Transactional Service Level Agreement*

Maria Grazia Buscemi and Hernán Melgratti

IMT, Lucca Institute for Advanced Studies, Italy.
m.buscemi@imtlucca.it, h.melgratti@imtlucca.it

Abstract. Several models based on process calculi have addressed the definition of linguistic primitives for handling long running transactions and Service Level Agreement (SLA) in service oriented applications. Nevertheless, the approaches appeared in the literature deal with these aspects as independent features. We claim that transactional mechanisms are relevant for programming multi-step SLA negotiations and, hence, it is worth investigating the interplay among such formal approaches. In this paper we propose a process calculus, the *committed cc-pi*, that combines two proposals: (i) cc-pi calculus accounting for SLA negotiation and (ii) cJoin as a model of long running transactions. We provide both a small- and a big-step operational semantics of committed cc-pi as labelled transition systems, and we prove a correspondence result.

1 Introduction

Service Oriented Computing (SOC) is an emerging paradigm in distributed computing. Services are autonomous computational entities that can be described, published, and dynamically discovered for developing distributed, interoperable applications. Along with functional properties, services may expose non-functional properties including Quality of Service (QoS), cost, and adherence to standards. Non-functional parameters play an important role in service discovery and binding as, e.g., multiple services able to fulfill the same user request (because they provide the same functionality) can still be differentiated according to their non-functional properties. Service Level Agreements (SLAs) capture the mutual responsibilities of the provider of a service and of its client with respect to non-functional properties, with emphasis on QoS values.

The terms and conditions appearing in a SLA contract can be negotiated among the contracting parties prior to service execution. In the simplest case, one of the two parties exposes a contract template that the other party must fill in with values in a given range; in case of failure, no agreement is reached and a new negotiation must be initiated. However, in general, arbitrary complex scenarios involving distributed transactions may occur: (i) third parties may take part to or just exert some influence on a negotiation, (ii) negotiations can be nested, (iii) if a commit cannot be achieved, compensation mechanisms may be activated, e.g. clients may relax their own SLA requirements and providers may add further service guarantees until an agreement is reached.

Several approaches have appeared in the literature for specifying and enforcing SLA contracts [14, 10, 7, 2] and for modelling and analysing long running transactions in the

* Research supported by the FET-GC II Project IST-2005-16004 SENSORIA and by the Italian FIRB Project TOCAI.IT

context of name passing calculi [5, 11, 4]. However, such theories treat these two issues as independent features. By contrast, we claim that transactions can be conveniently employed for programming SLA negotiation scenarios. In this paper, we propose the *committed cc-pi calculus (committed cc-pi)*, a language for specifying SLAs that also features coordination primitives tailored to multi-party negotiations. More specifically, committed cc-pi extends cc-pi [7] with the transactional mechanism of cJoin[5] for handling commits and aborts of negotiations along with possible activations of compensations. We remind that, unlike compensatable flows [8, 6], the approaches in [5, 11, 4] rely on a notion of compensation that is essentially an exception handling mechanism.

The cc-pi calculus [7] is a simple model for SLA contracts inspired by two basic programming paradigms: name-passing calculi (see e.g. [12]) and concurrent constraint programming [13]. More in detail, cc-pi combines synchronous communication and a restriction operation *à la* process calculi with operations for creating, removing and making logical checks on constraints. The constraint systems employed in cc-pi are based on the *c-semiring* structures [3], which are able to model networks of constraints for defining constraint satisfaction problems and to express fuzzy, hierarchical, or probabilistic values.

cJoin [5] is an extension of the Join calculus [9] with primitives for distributed nested commits. The two key operations of cJoin are: the "abort with compensation", to stop a negotiation and activate a compensating process; and the "commit", to store a partial agreement among the parties before moving to the next negotiation phase.

Before introducing committed cc-pi, we single out the transactional primitives of cJoin and add them to the pi-calculus. This intermediate step highlights the interplay of compensating transactions with a channel-based interaction mechanism that is different from Join and it is intended to make the treatment of constraints easier to understand.

Synopsis. In §2 we highlight the main features of cc-pi, and in §3 we briefly recall cJoin and we present a transactional extension of the pi-calculus inspired by cJoin. In §4 we introduce the committed cc-pi calculus by giving its syntax and operational semantics in terms of labelled transition system and we show some examples of modelling transactional SLA negotiations. In §5 we define a big-step semantics of committed cc-pi and we prove a correspondence result.

2 Constrained-based SLA negotations

The cc-pi calculus integrates the Pi-F calculus [15] with a constraint handling mechanism. The Pi-F calculus is a variant of the pi-calculus whose synchronisation mechanism is global and, instead of binding formal names to actual names, it yields an *explicit fusion*, i.e., a simple constraint expressing the equalities of the transmitted parameters. cc-pi extends Pi-F by generalising explicit fusions to arbitrary constraints and by adding primitives like `tell` and `ask`, which are inspired by the constraint-based computing paradigms. We defer a technical treatment of the syntax and semantics of the cc-pi to §4, where we will give a formal presentation of an extended version of cc-pi including transactional features. Here, we simply overview the main principles of the calculus.

Underlying constraint system. The cc-pi calculus is parametric with respect to *named constraints*, which are meant to model different SLA domains. Consequently, it is not necessary to develop ad hoc primitives for each different kind of SLA to be modelled. A named constraint c is defined in terms of c-semiring structures and comes equipped with a notion of *support* $\mathrm{supp}(c)$ that specifies the set of "relevant" names of c, i.e. the names that are affected by c. The notation $c(x,y)$ indicates that $\mathrm{supp}(c) = \{x,y\}$. In this work, we leave such underlying theory implicit and we refer the interested reader to [7, 3]. For our purposes, we simply assume usual notions of entailment relation ($\vdash$), of combination of constraints ($\times$) and of consistency predicate (see e.g. [13]). Moreover, we will only consider *crisp* constraints (instead of the more general *soft* constraints), i.e. we will assume a constraint system leading to solutions consisting of a set of tuples of legal domain values. As an example the constraint $c(x,y) = (7 \leq x \leq 9) \times (15 \leq y \leq 18)$ specifies that the names x and y can only assume domain values in the range $[7,\ldots,9]$ and $[15,\ldots,18]$. Assuming a constraint $d(x,y) = (6 \leq x \leq 8) \times (17 \leq y \leq 19)$, the result of combining c and d is the intersection of their respective possible values, i.e. the constraint $e(x,y) = c(x,y) \times d(x,y) = (7 \leq x \leq 8) \times (17 \leq y \leq 18)$. We say a constraint to be inconsistent when it has no tuples, and we write 0 for the inconsistent constraint.

