an exception handler. The exception handler construct $\mathtt{try}\ P\ \mathtt{catch}\ \mathit{Handler}$ actively allows a process P to run until some exception is thrown inside P. At that moment, P is terminated, and the $\mathit{Handler}$ process guarded by $\mathtt{try} - \mathtt{catch}$, will be activated concurrently with the continuation Excp of the $\mathtt{throw}.\mathit{Excp}$ that originated the exception, in the context of the given $\mathtt{try} - \mathtt{catch} -$ block.

Only conversation names (in Λ) may be subject to binding, and freshly generated via $(\nu a)P$. The distinguished occurrences of a, x and $\mathcal{X}$ are binding occurrences in $(\boldsymbol{\nu} a)P$, $l^d?(x).P$, and $\mathtt{rec}\,\mathcal{X}.P$, respectively. The sets of free ($\mathit{fn}(P)$) and bound ($\mathit{bn}(P)$) names, free variables ($\mathit{fv}(P)$), and free process variables ($\mathit{fpv}(P)$) in a process P are defined as usual. We implicitly identify α-equivalent processes.

The operational semantics of the CC is defined by a labeled transition system. For clarity, we split the presentation in three sets of rules, one (in Figure 2) containing the

rules for the basic operators, which are similar to the corresponding ones in the π-calculus (see [15]), other (in Figure 3) grouping the rules specific to the conversations, and a final one (in Figure 4) grouping the rules specific to exceptions. A transition $P \xrightarrow{\lambda} Q$ states that process P may evolve to process Q by performing the action represented by the transition label λ. Transition labels (λ) and actions (σ) are given by

$$\sigma ::= \boldsymbol{\tau} \mid l^d!(a) \mid l^d?(a) \mid \texttt{throw} \qquad \lambda ::= c\,\sigma \mid \sigma \mid (\nu a)\lambda$$

An action $\boldsymbol{\tau}$ denotes an internal communication, and actions $l^d!(a)$ and $l^d?(a)$ represent communications with the environment; these correspond to the basic actions a process may perform in the context of a given conversation. To capture the observational semantics of processes, transition labels need to register not only the action but also the conversation where the action takes place. So, a transition label λ containing $c\ \sigma$ is said to be *located at* conversation c (or just *located*), otherwise is said to be *unlocated*. In $(\nu a)\lambda$ the distinguished occurrence of a is bound with scope λ (cf., the π-calculus bound output actions). For a communication label λ we denote by $\overline{\lambda}$ the dual matching label obtaining by swapping inputs with outputs, such that $\overline{l^d!(a)} = l^d?(a)$ and $\overline{l^d?(a)} = l^d!(a)$. We use *fn*($\lambda$) and *bn*($\lambda$) to denote (respectively) the free and bound names of a transition label.

Transition rules presented in Figure 2 closely follow the ones for the π-calculus and should be fairly clear to a reader familiar with mobile process calculi. For example, rule (opn) corresponds to the bound output or extrusion rule, in which a bound name a is extruded to the environment in an output message λ: we define $out(\lambda) = a$ if $\lambda = l^d!(a)$ or $\lambda = c\,l^d!(a)$ and $c \neq a$. We discuss the intuitions behind the rules for conversation contexts (Figure 3). In rule (her) an $\uparrow$ directed message (to the caller conversation) becomes $\downarrow$ (in the current conversation), after passing through the conversation access boundary. We note by λ^d a transition label λ^d containing the direction d ($\uparrow, \downarrow$), and by $\lambda^{d'}$ the label obtained by replacing d by d' in λ^d (e.g., if $\lambda^{\uparrow}$ is $\texttt{askPrice}^{\uparrow}?(a)$ then $\lambda^{\downarrow}$ is $\texttt{askPrice}^{\downarrow}?(a)$). In rule (loc) an unlocated $\downarrow$ message (in the current conversation) gets explicitly located at the conversation c in which it originates. Given an unlocated label λ, we represent by $c \cdot \lambda$ the label obtained by locating λ at c (e.g., if $\lambda^{\downarrow}$ is $\texttt{askPrice}^{\downarrow}?(p)$ then $c \cdot \lambda^{\downarrow}$ is $c\ \texttt{askPrice}^{\downarrow}?(p)$). In rule (thr) an already located communication label transparently crosses some other conversation boundary, and likewise for a $\boldsymbol{\tau}$ label in (tau). The rules for exceptions (in Figure 4) should be easily understood, given the explanation already provided above.

The operational semantics of the CC is given by a relation of reduction, noted $P \rightarrow Q$, and defined as $P \xrightarrow{\boldsymbol{\tau}} Q$. The behavioral semantics is defined by a standard notion of bisimilarity, defined in terms of our labeled transition system.

Definition 2.1 (Strong Bisimulation). *A (strong) bisimulation is a symmetric binary relation $\mathcal{R}$ on processes such that, for all processes P and Q, if $P\mathcal{R}Q$ then:*

If $P \xrightarrow{\lambda} P'$ and $bn(\lambda) \cap fn(Q) = \emptyset$ then there is Q' such that $Q \xrightarrow{\lambda} Q'$ and $P'\mathcal{R}Q'$.

We denote by $\sim$ (strong bisimilarity) the largest bisimulation.

Theorem 2.2. *Strong bisimilarity is a congruence.*

For input, we consider the universal instantiation congruence principle: if $P\{x/a\} \sim Q\{x/a\}$ for all a then $l^d?(x).P \sim l^d?(x).Q$ (cf., [15] Theorem 2.2.8(2)). Using conversation contexts and the basic message based communication mechanisms, useful programming abstractions for service-oriented systems may be idiomatically defined in the CC, namely service definition and instantiation constructs (defined as primitives in [17]).

$$\begin{aligned}
\texttt{def}\ s \Rightarrow P \quad &\triangleq s?(x).x \blacktriangleleft [P] \\
\texttt{new}\ n \cdot s \Leftarrow Q &\triangleq (\boldsymbol{\nu} c)(n \blacktriangleleft [s!(c)] \mid c \blacktriangleleft [Q]) \\
\star\texttt{def}\ s \Rightarrow P \quad &\triangleq \texttt{rec}\, \mathcal{X}.s?(x).(\mathcal{X} \mid x \blacktriangleleft [P])
\end{aligned}$$

The **def** form publishes a service definition, while the **new** form instantiates a service definition. Interaction between **new** and **def** results in the creation of a fresh conversation context where a session between client and server may take place.

We present a simple example of a CC specification of a service oriented system. Consider a service provider *Artic* providing a persistent service *getTemp*. Whenever invoked such service reads the current value of a sensor at the service provider site, and sends it to the caller endpoint.

$$ASite \triangleq Artic \blacktriangleleft [\ Sensor \mid \star\ \texttt{def}\ getTemp \Rightarrow (readSens^{\uparrow}?(x).value!(x))\ ]$$

Sensor is a process running in context $Artic \blacktriangleleft [\cdots]$ that may output $ReadSens(t)$ messages therein. To use the service in "one shot", a client may use the code

$$Client \triangleq \texttt{new}\ Artic \cdot getTemp \Leftarrow (value?(x).temp^{\uparrow}!(x))$$

The net effect of this code is to drop a $temp(t)$ message in the client conversation context, where t is the temperature value as read at the remote *Artic* site. The conversation in this example is just the exchange of a $value(t)$ message. More examples of CC models of service oriented scenarios may be found in [16]. We will show in the rest of the paper how the basic abstractions present in the CC are enough to express general structured compensating transactions, and illustrate how a process calculus model may be used to reason about the correctness of such programming abstractions. In order to carry out such a study in a abstract, implementation independent setting, we introduce and analyze in the next section a general model of stateful compensating transactions.

3 A Stateful Model of Compensating Transactions

In this section we construct a model of compensating transactions that provides a framework to reason about compensations.

Definition 3.1 (Compensation Model). *A compensation model is a pair $(\mathcal{S}, \mathcal{D})$ where $\mathcal{S}$ gives its static structure and $\mathcal{D}$ gives its dynamic structure. The static structure $\mathcal{S} = (S, \mid, \#, \bowtie)$ is defined such that:*

- *S is a set of (abstract) states*
- *$\mid$ is a partial composition operation on states*
- *$\#$ is an apartness relation on states*

Atomic actions:

$A, B ::= a$ (*Primitive action)*
$\mid \langle R \rangle$ (*Transaction)*

Basic programs:

$P, Q ::= A$ (*Action)*
$\mid P \,;\, Q$ (*Sequential composition)*
$\mid P \oplus Q$ (*Choice)*
$\mid P \mid Q$ (*Parallel composition)*
$\mid P \rhd Q$ (*Exception handler)*
$\mid \mathit{skip}$ (*Normal termination)*
$\mid \mathit{throw}$ (*Throw an interrupt)*

Compensable programs:

$R, T ::= A \div B$ (*Compensation pair)*
$\mid R \,;\, T$
$\mid R \oplus T$
$\mid R \mid T$
$\mid R \rhd T$
$\mid \mathit{skipp}$
$\mid \mathit{throww}$

Fig. 5. Syntax of Compensating CSP.

– $\bowtie$ *is an equivalence relation on* S

The relation $s\#s'$ *is symmetric. We have that* $s \mid s'$ *is defined if and only if* $s\#s'$*. The composition operation* $\mid$ *is symmetric, associative, and commutative.*

The dynamic structure $\mathcal{D} = (\Sigma, \xrightarrow{a})$ *is defined such that:*

– Σ *is a set of primitive actions*
– $\xrightarrow{a}$ *is a labeled (by elements of* Σ*) transition system between states.*

Independent transitions are assumed to preserve apartness, as expressed by the following locality principle: if $s\#s'$*,* $s \xrightarrow{a} t$*, and* $s' \xrightarrow{b} t'$ *then* $t\#t'$*.*

Intuitively, we can view a state as a collection of state entities. In this context, given states s and s', the apartness relation asserts that the state entities of s and s' do not interfere. If two states s and s' are independent ($s\#s'$), they may be composed into a new state $t = s \mid s'$. Actions denote imperative transformations on states, in the usual sense. The equivalence relation $\bowtie$ expresses what states are to be considered "equivalent" from the view point of the model. Remember that undoing an action may roll the system back to a state that may not be strictly semantically equivalent to the initial one in terms of the underlying operational semantics, but "equivalent enough" for the purpose of compensation.

3.1 Compensating CSP

Our cCSP syntax (in Figure 5) includes two types of atomic actions: primitive actions and transactions. Both types of actions can be seen as atomic in the sense that they either occur as a whole, or they do not occur at all. While primitive actions are atomic by definition, for transactions it is necessary to prove that their behavior approximates atomicity (see Theorem 3.9).

In basic programs atomic activities can be composed by sequential, parallel, and internal choice operators. The language includes two primitive programs: throw which

$$
\begin{aligned}
\mathit{Bookstore} &= \langle \mathit{chooseBooks} \ ; \ \mathit{Pay} \ ; \ \mathit{ProcessOrder} \rangle \\
\mathit{ChooseBooks} &= \mathsf{checkout} \div \mathit{skip} \oplus (\mathit{ChooseBook} \ ; \ \mathit{ChooseBooks}) \\
\mathit{ChooseBook} &= (\mathsf{addBook} \div \mathsf{removeBook} \ ; \ (\mathsf{priceOk} \div \mathit{skip} \oplus \mathit{throww})) \rhd \mathit{skipp} \\
\mathit{Pay} &= \mathsf{processCard} \div \mathsf{refund} \rhd (\mathsf{sendNotice} \div \mathit{skip} \ ; \ \mathit{throww})
\end{aligned}
$$

Fig. 6. Bookstore example.

raises an exception causing the program to stop and fail; and $skip$ that describes an inactive program. The exception handler may be used to catch abortion: in $P \rhd Q$, an abortion of P triggers execution of the handler Q. Transaction $\langle R \rangle$ converts a compensable program into a basic one with atomic behavior.