In cc-pi, the parties involved in a negotiation are modelled as communicating processes and the SLA guarantees and requirements are expressed as constraints that can be generated either by a single process or as a result of the synchronisation of two processes. Moreover, the restriction operator of the cc-pi calculus can limit the scope of names thus allowing for local stores of constraints, which may become global after a synchronisation. A single process $P = \texttt{tell}\ c.Q$ can place a constraint c corresponding to a certain requirement/guarantee and then evolve to process Q. Alternatively, two processes $P = \overline{p}\langle\tilde{x}\rangle.P'$ and $Q = p\langle\tilde{y}\rangle.Q'$ that are running in parallel $(P \mid Q)$ can synchronise with each other on the port p by performing the output action $\overline{p}\langle\tilde{x}\rangle$ and the input action $p\langle\tilde{y}\rangle$, respectively, where $\tilde{x}$ and $\tilde{y}$ stand for sequences of names. Such a synchronisation creates a constraint induced by the identification of the communicated parameters $\tilde{x}$ and $\tilde{y}$, if the store of constraints obtained by adding this new constraint is consistent, otherwise the system has to wait that a process removes some constraint (action $\texttt{retract}\ c$).

Example 1. Consider a user that is looking for a web hosting solution with certain guarantees about the supplied bandwidth and cost. We assume the client and the provider to communicate over channel r the information about the requested bandwidth, and over channel p the information about the price of the service. The constant rb stands for the minimal bandwidth accepted by the client, while ob represents the maximal bandwidth offered by the provider. Moreover, the provider fixes the price uc as the cost for any unit of bandwidth, and the client the maximal cost c it is intended to pay for the service. Then, the whole system can be modelled by the following two processes: one describing the behaviour of the client $\mathsf{Client}_{\mathsf{rb},\mathsf{c}}(r,p)$ and the other the provider $\mathsf{Provider}_{\mathsf{ob},\mathsf{uc}}(r,p)$.

$$\mathsf{Client}_{\mathsf{rb},\mathsf{c}}(r,p) \equiv (bw)(cost)(\texttt{tell}\ (bw \geq \mathsf{rb}).\overline{r}\langle bw\rangle.\texttt{tell}\ (cost \leq \mathsf{c}).p\langle cost\rangle)$$

$$\mathsf{Provider}_{\mathsf{ob},\mathsf{uc}}(r,p) \equiv (bw')(cost')(\texttt{tell}\ (bw' \leq \mathsf{ob}).r\langle bw'\rangle.\texttt{tell}\ (bw' * \mathsf{uc} = cost').\ \overline{p}\langle cost'\rangle)$$

The client starts by fixing the constraint about the minimal requested bandwidth, then it contacts the provider by communicating on channel r and, after that, it fixes the maximal cost it can afford and synchronises on p. The provider behaves similarly, by fixing an upper bound ob on the offered bandwidth, accepting a request from the client over r and, then, fixing the cost of the service and synchronising with the client.

Consider the following system composed of a client and two providers.

$$\mathsf{S} \equiv (r)(p)\mathsf{Client}_{\mathsf{4Gb},\$100} \mid \mathsf{Provider}_{\mathsf{6Gb},\$20} \mid \mathsf{Provider}_{\mathsf{3Gb},\$15}$$

The client requests at least 4 Gigabytes (Gb), while one provider offers at most 6Gb and the other 3Gb. As expected, after each party has placed its own constraint on the required/offered bandwidth, the client can synchronize only with the first provider (the interaction with the second one is not possible since the constraints $bw \geq \mathsf{4Gb}$, $bw' \leq \mathsf{3Gb}$, $bw = bw'$ are inconsistent). As a next step, the first provider and the client fix the constraints about the cost of the service, the synchronisation over p takes place, and the negotiation succeeds. The released constraints are the agreed parameters of the contract. If we consider a domain of integer solutions, the contract is either the solution $bw = bw' = \mathsf{4Gb}$ and $cost = cost' = \$80$, or $bw = bw' = \mathsf{5Gb}$ and $cost = cost' = \$100$. Note that `tell` prefixes handle local stores of constraints, while synchronisations allow to identify variables belonging to different stores, thus yielding a global store.

3 Compensating transactions

cJoin is a process calculus that provides programming primitives for handling transactions among interacting processes. The communication primitives of cJoin are inherited from the Join calculus [9], which is a process calculus with asynchronous name-passing communication, while the transactional mechanism is based on compensations, i.e., partial execution of transactions are recovered by executing user-defined programs instead of providing automatic roll-back. So, in addition to the usual primitives of Join, cJoin provides a new kind of terms of the form $[P : Q]$, denoting a process P that is required to execute until completion. In case P cannot successfully complete, i.e., when P reaches the special process **abort**, the corresponding compensation Q is executed.

The main idea in cJoin is that transaction boundaries are not permeable to ordinary messages, so that a transactional process $[P : Q]$ can only compute locally. However, a limited form of interaction is allowed with other transactional processes: in this case, after the interaction, the transactional processes become part of the same larger transaction, and hence all parties should reach the same agreed outcome, i.e., if some party commits (resp. aborts) then all of them commit (resp. aborts).

Rather than providing the formal definition of cJoin, below we focus on its transactional primitives. To this purpose, we present *committed pi*, an extension of the pi-calculus with the transaction mechanism introduced in cJoin. This choice aims to show the interplay of compensating transactions with the channel-based process communication of pi-calculus (and of cc-pi), thus making the transactional extension of cc-pi presented in §4 more straightforward.