When dealing with compensable programs a transaction processing system must ensure that on failure of a transaction, all the necessary atomic compensations are performed in an appropriate order to compensate for the effect of everything that has actually happened so far. To achieve reversibility, compensable programs have compensation pairs as their building unit. A compensation pair is constructed from two atomic activities. In the pair $A \div B$, if forward behavior A successfully terminates, then compensation behavior B is stored, to be possibly used in a later compensation. If A fails, the compensation is just discarded. Notice that the model assumes that compensations never fail, as it is intended for reasoning about consistent programs (see Definition 3.5). Sequential composition of compensable processes must ensure that the compensations for all actions performed will be accumulated in the reverse order to their original performance. Parallel composition of compensable processes ensures that compensations for performed actions will be accumulated in parallel. Internal choice is similar to the basic case. Programs $skip$ and $throw$ are mapped to compensable programs by assigning program $skip$ as their compensation, so that $skipp \triangleq skip \div skip$ and $throww \triangleq throw \div skip$. Notice that it is irrelevant which compensation is paired with $throw$, as only a successful terminated program may run its compensation ($throw$ always fails). Compensable exception handler behaves similarly to the correspondent standard operator: an abortion raised by R will trigger the execution of handler T.

Figure 6 presents a transaction for ordering books. The transaction starts with program $chooseBooks$ that allows the user to repeatedly select individual books until checkout occurs. In $ChooseBook$ we have a compensation pair. On the occurrence of forward action addBook, that adds the book to the client's basket, compensation removeBook is stored. Next, if the price of the book is deemed too expensive, an exception is thrown triggering the execution of compensation removeBook. Compensation reverses the forward action by removing the book that has just been added to the basket. This exception is then caught by a handler to avoid its propagation, allowing the user to continue selecting books. The second program in the transaction is exception handler Pay. It starts with a compensation pair with forward action processCard and compensation refund. On abort of action processCard the fault is catch by a handler. The aim of this handler is to notify the client whenever there are problems processing the payment with the credit card. Having done that, the handler re-throws the exception, causing the compensation stored until that point to be executed, removing all books

Rules (xi), (xii), (f), (g), and (h) assume that $s = r \,|\, t$, $r \# t$, and $s' = r' \,|\, t'$.

$$(skip)^s_s \checkmark \;\; (i) \qquad (throw)^s_s \; * \;\; (ii) \qquad \frac{s \xrightarrow{a} s'}{(a)^s_{s'} \checkmark}(iii) \qquad \frac{\nexists_{s'}\, s \xrightarrow{a} s'}{(a)^s_s \, *}(iv)$$

$$\frac{(P)^s_{s'} \checkmark \quad (Q)^{s'}_{s''} \checkmark}{(P \,;\, Q)^s_{s''} \checkmark}(v) \qquad \frac{(P)^s_{s'} \, *}{(P \,;\, Q)^s_{s'} \, *}(vi) \qquad \frac{(P)^s_{s'} \checkmark \quad (Q)^{s'}_{s''} \, *}{(P \,;\, Q)^s_{s''} \, *}(vii)$$

$$\frac{(P)^s_{s'}\, d}{(P \oplus Q)^s_{s'}\, d}(viii) \qquad \frac{(Q)^s_{s'}\, d}{(P \oplus Q)^s_{s'}\, d}(ix) \qquad \frac{(R)^s_{s'} \overset{d}{\leftarrow} P}{(\langle R \rangle)^s_{s'}\, d}(x)$$

$$\frac{(P)^r_{r'} \checkmark \quad (Q)^t_{t'} \checkmark}{(P \mid Q)^s_{s'} \checkmark}(xi) \qquad \frac{(P)^r_{r'}\, d_p \quad (Q)^t_{t'}\, d_q \quad d_p = * \vee d_q = *}{(P \mid Q)^s_{s'} \, *}(xii)$$

$$\frac{(P)^s_{s'} \checkmark}{(P \rhd Q)^s_{s'} \checkmark}(xiii) \qquad \frac{(P)^s_{s'} \; * \quad (Q)^{s'}_{s''} \checkmark}{(P \rhd Q)^s_{s''} \checkmark}(xiv) \qquad \frac{(P)^s_{s'} \; * \quad (Q)^{s'}_{s''} \, *}{(P \rhd Q)^s_{s''} \, *}(xv)$$

Fig. 7. Rules for basic programs.

from the basket. Although we omit the description of the last program, $ProcessOrder$, the cCSP semantics ensures that this program is executed only when the payment succeeds. Furthermore, a failure on $ProcessOrder$ would reverse all actions done until then, *i.e.*, return all books in the basket and refund the clients payment.

Having defined its syntax and informal semantics, we now show how cCSP can be interpreted in a compensation model.

Definition 3.2 (cCSP semantics). *Given a compensation model $\mathcal{M}$, a cCSP interpretation in $\mathcal{M}$ is a pair (B, C) of relations $B \subseteq S \times P \times S \times T$ and $C \subseteq S \times R \times S \times T \times P$ that specify the effects and the final status of programs. The final state is represented by an element of $T = \{*, \checkmark\}$, where $\checkmark$ means success and $*$ abortion. The last component of relation C specifies the stored compensation of a compensable process.*

We write $(P)^s_{s'}\; d$ for $(s, P, s', d) \in B$ and $(R)^s_{s'} \overset{d}{\leftarrow} P$ for $(s, R, s', d, P) \in C$. B and C are required to minimally satisfy the properties shown in Figs. 7 and 8.

We briefly review some of the rules of Figure 7 for basic programs: rules (iii) and (iv) state, respectively, that primitive action a either terminates successfully if it causes the state to evolve, or fails if it cannot evolve from state s; rules $(viii)$ and (ix) show that basic program $P \oplus Q$ replicates the outcome of either program P or program Q.

For defining the properties for compensable programs it is necessary to introduce the notion of forward programs. The definition of forward programs is trivial, with the exception of $(R \rhd T)^+$. For this program it is necessary to propagate the outcome of R and clear all stored compensations.

Definition 3.3 (Forward program). *For a compensable program R we define the forward program R^+ (or positive program) inductively follows:*

$$(A \div B)^+ \triangleq A \qquad skipp^+ \triangleq skip \qquad throww^+ \triangleq throw \qquad (R \,;\, T)^+ \triangleq R^+ \,;\, T^+$$
$$(R \oplus T)^+ \triangleq R^+ \oplus T^+ \qquad (R \mid T)^+ \triangleq R^+ \mid T^+ \qquad (R \rhd T)^+ \triangleq \langle R \rangle \rhd T^+$$

$$\frac{(A)^{s}_{s'}\ \checkmark}{(A \div B)^{s}_{s'} \overset{\checkmark}{\leftarrow} B}(a) \qquad \frac{(A)^{s}_{s}\ *}{(A \div B)^{s}_{s} \overset{*}{\leftarrow} \mathit{skip}}(b) \qquad \frac{(R)^{s}_{s'} \overset{\checkmark}{\leftarrow} P \quad (T)^{s'}_{s''} \overset{\checkmark}{\leftarrow} Q}{(R\ ;\ T)^{s}_{s''} \overset{\checkmark}{\leftarrow} Q\ ;\ P}(c)$$

$$\frac{(R)^{s}_{s'} \overset{*}{\leftarrow} \mathit{skip}}{(R\ ;\ T)^{s}_{s'} \overset{*}{\leftarrow} \mathit{skip}}(d) \qquad \frac{(R)^{s}_{s'} \overset{\checkmark}{\leftarrow} P \quad (T)^{s'}_{s''} \overset{*}{\leftarrow} \mathit{skip} \quad (P)^{s''}_{r}\ \checkmark}{(R\ ;\ T)^{s}_{r} \overset{*}{\leftarrow} \mathit{skip}}(e)$$

$$\frac{(R)^{r}_{r'} \overset{\checkmark}{\leftarrow} P \quad (T)^{t}_{t'} \overset{\checkmark}{\leftarrow} Q}{(R \mid T)^{s}_{s'} \overset{\checkmark}{\leftarrow} P \mid Q}(f) \qquad \frac{(R)^{r}_{r''} \overset{\checkmark}{\leftarrow} P \quad (T)^{t}_{t'} \overset{*}{\leftarrow} \mathit{skip} \quad (P)^{r''}_{r'}\ \checkmark}{(R \mid T)^{s}_{s'} \overset{*}{\leftarrow} \mathit{skip}}(g)$$

$$\frac{(R)^{r}_{r'} \overset{*}{\leftarrow} \mathit{skip} \quad (T)^{t}_{t''} \overset{\checkmark}{\leftarrow} Q \quad (Q)^{t''}_{t'}\ \checkmark}{(R \mid T)^{s}_{s'} \overset{*}{\leftarrow} \mathit{skip}}(h) \qquad \frac{(R)^{s}_{s'} \overset{d}{\leftarrow} P}{(R \oplus T)^{s}_{s'} \overset{d}{\leftarrow} P}(i) \qquad \frac{(T)^{s}_{s'} \overset{d}{\leftarrow} Q}{(R \oplus T)^{s}_{s'} \overset{d}{\leftarrow} Q}(j)$$

$$\frac{(R)^{s}_{s'} \overset{\checkmark}{\leftarrow} P}{(R \rhd T)^{s}_{s'} \overset{\checkmark}{\leftarrow} P}(l) \qquad \frac{(R)^{s}_{s'} \overset{*}{\leftarrow} \mathit{skip} \quad (T)^{s'}_{s''} \overset{\checkmark}{\leftarrow} Q}{(R \rhd T)^{s}_{s''} \overset{\checkmark}{\leftarrow} Q}(m) \qquad \frac{(R)^{s}_{s'} \overset{*}{\leftarrow} \mathit{skip} \quad (T)^{s'}_{s''} \overset{*}{\leftarrow} \mathit{skip}}{(R \rhd T)^{s}_{s''} \overset{*}{\leftarrow} \mathit{skip}}(n)$$

Fig. 8. Rules for compensable programs.

Figure 8 shows the properties compensable programs must satisfy. Next, we explain the rules for compensable exception handler. The remaining rules for compensable programs are analogous. Rule (n) states that when R and T abort in sequence, first R and then T, the overall program also aborts and consequently no compensation is stored. Rule (l) shows that whenever R succeeds with stored compensation P, both the final status and compensation program are raised to process $R \rhd T$. Finally, in rule (m) handler T starts executing after program R has aborted. T terminates with success with Q as its compensation. Again, as in (l), these results are lifted to $R \rhd T$.

The syntactic definition of a compensation pair $A \div B$ does not a priori impose any relation between forward action A and compensation action B. However, if transactions are expected to have an all or nothing semantics, compensation B should be programmed to cancel (or revert) the effects of action A, leaving the system in a state $\bowtie$-equivalent to the initial one.

Definition 3.4 (Reverts). *Given atomic actions A and B, we say that B reverts A w.r.t. $\bowtie$ if for all s, t, t' such that $(A)^{s}_{t}\ \checkmark$, if $t' \bowtie t$ then $(B)^{t'}_{s'}\ \checkmark$ and $s \bowtie s'$.*

The above definition describes the property a compensation should satisfy: given a compensation pair $A \div B$, action B should revert A. We expect that compensable programs and transactions are constructed in a way consistent to the underlying compensation model in order to be meaningful.

Definition 3.5 (Consistency). *A compensable program R is $\bowtie$-consistent when for every compensation pair $A \div B$ occurring in R, B reverts A w.r.t. $\bowtie$.*

Theorem 3.6 below states that consistent compensable programs are globally reversible. In this context, a programmer just has to ensure reversibility of individual compensation pairs, according to the intended compensation model, to achieve reversibility of arbitrary structured transactions.