3.1 From cjoin to committed pi

Assume an infinite, countable set of names $\mathcal{N}$, ranged over by $a, b, x, y, z, \ldots$ and a set of process identifiers, ranged over by D. The syntax of committed pi processes is given in Figure 1(a). As in the pi-calculus, a process is either the inert process $\mathbf{0}$, the parallel composition $P|P'$ of two processes, a guarded choice $\Sigma_i \alpha_i.P_i$ where $\alpha_i.P_i$ is either (i) an agent $x(\widetilde{y}).P$ that accepts a message on channel x and then continues as P, (ii) the synchronous emission of a message $\overline{x}\langle\widetilde{y}\rangle$ with continuation P, or (iii) the internal choice $\tau.P$. The process $(\nu x)P$ defines the private channel x. A defining equation for a process identifier D is of the form $D(\tilde{x}) \stackrel{\text{def}}{=} P$ where the free names of P are included in $\tilde{x}$. Then, for any process $D(\tilde{y})$ we require $|\tilde{x}| = |\tilde{y}|$. In addition, committed pi provides two primitives for handling transactions: $[P : Q]$ for defining a transactional process P with compensation Q, and **abort** for indicating an aborted transaction.

We write $(\nu x_1 \ldots x_n)P$ as an abbreviation for $(\nu x_1) \ldots (\nu x_n)P$. When $\widetilde{x} = x_1 \ldots x_n$ and $n = 0$, $(\nu\widetilde{x})P$ stands for P. We abbreviate $\overline{z}\langle\widetilde{y}\rangle.\mathbf{0}$ by $\overline{z}\langle\widetilde{y}\rangle$ and we write M for a process consisting only on sent messages, i.e. $M = \overline{x_1}\langle\widetilde{y_1}\rangle | \ldots | \overline{x_n}\langle\widetilde{y_n}\rangle$. M is $\mathbf{0}$ when $n = 0$. The reduction semantics is the least relation satisfying the rules in Figure 1(c) (modulo the the usual structural equivalence rules in Figure 1(b)). Free and bound names (written $fn(P)$ and $bn(P)$) are defined as usual.

Rules (COMM), (TAU), (PAR), and (RES) are the standard ones for the synchronous pi-calculus. Rule (TRANS) describes the internal computations of a transactional process, in which the compensation Q is kept frozen. Rule (TR-COMP) handles the case of a transaction that aborts, which causes the remaining part of the transactional process to be removed and the associated compensation Q to be activated. Instead, rule (COMMIT) defines the behaviour of a transaction that commits. A transaction commits when it produces a set of output messages M, each of them followed by $\mathbf{0}$, i.e., there are no remaining computation inside the transaction. At this point, all produced messages M are released and the associated compensation is discarded. Last rule (TR-COMM) describes the interaction among two transactions. In particular, when one transactional process sends a message that is received by another transactional process both transactional scopes are merged into a larger one containing the remaining parts of the original transactions and its compensation is the parallel composition of the original ones.

Example 2. Consider the typical scenario in which a user books a room through a hotel reservation service. The ideal protocol can be sketched as below.

$$\mathsf{C} \equiv \overline{request}\langle data\rangle.offer(price).\overline{accept}\langle cc\rangle$$
$$\mathsf{H} \equiv request(details).\overline{offer}\langle rate\rangle.accept(card)$$

The client starts by sending a booking request to the hotel, which answers it with a rate offer. After receiving the offer, the client accepts it. Nevertheless, there are several situations in which parties may be forced/inclined not to complete the execution of the protocol (e.g., the hotel has no available rooms for the requested day, or the client does not obtain convenient rates).

$$\mathsf{Client} \equiv [\overline{request}\langle data\rangle.offer(price).(\overline{accept}\langle cc\rangle + \tau.\mathbf{abort}) : alt(h).Q]$$
$$\mathsf{Hotel} \equiv [request(details).(\overline{offer}\langle rate\rangle.accept(card) + \tau.\mathbf{abort}) : \overline{alt}\langle hotel\rangle]$$

$$P ::= \mathbf{0} \mid P|P \mid \Sigma_i \alpha_i.P_i \mid (\nu x)P \mid D(\tilde{y}) \mid [P:Q] \mid \mathbf{abort}$$
$$\alpha ::= x(\tilde{y}) \mid \overline{x}\langle\tilde{y}\rangle \mid \tau$$

(a) Syntax

$$P \mid \mathbf{0} \equiv P \qquad P \equiv Q \quad \text{if } P \equiv_\alpha Q$$
$$P \mid Q \equiv Q \mid P \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad (\nu x)P \mid Q \equiv (\nu x)(P|Q) \ \text{ if } x \notin fn(Q)$$
$$P + Q \equiv Q + P \qquad (P+Q)+R \equiv P+(Q+R)$$
$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad D(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\} \quad \text{if } D(\tilde{x}) \stackrel{\text{def}}{=} P$$

(b) Structural equivalence

(COMM)
$$x(\widetilde{y}).P + P' \mid \overline{x}\langle\widetilde{z}\rangle.Q + Q' \ \rightarrow\ P\{\widetilde{z}/\widetilde{y}\}|Q$$

(TAU)
$$\tau.P + Q \ \rightarrow\ P$$

(PAR)
$$\frac{P \ \rightarrow\ P'}{P|Q \ \rightarrow\ P'|Q}$$

(RES)
$$\frac{P \ \rightarrow\ P'}{(\nu x)P \ \rightarrow\ (\nu x)P'}$$

(TRANS)
$$\frac{P \ \rightarrow\ P'}{[P:Q] \ \rightarrow\ [P':Q]}$$

(TR-COMP)
$$[\mathbf{abort}|P:Q] \ \rightarrow\ Q$$

(COMMIT)
$$[M:Q] \ \rightarrow\ M$$

(TR-COMM)
$$[(\nu\widetilde{x})y(\widetilde{v}).P_1 + R_1 \,|\, P_1' \,:\, Q_1] \mid [(\nu\widetilde{z})\overline{y}\langle\widetilde{w}\rangle.P_2 + R_2 \,|\, P_2' \,:\, Q_2] \ \rightarrow\ [(\nu\widetilde{x}\widetilde{z})P_1\{\widetilde{w}/\widetilde{v}\} \,|\, P_1' \,|\, P_2 \,|\, P_2' \,:\, Q_1 \,|\, Q_2]$$
$$\text{if } y \notin \widetilde{x} \cup \widetilde{z} \text{ and } \widetilde{x} \cap fn(P_2|P_2') = \emptyset \text{ and } \widetilde{z} \cap fn(P_1|P_1') = \emptyset$$