Theorem 3.6 (Reversibility). *Let R be a $\bowtie$-consistent compensable program such that $(R)^{s}_{s'} \overset{\checkmark}{\leftarrow} P$ for some P and $s, s' \in S$. Then $(P)^{s'}_{s''}$ $\checkmark$ and $s'' \bowtie s$.*

Proof. Induction in the structure of the program R. ∎

3.2 Cancellation Semantics

In this section we show that, based on the general notions of compensation model and definition of $\bowtie$-consistency, atomicity of transactions (in a sense to be made precise below) can be effectively achieved. For that very reason, transactions may be used as atomic actions in compensable processes, either as forward or backward actions within compensation pairs. For example, a compensation pair may have a primitive action as its forward activity, while its backward action is a complex transaction. Conversely, a complex transaction might be compensated by a primitive action.

The notion of cancellation for atomic activities can then be extended to compensable programs. Theorem 3.7 asserts that whenever abortion is induced within a $\bowtie$-consistent compensable program, all its effects will be reverted. More specifically, the program will terminate in a state equivalent to the initial state.

Theorem 3.7 (Cancellation). *Let compensable program R be an $\bowtie$-consistent program such that $(R)^{s}_{s'} \overset{*}{\leftarrow}$ skip for some $s, s' \in S$. Then $s' \bowtie s$.*

Proof. Induction in the structure of the program R. ∎

For our next results it is useful to introduce a notion of program behavioral preorder; $P \sqsubseteq Q$ means that program P is simulated by Q w.r.t. the intended compensation model. We restrict $\sqsubseteq$ to basic programs because our results refer only to transactions (if needed, $\sqsubseteq$ could be easily extended to compensable programs).

Definition 3.8 (Program equivalence). *Program preorder $\sqsubseteq$ is the relation on basic programs defined by $P \sqsubseteq Q$ if and only if for all s, s' such that $s \bowtie s'$, if $(P)^{s}_{t}$ d then $(Q)^{s'}_{t'}$ d and $t \bowtie t'$.*

Our main theorem states that the behavior of transactions approximates atomicity: a transaction either throws doing "nothing", because its forward actions have been reverted, or terminates successfully after executing all of its forward actions.

Theorem 3.9 (Atomicity). *Let R be a $\bowtie$-consistent compensable program.*
Then $\langle R \rangle \sqsubseteq R^{+} \oplus throw$.

4 Compensating Transactions in the Conversation Calculus

In this section we present a provably correct embedding of the cCSP language for structured compensating transactions in the Conversation Calculus. We consider a basic action to be implemented by a CC process P conforming to the following behavior: after some interactions with the environment it either sends (only once) the message $ok^{\downarrow}!$ in the current conversation context without any further action, or aborts, by throwing an

$$
\begin{array}{lcl}
[\![P \div Q]\!]_{ok,ab,cm,cb} & \triangleq & [\ \mathtt{try}\ [\![P]\!]_{ok}\ \mathtt{catch}\ ab^{\uparrow}! \mid \\
& & \quad ok?.ok^{\uparrow}!.(cm^{\uparrow}?.[\![Q]\!]_{cb} \mid cb?.cb^{\uparrow}!)\ ] \\
[\![T_1; T_2]\!]_{ok,ab,cm,cb} & \triangleq & [\ [\![T_1]\!]_{ok_1,ab_1,cm_1,cb} \mid \\
& & \quad ab_1?.ab^{\uparrow}! \mid \\
& & \quad ok_1?.[\![T_2]\!]_{ok,ab,cm,cm_1} \mid \\
& & \quad ab?.cm_1!.cb?.ab^{\uparrow}! \\
& & \quad ok?.ok^{\uparrow}!.cm^{\uparrow}?.cm!.cb?.cb^{\uparrow}!\ ] \\
[\![T_1 \oplus T_2]\!]_{ok,ab,cm,cb} & \triangleq & [\ t! + f! \mid t?.[\![T_1]\!]_{ok,ab,cm,cb} \mid \\
& & \quad f?.[\![T_2]\!]_{ok,ab,cm,cb} \mid ab?.ab^{\uparrow}! \mid \\
& & \quad ok?.ok^{\uparrow}!.cm^{\uparrow}?.cm!.cb?.cb^{\uparrow}!\ ] \\
[\![T_1 \mid T_2]\!]_{ok,ab,cm,cb} & \triangleq & [\ [\![T_1]\!]_{ok_1,ab,cm_1,cb_1} \mid [\![T_2]\!]_{ok_2,ab,cm_2,cb_2} \mid \\
& & \quad ok_1?.ok_2?.ok^{\uparrow}!. \\
& & \qquad cm^{\uparrow}?.(cm_1! \mid cm_2! \mid \\
& & \qquad\qquad cb_1?.cb_2?.cb^{\uparrow}!) \mid \\
& & \quad ab?.(ok_1?.cm_1!.cb_1?.ab^{\uparrow}! \mid \\
& & \qquad ok_2?.cm_2!.cb_1?.ab^{\uparrow}! \mid \\
& & \qquad ab?.ab^{\uparrow}!)\ ] \\
[\![T_1 \rhd T_2]\!]_{ok,ab,cm,cb} & \triangleq & [\ [\![T_1]\!]_{ok_1,ab_1,cm_1,cb_1} \\
& & \quad ok_1?.ok^{\uparrow}!. \\
& & \qquad cm^{\uparrow}?.(cm_1! \mid cb_1?.cb^{\uparrow}!) \mid \\
& & \quad ab_1?.([\![T_2]\!]_{ok_2,ab_2,cm_2,cb_2} \mid \\
& & \qquad ok_2?.ok^{\uparrow}!. \\
& & \qquad\quad cm^{\uparrow}?.(cm_2! \mid cb_2?.cb^{\uparrow}!) \mid \\
& & \qquad ab_2?.ab^{\uparrow}!)\ ] \\
[\![\mathit{throww}]\!]_{ok,ab,cm,cb} & \triangleq & [\ ab^{\uparrow}!\ ] \\
[\![\mathit{skipp}]\!]_{ok,ab,cm,cb} & \triangleq & [\ ok^{\uparrow}!.cm^{\uparrow}?.cb^{\uparrow}!\ ] \\
[\![A]\!]_{ok} & \triangleq & A_{ok} \\
[\![P; Q]\!]_{ok} & \triangleq & [\ [\![P]\!]_{ok_1} \mid ok_1?.[\![Q]\!]_{ok} \mid ok?.ok^{\uparrow}!\ ] \\
[\![\langle T \rangle]\!]_{ok} & \triangleq & [\ [\![T]\!]_{ok,ab,cm,cb} \mid ab?.\mathtt{throw}.\mathbf{0} \mid ok?.ok^{\uparrow}!\ ] \\
[\![P]\!]_{cm,cb} & \triangleq & [\ cm^{\uparrow}?.[\![P]\!]_{cb} \mid cb?.cb^{\uparrow}!\ ]
\end{array}
$$

Fig. 9. Encoding Structured Compensating Transactions in the CC.

exception, without any further action. If the outcome is abortion, the system should be left in the "same" state (in the sense of an appropriate $\bowtie$ relation) as it was before the execution of the basic action was attempted.

Our encoding is defined in Fig. 9. We use the abbreviation $[\ P\ ] \triangleq (\boldsymbol{\nu} n)(n \blacktriangleleft [\ P\ ])$, to represent an anonymous (restricted) context (useful to frame local computations). We denote by $[\![P]\!]_{ok}$ the encoding of basic actions P (including structured compensating transactions) into a conversation calculus process $[\![P]\!]_{ok}$. The ok index represents the message label that signals the successful completion of the basic action, while abortion is signaled by throwing an exception. We only present the cases for the sequential composition $P; Q$ and for transactions $\langle T \rangle$; the other constructions are handled along standard lines (cf. Milner's encoding of concurrent programs [13]), using the termination signal ok to thread the flow graph. We also assume given some atomic actions A, implemented by certain CC processes A_{ok}.

The encoding of compensable transaction T is denoted by $[\![T]\!]_{ok,ab,cm,cb}$. The encoding of T will either issue a single message $ok^{\downarrow}$ to signal successful completion (and the implicit installation of compensation handlers) or (in exclusive alternative) a single message $ab^{\downarrow}$ to signal abortion. After successful completion, reception of a single message $cm^{\downarrow}$ ("compensate me") by the residual will trigger the compensation process. When compensation terminates, a single message $cb^{\downarrow}$ ("compensate back") will be issued, to possibly trigger compensation of previous successfully terminated activities.

We prove that our encoding is correct, by showing that it induces a compensating model in the general sense of Definitions 3.1 and 3.2. To that end, we introduce a few technical notions related to CC processes. A *passive process* is a process that can only perform an input labeled transition as a first possible action (no τ or output). We will consider the (stable) states of our CC compensating model to be passive in this sense, they should not exhibit any relevant autonomous activity, but may start interacting on client demand. After a transaction successfully terminates or aborts, the system should again reach a passive state. This reflects a view where clients engage into transactions with an external environment populated by waiting servers, and where the resources manipulated by a transaction are not subject to concurrent interference by other processes. For the CC compensating model we adopt a simple notion of apartness based on non-interference. We first define a relation of barb observation by $P \Downarrow x\ l \triangleq x \in \mathit{fn}(P)$ and $P \stackrel{\lambda^*\ x\ \sigma_l}{\Longrightarrow}$. Intuitively, $P \Downarrow x\ l$ means that P may interact via a message labeled by l in the free context name x. Then two processes are apart if they will not get to interact.

Definition 4.1 (CC Apartness). *Let $P\#Q$ be the largest symmetric relation on* CC *processes defined by $P\#Q \triangleq$ If $P \Downarrow x\ l$ and $Q \Downarrow y\ l$ then $x \neq y$.*

More flexible notions of apartness could be defined, but the one proposed here is enough to enforce the required locality principle, and already seems useful. We now introduce

Definition 4.2 (CC CM - Static structure). *The static structure of a* CC *compensating model is given by a tuple $\mathcal{S} = (S, \mid, \#, \bowtie)$ where*

- *$\bowtie$ is a congruence on* CC *processes containing bisimilarity*
- *S is a $\bowtie$-closed set of passive* CC *processes*
- *$\mid$ is parallel composition of* CC *processes*
- *$\#$ is apartness of* CC *processes*

Recall that composition $P \mid Q$ is defined in $\mathcal{S}$ only if $P\#Q$. We consider processes in S up to structural congruence, defined by the commutative monoid laws for $\mid$ and scope extrusion. We now characterize the dynamic structure of CC compensating models. Instead of defining a fixed set of basic activities, we specify in behavioral terms what we consider to be an atomic activity in a CC compensating model. We expect basic programs to be interpreted by atomic activities.

Definition 4.3 (CC Atomic Activity). *An atomic activity for a* CC *compensating model with structure $\mathcal{S} = (S, \mid, \#, \bowtie)$ is a* CC *process P such that, for all $Q \in S$,*

- *For all processes R, if $Q \mid P \Rightarrow R$ then there is Q' such that $R \Rightarrow\!\!\stackrel{\lambda}{\longrightarrow} Q'$ where either $\lambda = ok^{\downarrow}!$ or $\lambda = \mathtt{throw}$.*

– *If* $Q \mid P \Rightarrow \xrightarrow{\lambda} R$ *and* $\lambda = a^{\downarrow}$ *then* $a = ok^{\downarrow}!$ *and* $R \in S$.
– *If* $Q \mid P \Rightarrow \xrightarrow{\texttt{throw}} R$ *then* $Q \bowtie R$.

Intuitively, an atomic activity exercises capabilities (or services) provided from a state Q of S. It either successfully terminates, signaling such condition by sending an ok message in the current conversation context and leaving the system in a state of S, or fails, throwing an exception and leaving the system in a state of S $\bowtie$-equivalent to the initial one. The first condition ensures that either success or failure will happen (we assume that no divergence or deadlock occur). Moreover, since such atomic activity P may only throw an exception or drop a message ok in its enclosing conversation context, it is a consequence of the CC semantics that it may be embedded in any computational context while preserving the intended atomic conditions, so the restriction to parallel computational contexts in the Definition 4.3 does not bring any loss of generality.