(c) Reduction Semantics

Fig. 1. Syntax and Semantics of the committed pi calculus

The above protocol allows the client to abort the transaction after receiving an offer (for instance when the offer does not satisfy her expectations). Alternatively, the hotel may abort after receiving a request (for instance when no rooms are available). We illustrate the use of compensations by making the component Hotel to generate the single message $\overline{alt}\langle hotel\rangle$ to provide the client with an alternative hotel to contact. The compensation of Client is a process that receives a message on the port *alt* and then behaves like Q, which stands for the process that contacts the alternative hotel h.

The process Client|Hotel behaves in committed pi as follows. When both transactions communicate through the port *request* for the first time they are merged in a unique larger transaction, whose transactional process and compensation correspond respectively to the parallel composition of the residuals of the original transactions and to the parallel composition of the original compensations, as shown below

$$\begin{aligned}\mathsf{Client}|\mathsf{Hotel} \rightarrow\ & [\mathit{offer}(\mathit{price}).(\overline{\mathit{accept}}\langle cc\rangle + \tau.\mathbf{abort}) \\ & \mid (\overline{\mathit{offer}}\langle rate\rangle.accept(card) + \tau.\mathbf{abort}) : alt(h).Q \mid \overline{alt}\langle hotel\rangle]\end{aligned}$$

¿From this moment the system may evolve as usual. Assuming the hotel sends an offer and the client replies with a confirmation, the system commits the transaction as follows

$$\begin{aligned}&\rightarrow [(\overline{accept}\langle cc\rangle \ +\ \tau.\mathbf{abort}) \mid accept(card) : alt(h).Q \mid \overline{alt}\langle hotel\rangle] \\ &\rightarrow [\mathbf{0} : alt(h).Q \mid \overline{alt}\langle hotel\rangle] \\ &\rightarrow \mathbf{0}\end{aligned}$$

Otherwise, if we assume that the client refuses the offer then the system evolves as follows and activates the compensation of both parties.

$$\rightarrow [(\overline{accept}\langle cc\rangle + \tau.\mathbf{abort}) \mid accept(card) : alt(h).Q \mid \overline{alt}\langle hotel\rangle]$$
$$\rightarrow alt(h).Q \mid \overline{alt}\langle hotel\rangle$$

4 Committed cc-pi

In this section we enrich cc-pi with the transactional mechanism described above. Before introducing the extended calculus, we show an example that motivates the addition of compensating transactions for modelling SLA negotiations.

Example 3. Consider the system shown in Example 1. Suppose that the client is intended to pay a maximal cost \$60 instead of \$100. The new system is as follows.

$$\mathsf{S}' \equiv (r)(p)\mathsf{Client}_{\mathsf{4Gb,\$60}} \mid \mathsf{Provider}_{\mathsf{6Gb,\$20}} \mid \mathsf{Provider}_{\mathsf{3Gb,\$15}}$$

The evolution of S' is as in the original system until the first provider and the client place their own constraints on the cost (as before the client cannot synchronise with the other provider). Then, the negotiation between the client and the first provider fails, because the constraints $cost \leq \mathsf{c}$, $bw' \leq \mathsf{ob}$ and $cost = cost'$ are inconsistent, and the system is stuck. Later in §4.3, we will see how to model this scenario using the transactional mechanism of committed cc-pi.

4.1 Syntax

The syntax of committed cc-pi processes is specified in Figure 2(a). Assume the infinite set of names $\mathcal{N}$, ranged over by $x, y, z, \ldots$ and a set of process identifiers, ranged over by D. We let c range over the set of constraints of an arbitrary named c-semiring $\mathcal{C}$.

The syntax of the calculus is the same as for the cc-pi except for the inclusion of a transactional primitive which is inspired by committed pi. The τ prefix stands for a silent action, output $\overline{x}\langle\widetilde{y}\rangle$ and input $x\langle\widetilde{y}\rangle$ are complementary prefixes used for communications. Unlike other calculi, the input prefix is not binding, hence input and output operations are fully symmetric and the synchronisation of two complementary prefixes $x\langle\widetilde{y}\rangle$ and $\overline{x}\langle\widetilde{z}\rangle$, rather than binding $\widetilde{y}$ to $\widetilde{z}$, yields the name fusion $\widetilde{y} = \widetilde{z}$. Prefix `tell` c generates a constraint c and puts it in parallel with the other constraints, if the resulting parallel composition of constraints is consistent; `tell` c aborts otherwise. Prefix `ask` c is enabled if c is entailed by the set of constraints in parallel. Prefix `retract` c removes a constraint c, if c is present. *Unconstrained processes* U are essentially processes that can only contain constraints c in prefixes `tell` c, `ask` c, and `retract` c. As usual, $\mathbf{0}$ stands for the inert process and $U \mid U$ for the parallel composition. $\sum_i \pi_i.U_i$ denotes a mixed choice in which some guarded unconstrained process U_i is chosen when the corresponding guard π_i is enabled. Restriction $(x)U$ makes the name x local in U. A defining equation for a process $D(\tilde{y})$ is of the form $D(\tilde{x}) \stackrel{\text{def}}{=} U$ where $|\tilde{x}| = |\tilde{y}|$ and the free names of U must be included in $\tilde{x}$. The transaction primitive $[P : Q].U$ defines a

$$
\begin{array}{lrl}
\textit{Prefixes} & \pi ::= & \tau \mid \overline{x}\langle\tilde{y}\rangle \mid x\langle\tilde{y}\rangle \mid \texttt{tell}\ c \mid \texttt{ask}\ c \mid \texttt{retract}\ c \\
\textit{Unconstrained Processes} & U ::= & \mathbf{0} \mid U|U \mid \sum_i \pi_i.U_i \mid (x)U \mid D(\tilde{y}) \mid [P:Q].U \\
\textit{Constrained Processes} & P ::= & U \mid c \mid P|P \mid (x)P
\end{array}
$$