Definition 4.4 (CC CM - Dynamic structure). *The dynamic structure of a* CC *compensating model is given by a pair* $\mathcal{D} = (\Sigma, \xrightarrow{a})$ *where*

– Σ *is a set of* CC *atomic activities*
– *For any* $A \in \Sigma$ *and* $P, Q \in S$ *define* $P \xrightarrow{A} Q \triangleq P \mid A \Rightarrow \xrightarrow{ok^{\downarrow}!} Q$.

Notice that activity *skip* is represented by the CC process $ok^{\downarrow}!$ and *throw* by the CC process $\texttt{throw}.\mathbf{0}$. Moreover, $P \not\xrightarrow{A}$ if and only if $P \mid A \Rightarrow \xrightarrow{\texttt{throw}} Q$ and $Q \bowtie P$.

We can now state our main Theorem 4.5, asserting the correctness of our encoding. It states that the mapping $[\![-]\!]_{ok}$ yields a sound embedding of arbitrary ($\bowtie$-consistent) structured compensating transactions in any CC compensating model.

Theorem 4.5 (Correctness). *Let* $\mathcal{S} = (S, \mid, \#, \bowtie)$ *and* $\mathcal{D} = (\Sigma, \xrightarrow{a})$ *define a* CC *compensating model* $\mathcal{M} = (\mathcal{S}, \mathcal{D})$. *If* $\langle T \rangle$ *is a* $\bowtie$*-consistent* CC *program over* Σ, *then* $[\![\langle T \rangle]\!]_{ok}$ *is a* CC *atomic activity, that either behaves as* T^{+}, *or aborts without any observable behavior modulo* $\bowtie$.

The proof builds on the developments in Section 3, in particular on the fact that our encoding satisfies the operational principles required by a cCSP interpretation in any compensating model (Definition 3.2), as formalized by the following technical Lemma.

Lemma 4.6. *Let* $\mathcal{S} = (S, \mid, \#, \bowtie)$ *and* $\mathcal{D} = (\Sigma, \xrightarrow{a})$ *define a* CC *compensating model* $\mathcal{M} = (\mathcal{S}, \mathcal{D})$. *Let* B *and* C *be the relations* $(-)^{P}_{R}\ d$ *and* $(-)^{P}_{R} \xleftarrow{d} -$ *defined on* CC *processes as follows:*

– $(A)^{P}_{R} * \triangleq P \mid [\![A]\!]_{ok} \Rightarrow \xrightarrow{\texttt{throw}} R$ *and* $R \bowtie P$
– $(A)^{P}_{R}\ \checkmark \triangleq P \mid [\![A]\!]_{ok} \Rightarrow \xrightarrow{ok^{\downarrow}!} R$ *and* $R \in S$.
– $(T)^{P}_{R} \xleftarrow{*} skip \triangleq P \mid [\![T]\!]_{ok,ab,cm,cb} \Rightarrow \xrightarrow{ab^{\downarrow}!} R$ *and* $R \bowtie P$
– $(T)^{P}_{R} \xleftarrow{\checkmark} Q \triangleq P \mid [\![T]\!]_{ok,ab,cm,cb} \Rightarrow \xrightarrow{ok^{\downarrow}!} \sim R \mid [\![Q]\!]_{cm,cb}$ *and* $R \in S$.

Then, the relations B *and* C *thus defined are a* cCSP *interpretation for* $\mathcal{M}$.

$$
\begin{array}{ll}
\mathit{Trans} & \triangleq \langle(\mathsf{bookFlight} \div \mathsf{cancelFlight}); \mathsf{bookHotel} \div \mathit{skip}\rangle \\
\mathsf{bookFlight} & \triangleq \mathbf{new}\ \mathit{Airline} \cdot \mathit{book} \Leftarrow \\
 & \qquad (\mathit{flight}!().\mathit{ack}?().\mathit{pay}!().\mathit{ok}^{\uparrow}!()) \\
\mathsf{cancelFlight} & \triangleq \mathbf{new}\ \mathit{Airline} \cdot \mathit{cancel} \Leftarrow \\
 & \qquad (\mathit{flight}!().\mathit{ack}?().\mathit{refund}!().\mathit{ok}^{\uparrow}!() \\
\mathsf{bookHotel} & \triangleq \mathbf{new}\ \mathit{Hotel} \cdot \mathit{book} \Leftarrow \\
 & \qquad (\mathit{room}!().\mathit{ack}?(\mathit{rp}).\mathbf{if}\ \mathit{rp}\ \mathbf{then}\ \mathit{pay}!().\mathit{ok}^{\uparrow}!()\ \mathbf{else}\ \mathbf{throw}) \\
\mathit{AirlineSite} & \triangleq \mathit{Airline} \blacktriangleright [\ \mathbf{def}\ \mathit{book} \Rightarrow - \mid \mathbf{def}\ \mathit{cancel} \Rightarrow - \mid \cdots] \\
\mathit{HotelSite} & \triangleq \mathit{Hotel} \blacktriangleright [\ \mathbf{def}\ \mathit{book} \Rightarrow - \mid \cdots] \\
\mathit{World} & \triangleq \mathit{AirlineSite} \mid \mathit{HotelSite} \\
\mathit{System} & \triangleq \mathit{World} \mid \mathit{Trans}
\end{array}
$$

Fig. 10. A travel booking example with compensable transactions in CC.

Proof. Induction on the defining rules for a cCSP interpretation (Definition 3.2). Notice that the auxiliary encoding $[\![Q]\!]_{cm,cb}$ (Figure 9) coerces a basic program Q into a thunk that may be activated by a message cm, and signals termination by a message cb. ∎

As a consequence of Lemma 4.6, all results in Section 3, in particular Theorem 3.9 apply to any CC compensation model. Indeed, our main Theorem 4.5 is just a specialization of Theorem 3.9 for any concrete CC compensation model.