(a) Syntax

$$
\begin{array}{lll}
P|\mathbf{0} \equiv P & P+Q \equiv Q+P & (x)(y)P \equiv (y)(x)P \\
P|Q \equiv Q|P & (P+Q)+R \equiv P+(Q+R) & P|(x)Q \equiv (x)(P|Q) \ \textit{if}\ x \notin \mathrm{fn}(P) \\
(P|Q)|R \equiv P|(Q|R) & D(\tilde{y}) \equiv U\{\tilde{y}/\tilde{x}\} \ \textit{if}\ D(\tilde{x}) \stackrel{\mathrm{def}}{=} U & (x)\mathbf{0} \equiv \mathbf{0} \\
[(x)P:Q].U \equiv (x)[P:Q].U \ \ \textit{if}\ x \notin \mathrm{fn}(Q,U) & &
\end{array}
$$

(b) Structural equivalence

(TAU)
$$C\,|\,\tau.U \xrightarrow{\tau} C\,|\,U$$

(OUT)
$$C\,|\,\overline{x}\langle\tilde{y}\rangle.U \xrightarrow{\overline{x}\langle\tilde{y}\rangle} C\,|\,U$$

(INP)
$$C\,|\,x\langle\tilde{y}\rangle.U \xrightarrow{x\langle\tilde{y}\rangle} C\,|\,U$$

(TELL)
$$C\,|\,\texttt{tell}\ c.U \xrightarrow{\tau} C\,|\,c\,|\,U \ \text{ if } C\,|\,c \text{ consistent}$$

(ABT-TELL)
$$C\,|\,\texttt{tell}\ c.U \xrightarrow{\texttt{abr}} \mathbf{0} \ \text{ if } C\,|\,c \text{ not consistent}$$

(ASK)
$$C\,|\,\texttt{ask}\ c.U \xrightarrow{\tau} C\,|\,U \quad \text{if } C \vdash c$$

(RETRACT)
$$C\,|\,\texttt{retract}\ c.U \xrightarrow{\tau} (C-c)\,|\,U$$

(COMM)
$$\frac{C\,|\,U \xrightarrow{x\langle\widetilde{y}\rangle} C\,|\,U' \quad C\,|\,V \xrightarrow{\overline{z}\langle\widetilde{w}\rangle} C\,|\,V'}{C\,|\,U\,|\,V \xrightarrow{\tau} C\,|\,\widetilde{y}=\widetilde{w}\,|\,U'\,|\,V'} \quad \text{if } |\widetilde{y}| = |\widetilde{w}| \text{ and } C\,|\,\widetilde{y}=\widetilde{w} \text{ consistent and } C \vdash x = z$$

(ABT-COMM)
$$\frac{C\,|\,U \xrightarrow{x\langle\widetilde{y}\rangle} P \quad C\,|\,V \xrightarrow{\overline{z}\langle\widetilde{w}\rangle} Q}{C\,|\,U\,|\,V \xrightarrow{\texttt{abr}} \mathbf{0}} \quad \text{if } |\widetilde{y}| = |\widetilde{w}| \text{ and } C\,|\,\widetilde{y}=\widetilde{w} \text{ not consistent and } C \vdash x = z$$

(PAR)
$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \neq \texttt{abr}}{P\,|\,U \xrightarrow{\alpha} P'\,|\,U}$$

(ABT-PAR)
$$\frac{P \xrightarrow{\texttt{abr}} \mathbf{0}}{P\,|\,Q \xrightarrow{\texttt{abr}} \mathbf{0}}$$

(SUM)
$$\frac{C\,|\,\pi_i.U_i \xrightarrow{\alpha} U'}{C\,|\,\sum \pi_i.U_i \xrightarrow{\alpha} U'}$$

(RES)
$$\frac{P \xrightarrow{\tau} P'}{(x)P \xrightarrow{\tau} (x)P'}$$

(TRANS)
$$\frac{P \xrightarrow{\tau} P'}{[P:Q].U \xrightarrow{\tau} [P':Q].U}$$

(TR-COMP)
$$\frac{P \xrightarrow{\texttt{abr}} P'}{[P:Q].U \xrightarrow{\tau} Q}$$

(TR-COMMIT)
$$[C:Q].U \xrightarrow{\tau} C\,|\,U$$

(TR-PAR)
$$\frac{[P:Q].U \xrightarrow{\tau} P'}{C\,|\,[P:Q].U \xrightarrow{\tau} C\,|\,P'}$$

(TR-COMM)
$$\frac{C_1\,|\,U_1 \xrightarrow{\overline{x}\langle\widetilde{y}\rangle} R_1 \quad C_2\,|\,U_2 \xrightarrow{z\langle\widetilde{w}\rangle} R_2 \quad |\widetilde{y}| = |\widetilde{w}| \text{ and } C_1\,|\,C_2\,|\,\widetilde{y}=\widetilde{w} \text{ consistent and } C\,|\,C_1\,|\,C_2 \vdash x = z}{C\,|\,[C_1\,|\,U_1 : Q_1].V_1\,|\,[C_2\,|\,U_2 : Q_2].V_2 \xrightarrow{\tau} C\,|\,[R_1\,|\,R_2\,|\,\widetilde{y}=\widetilde{w} : Q_1\,|\,Q_2].(V_1\,|\,V_2)}$$

(c) Labelled Semantics

Fig. 2. Syntax and Small-Step Semantics of the committed cc-pi calculus

transactional process P which evolves to U in case of a commit, while otherwise activates the compensation Q. *Constrained processes* P are defined like unconstrained processes U but for the fact that P may have constraints c in parallel with processes. We simply write processes to refer to constrained processes.

4.2 Operational semantics

The *structural equivalence* relation $\equiv$ is defined as the least equivalence over processes closed with respect to α-conversion and satisfying the rules in Figure 2(b). Note that the notion of *free names* $\mathrm{fn}(P)$ of a process P is extended to handle constraints by stating that the set of free names of a constraint c is the support $\mathrm{supp}(c)$ of c. The structural axioms can be applied for reducing every process P into a normal form $(\tilde{x})\,(C\,|\,U)$, where C is a parallel composition of constraints and U can only contain restrictions under prefixes, i.e. $U \not\equiv (\widetilde{y})U'$.