We now illustrate our encoding and associated results with a simple example. Here, we freely mix cCSP transactions T inside the CC code as an abbreviation of the intended encoding $[\![T]\!]_{ok}$. Consider the CC definitions in Fig. 10 (we assume a standard $\mathbf{if} - \mathbf{then} - \mathbf{else}$ construct for the example). Trans represents a long running transaction for booking a flight and hotel accommodation. The transaction is defined from the CC basic activities bookFlight, cancelFlight and bookHotel. Each of these activities is implemented by service instantiations that trigger a conversation (a session) with an appropriate service provider. Such service providers reside in process World. The set S_W of passive processes accessible (via labeled transitions) from World can be presented as a CC compensating model, by introducing an appropriate equivalence $\bowtie$ on states. Such equivalence might e.g. equate states of $\mathit{AirlineSite}$ that differ on the total payment amount and back log state, but not on the travels effectively sold (in many situations we may just take $\bowtie = \sim$). Then Theorem 4.5 allows us to conclude that if CancelFlight and BookFlight are atomic activities in the sense of Definition 4.3 and if cancelFlight reverts BookFlight in the sense of Definition 3.4 (so that Trans is $\bowtie$-consistent), then the two following conditions hold

- For all $S \in S_W$, if $S \mid \mathit{Trans} \Rightarrow \xrightarrow{\lambda} R$ and $\lambda = a^{\downarrow}$ then $a = \mathit{ok}^{\downarrow}!$ and $R \in S_W$.
- For all $S \in S_W$, if $S \mid \mathit{Trans} \Rightarrow \xrightarrow{\mathtt{throw}} R$ then $R \bowtie S$.

By Theorem 4.5, we conclude Trans either behaves as Trans^{+} (bookFlight; bookHotel) or throws an exception without any observable effect modulo $\bowtie$. Notice that since Trans local behavior may only result on an exception being thrown or on a single message ok being dropped in its conversation context. It is a consequence of the CC semantics that it will always exhibit such local behavior locally inside any arbitrary client computational context $\mathcal{C}\,[\mathit{Trans}]$ not interfering with the states of S_W.

5 Related Work and Concluding Remarks

We have developed an analysis of structured compensating transactions, with an application to models of service-oriented systems expressed in the Conversation Calculus [17, 16], a session based process calculus for service-oriented computing [11, 3, 2].

Our approach is inspired by Korth *et al.* [10], where compensating transactions have been introduced as a way to overcome the limitations of atomicity when dealing with long-running transactions. The authors propose the use of compensating transactions to allow access to uncommitted data and to undo committed transactions in databases. More recently, Butler *et al.* [7] defined a framework to reason about compensation soundness. The authors define a cancellation semantics for a structured language cCSP based on a cancellation function that processes traces, extracting reversible forward and compensation actions from process traces. Their approach does not explicitly characterize reversibility in terms of state equivalence, and is restricted to compensation pairs of primitive actions, while with our approach transactions may be used as atomic actions in compensable programs. Furthermore, our approach builds on a generic model that may be used to reason about other compensable languages than cCSP. Several process calculi, both flow and interactions based, have been proposed with support for forms of compensation and primitives for handling the unexpected [6, 1, 4, 5, 12].

In this work, besides showing how to encode compensating transactions from lower level constructs in the Conversation Calculus, we also put forward a general framework that may be used to reason about the correctness (e.g., atomicity) of arbitrary compensating transactions in arbitrary programs. In order to perform such an analysis, one just needs to define an appropriate equivalence on system states, which is in general application dependent, and to show reversibility for the considered alphabet of atomic actions. Our framework naturally supports distributed transactions (since basic actions may be realized by service calls), and may benefit from the use of types (for example, it would be interesting to consider state equivalence relations based on behavioral types). It would also be challenging to study more refined notions of apartness and locality, allowing for extended yet safe forms of interference between concurrent transactions.

Acknowledgments We thank CITI, IST FP6 IP Sensoria, and our coleagues of the Sensoria project.

References

1. L. Bocchi, C. Laneve, and G. Zavattaro. A Calculus for Long-Running Transactions. In 2884 of *LNCS*, *FMOODS*, pages 124–138, 2003.
2. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: a Service Centered Calculus. In 4184 of *LNCS*, *WS-FM*, pages 38–57, 2006.
3. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In 5051 of *LNCS*, *FMOODS*, pages 19–38, 2008.
4. R. Bruni, H. C. Melgratti, and U. Montanari. Nested Commits for Mobile Calculi: Extending Join. In Kluwer Academics, *IFIP TCS*, pages 563–576, 2004.
5. R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. In *POPL*, pages 209–220, 2005.

6. M. Butler and C. Ferreira. A Process Compensation Language. In 1945 of *LNCS*, *IFM*, pages 61 – 76, 2000.
7. M. J. Butler, C. A. R. Hoare, and C. Ferreira. A Trace Semantics for Long-Running Transactions. In 3525 of *LNCS*, *25 Years CSP*, pages 133–150, 2004.
8. I. Lanese C. Guidi, F. Montesi and G. Zavattaro. Dynamic Fault Handling for Service Oriented Applications. In *Proceedings of ECOWS*, 2008.
9. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
10. H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *16th VLDB Conference*, 1990.
11. I. Lanese, V. T. Vasconcelos, F. Martins, and A. Ravara. Disciplining Orchestration and Conversation in Service-Oriented Computing. In *5th ICSEFM*, pages 305–314. IEEE Computer Society Press, 2007.
12. C. Laneve and G. Zavattaro. Foundations of Web Transactions. In 3441 of *LNCS*, *FoSSaCS*, pages 282–298, 2005.
13. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
14. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I + II. *Information and Computation*, 100(1):1–77, 1992.
15. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
16. H. T. Vieira, L. Caires, and J. C. Seco. A Model of Service Oriented Computation. TR-DI/FCT/UNL 6/07, Universidade Nova de Lisboa, Departamento de Informatica, 2007.
17. H. T. Vieira, L. Caires, and J. C. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In 4960 of *LNCS*, ESOP, pages 269–283, 2008.