Well-formedness. Let $Ch \subseteq \mathcal{N}$ be a set of *channel names* that can only be fused among each other and let $\mathrm{chn}(P)$ be the set of channel names occurring free in P. P is *well-formed* if there exists a process $Q \equiv P$ such that every occurrence of transaction in Q has the form $(x_1,\ldots,x_n)[P' : Q'].U$, where $(\mathrm{fn}(P',Q') \setminus \mathrm{chn}(P',Q')) \subseteq \{x_1,\ldots,x_n\}$. For example, $P \equiv (x)\,(\texttt{tell}\ (x=z)\,|\,(w)[\overline{y}\langle w\rangle.\mathbf{0} : Q].U)$ is well-formed, but $R \equiv (x)\,(\texttt{tell}\ (x=y)\,|\,[\overline{y}\langle x\rangle.\mathbf{0} : Q].U)$ is not. Hereafter, we assume all processes to be well-formed.

Let $A = \{\tau, \overline{x}\langle\tilde{y}\rangle, x\langle\tilde{y}\rangle, \texttt{abr}\,|\,x, y_i \in \mathcal{N}$ for $\tilde{y} = \langle y_1,\ldots,y_n\rangle\ \}$ be a set of labels and let α range over A. The labelled transition semantics of processes (taken up to structural equivalence $\equiv$) is the smallest relation $P \xrightarrow{\alpha} Q$, defined by the inference rules in Figure 2(c), where: C stands for the parallel composition of constraints $c_1\,|\,\ldots\,|\,c_n$; C *consistent* means $(c_1 \times \ldots \times c_n) \neq 0$; $C \vdash c$ if $(c_1 \times \ldots \times c_n) \vdash c$; $C - c$ stands for $c_1\,|\,\ldots\,|\,c_{i-1}\,|\,c_{i+1}\,|\,\ldots\,|\,c_n$ if $c = c_i$ for some i, while $C - c = C$ otherwise.

The choice of giving a labelled transition semantics rather than a reduction semantics is stylistic and not relevant for our treatment. After this remark, the rules in Figure 2(c) essentially include the original rules of cc-pi plus the rules concerning the transactional mechanism. Roughly, the idea behind this operational semantics is to proceed as follows. First, rearranging processes into the normal form $(x_1)\ldots(x_n)\,(C\,|\,U)$ by applying the structural axioms. Next, applying the rules (TELL), (ASK), (RETRACT) for primitives on constraints and the rule (OUT), (INP), possibly (SUM) and (COMM) for synchronising processes. Finally, closing with respect to parallel composition and restriction ((PAR), (RES)). More in detail, rule (TELL) states that if $C\,|\,c$ is consistent then a process can place c in parallel with C, the process aborts otherwise. Rule (ASK) specifies that a process starting with an `ask` c prefix evolves to its continuation when c is entailed by C and it is stuck otherwise. By rule (RETRACT) a process can remove c if c is one of the syntactic constraints of C. In rules (COMM), we write $\widetilde{y} = \widetilde{w}$ to denote the parallel composition of fusions $y_1 = w_1\,|\,\ldots\,|\,y_n = w_n$. Intuitively, two processes $\overline{x}\langle\widetilde{y}\rangle.P$ and $z\langle\widetilde{w}\rangle.Q$ can synchronise when the equality of the names x and z is entailed by C and the parallel composition $C\,|\,\widetilde{y} = \widetilde{w}$ is consistent. Note that it is legal to treat name equalities as constraints c over $\mathcal{C}$, because named c-semirings contain fusions. Rule (PAR) allows for the closure with respect to unconstrained processes in parallel. This rule disallows computations that consider only partial stores of constraints and, consequently,

it makes necessary to add the parallel composition of constraints C in several operational rules, such as (TAU) and (SUM), even though this might seem superfluous. The remaining rules deal with transactions. Rules (TRANS), (TR-COMP), (COMMIT), and (TR-COMM) serve the same purpose as the homologous rules of the committed pi. Note that the well-formedness assumption ensures that C, C_1 and C_2 can only share channel names. Rules (ABT-TELL) and (ABT-COMM) force an abort in case a process tries to place a constraint that is not consistent with the parallel composition of constraints C. Rules (ABT-PAR) extends the effect of an abort to the sibling processes. Unlike rule (PAR), rule (TR-PAR) allows closure with respect to constraints running in parallel. Note that such composition is legal in virtue of the well-formedness assumption which ensures $\mathrm{fn}(C) \cap fn(P,Q) = \emptyset$.

4.3 Example: a Transactional SLA

We now model in committed cc-pi the scenario depicted in Example 1. The client and the server can be sketched as follows.

$$\begin{aligned}\mathsf{Client}_{\mathsf{rb},\mathsf{c}}(r,p) &\equiv (bw)(cost)[\texttt{tell}\ (bw \geq \mathsf{rb}).\overline{r}\langle bw\rangle.\texttt{tell}\ (cost \leq \mathsf{c}).p\langle cost\rangle.\mathbf{0} : Q].\\ &\qquad U(bw,cost)\\ \mathsf{Provider}_{\mathsf{ob},\mathsf{uc}}(r,p) &\equiv (bw')(cost')[\texttt{tell}\ (bw' \leq \mathsf{ob}).r\langle bw'\rangle.\\ &\qquad \texttt{tell}\ (bw' * \mathsf{uc} = cost').\overline{p}\langle cost'\rangle.\mathbf{0} : Q'].U'(bw',cost')\end{aligned}$$

The specification above is the same as the one given in Example 1 using cc-pi, apart from the fact that here the sequences of actions taken by each party are within a transactional scope and that they include compensating processes Q and Q'. Assume the following system composed of a client and two providers.

$$\mathsf{S} \equiv (r)(p)\mathsf{Client}_{\mathsf{4Gb},\$100} \mid \mathsf{Provider}_{\mathsf{6Gb},\$20} \mid \mathsf{Provider}_{\mathsf{3Gb},\$15}$$

By executing the `tell` prefixes in all transactions we obtain the following derivation (we abbreviate $U(bw,cost)$ and $U'(bw',cost')$ with U and U' respectively).

$$\begin{aligned}\mathsf{S} \xrightarrow{\tau}{}^{*}\ & (r)(p)(bw)(cost)[bw \geq \mathsf{4Gb} \mid \overline{r}\langle bw\rangle.\texttt{tell}\ (cost \leq \$100).p\langle cost\rangle.\mathbf{0} : Q].U\\ & \mid (bw')(cost')[bw' \leq \mathsf{6Gb} \mid r\langle bw'\rangle.\texttt{tell}\ (bw' * \$20 = cost').\overline{p}\langle cost'\rangle.\mathbf{0} : Q'].U'\\ & \mid (bw')(cost')[bw' \leq \mathsf{3Gb} \mid r\langle bw'\rangle.\texttt{tell}\ (bw' * \$15 = cost').\overline{p}\langle cost'\rangle.\mathbf{0} : Q'].U'\end{aligned}$$

As in the non-transactional case, the client can synchronise only with the first provider. Hence, the only possible reduction is

$$\begin{aligned}\xrightarrow{\tau}\ & (r)(p)(bw)(cost)(bw')(cost')\\ & [bw \geq \mathsf{4Gb} \mid bw = bw' \mid bw' \leq \mathsf{6Gb}\\ & \mid \texttt{tell}\ (cost \leq \$100).p\langle cost\rangle.\mathbf{0} \mid \texttt{tell}\ (bw' * \$20 = cost').\overline{p}\langle cost'\rangle.\mathbf{0} : Q|Q'].(U|U')\\ & \mid (bw')(cost')[bw' \leq \mathsf{3Gb} \mid r\langle bw'\rangle.\texttt{tell}\ (bw' * \$15 = cost').\overline{p}\langle cost'\rangle.\mathbf{0} : Q'].U'\end{aligned}$$

Next, the provider and the client fix their constraints on the cost of the service, the communication over p takes place, and the transaction can commit:

$$\begin{aligned}\xrightarrow{\tau}{}^{*}\ & (r)(p)(bw)(cost)(bw')(cost')\\ & bw \geq \mathsf{4Gb} \mid bw = bw' \mid bw' \leq \mathsf{6Gb} \mid bw' * \$20 = cost' \mid cost \leq \$100 \mid U \mid U'\\ & \mid (bw')(cost')[bw' \leq \mathsf{3Gb} \mid r\langle bw'\rangle.\texttt{tell}\ (bw' * \$15 = cost').\overline{p}\langle cost'\rangle.\mathbf{0} : Q'].U'\end{aligned}$$

Consider now the variant shown in Example 3 in which the client is $\mathsf{Client}_{\mathsf{4Gb,\$60}}$ instead of $\mathsf{Client}_{\mathsf{4Gb,\$100}}$. In this case, the system may evolve as before until the client and the provider fix the constraint about variables *cost* and *cost'*. Afterwards, when they synchronise on p, the transaction aborts since the constraints are now not consistent. In such case the compensations Q and Q' are activated. Note that the precise definition of the compensations may dictate the strategy followed by each participant during the negotiation. For instance, for the client the compensation could be $\mathsf{Client}_{\mathsf{rb,c+\$10}}$. That is it may offer to pay more for the requested bandwidth, or alternatively $\mathsf{Client}_{\mathsf{rb-1Gb,c}}$ to request less bandwidth by offering the same price. Similarly, the provider may fix its own negotiation strategy.

5 Big-Step Operational Semantics

In this section we introduce an alternative definition for the semantics of committed cc-pi, which allows us to reason about transactional computations at different levels of abstraction. In particular, the big-step semantics is intended to single out the computations of a system that are not transient, or in other words, the states containing no running transactions. Therefore, the big-step semantics provides a description of the possible evolution of a system through stable states. Processes associated to stable states of the system are said *stable processes*. Formally, a process P is *stable* if $P \not\equiv (\tilde{x})\,[P_1 : Q_1].U_1 \,|\, P_2$, i.e. P does not contain active transactions. We remark that our definition of stable process is intentionally not preserved by weak bisimulation. In case such property is required, an alternative characterization of stable process could be given by slightly adapting the original semantics in order to make the beginning of transaction executions observable.

A committed cc-pi process P is a *shallow process* if every subterm of the form $[P' : Q']$ occurs under a prefix τ. Moreover, we require U shallow for any definition $D(\tilde{x}) \stackrel{\text{def}}{=} U$. The main idea behind shallow processes is that of syntactically distinguish transactional terms that have not been activated yet (i.e., those occurring after τ prefixes) from those that are already active (i.e., non stable processes). For instance, the process $\tau.[U : U']$ denotes a transaction that has not been activated, while the term $[U : U']$ stands for a transaction that is in execution.

Hereafter we assume all processes to be shallow. Moreover, we let P_S and U_S range over *stable* processes and *stable unconstrained* processes, respectively. We remark that any process P can be straightforwardly rewritten as a shallow process by adding τ prefixes before any transactions, without changing the meaning of the program.

The big-step or high-level semantics of processes is the smallest relation $P \stackrel{\tau}{\Rightarrow} Q$ induced by the rules in Figure 3. Most rules are analogous to the small-step semantics. The only rules that have been redefined are (TRANS) and (TR-COMM). In particular, the new (TR-COMM') allows the merge of transactions only when the synchronising processes U_{S_1} and U_{S_2} are stable. Similarly, rule (TRANS') requires internal reductions to be high-level steps, i.e., reductions from stable processes to stable processes. Hence, the reduction $[P_S : Q_S].U_S \stackrel{\tau}{\rightarrowtail} [P'_S : Q_S].U_S$ is not a high-level step, since it does not relate stable processes. In addition, rule (SEQ) stands for the sequential composition of

(TAU)
$$C \,|\, \tau.U \overset{\tau}{\rightarrowtail} C \,|\, U$$

(OUT)
$$C \,|\, \overline{x}\langle\tilde{y}\rangle.U \overset{\overline{x}\langle\tilde{y}\rangle}{\rightarrowtail} C \,|\, U$$

(INP)
$$C \,|\, x\langle\tilde{y}\rangle.U \overset{x\langle\tilde{y}\rangle}{\rightarrowtail} C \,|\, U$$

(TELL)
$$C \,|\, \texttt{tell}\ c.U \overset{\tau}{\rightarrowtail} C \,|\, c \,|\, U \quad \text{if } C \,|\, c \text{ consistent}$$

(ABT-TELL)
$$C \,|\, \texttt{tell}\ c.U \overset{\texttt{abr}}{\rightarrowtail} \mathbf{0} \quad \text{if } C \,|\, c \text{ not consistent}$$

(ASK)
$$C \,|\, \texttt{ask}\ c.U \overset{\tau}{\rightarrowtail} C \,|\, U \quad \text{if } C \vdash c$$

(RETRACT)
$$C \,|\, \texttt{retract}\ c.U \overset{\tau}{\rightarrowtail} (C - c) \,|\, U$$

(COMM)
$$\frac{C \,|\, U \overset{x\langle\tilde{y}\rangle}{\rightarrowtail} C \,|\, U' \quad C \,|\, V \overset{\overline{z}\langle\tilde{w}\rangle}{\rightarrowtail} C \,|\, V'}{C \,|\, U \,|\, V \overset{\tau}{\rightarrowtail} C \,|\, \tilde{y} = \tilde{w} \,|\, U' \,|\, V'} \quad \text{if } |\tilde{y}| = |\tilde{w}| \text{ and } C \,|\, \tilde{y} = \tilde{w} \text{ consistent and } C \vdash x = z$$

(ABT-COMM)
$$\frac{C \,|\, U \overset{x\langle\tilde{y}\rangle}{\rightarrowtail} P \quad C \,|\, V \overset{\overline{z}\langle\tilde{w}\rangle}{\rightarrowtail} Q}{C \,|\, U \,|\, V \overset{\texttt{abr}}{\rightarrowtail} \mathbf{0}} \quad \text{if } |\tilde{y}| = |\tilde{w}| \text{ and } C \,|\, \tilde{y} = \tilde{w} \text{ not consistent and } C \vdash x = z$$

(PAR)
$$\frac{P \overset{\alpha}{\rightarrowtail} P' \quad \alpha \neq \texttt{abr}}{P \,|\, U \overset{\alpha}{\rightarrowtail} P' \,|\, U}$$

(ABT-PAR)
$$\frac{P \overset{\texttt{abr}}{\rightarrowtail} \mathbf{0}}{P \,|\, Q \overset{\texttt{abr}}{\rightarrowtail} \mathbf{0}}$$

(SUM)
$$\frac{C \,|\, \pi_i.U_i \overset{\alpha}{\rightarrowtail} U'}{C \,|\, \sum \pi_i.U_i \overset{\alpha}{\rightarrowtail} U'}$$

(RES)
$$\frac{P \overset{\tau}{\rightarrowtail} P'}{(x)\,P \overset{\tau}{\rightarrowtail} (x)\,P'}$$

(TRANS')
$$\frac{P_S \overset{\tau}{\Rightarrow} P'_S}{[P_S : Q_S].U_S \overset{\tau}{\rightarrowtail} [P'_S : Q_S].U_S}$$

(TR-COMP)
$$\frac{P \overset{\texttt{abr}}{\rightarrowtail} P'}{[P : Q].U \overset{\tau}{\rightarrowtail} Q}$$

(TR-COMMIT)
$$[C : Q].U \overset{\tau}{\rightarrowtail} C \,|\, U$$

(TR-PAR)
$$\frac{[P : Q].U \overset{\tau}{\rightarrowtail} P'}{C \,|\, [P : Q].U \overset{\tau}{\rightarrowtail} C \,|\, P'}$$

(TR-COMM')
$$\frac{C_1 \,|\, U_{S_1} \overset{\overline{x}\langle\tilde{y}\rangle}{\rightarrowtail} R_{S_1} \quad C_2 \,|\, U_{S_2} \overset{z\langle\tilde{w}\rangle}{\rightarrowtail} R_{S_2} \quad |\tilde{y}| = |\tilde{w}| \text{ and } C_1 \,|\, C_2 \,|\, \tilde{y} = \tilde{w} \text{ consistent and } C \,|\, C_1 \,|\, C_2 \vdash x = z}{C \,|\, [C_1 \,|\, U_{S_1} : Q_{S_1}].V_1 \,|\, [C_2 \,|\, U_{S_2} : Q_{S_2}].V_2 \overset{\tau}{\rightarrowtail} C \,|\, [R_{S_1} \,|\, R_{S_2} \,|\, \tilde{y} = \tilde{w} : Q_{S_1} \,|\, Q_{S_2}].(V_1 \,|\, V_2)}$$

(SEQ)
$$\frac{P \overset{\tau}{\rightarrowtail} P' \quad P' \overset{\tau}{\rightarrowtail} P''}{P \overset{\tau}{\rightarrowtail} P''}$$

(UP)
$$\frac{P_S \overset{\tau}{\rightarrowtail} P'_S}{P_S \overset{\tau}{\Rightarrow} P'_S}$$

Fig. 3. Big step semantics

low-level steps, and rule (UP) states that a low-level step is a high-level step when the involved processes are stable.

Example 4. We show the big-step reductions for the example given in Section 4.3. In particular, we consider the shallow version of the processes $\mathsf{Client}_{\mathsf{rb,c}}$ and $\mathsf{Provider}_{\mathsf{ob,uc}}$ (i.e., by adding the prefixes τ before transactional scopes). For instance, the following system

$$\mathsf{S}' \equiv (r)(p)\mathsf{Client}_{\mathsf{4Gb,\$100}} \mid \mathsf{Provider}_{\mathsf{6Gb,\$20}} \mid \mathsf{Provider}_{\mathsf{5Gb,\$30}}$$

has the following two big-step reductions

$$\begin{aligned}\mathsf{S}' \overset{\tau}{\Rightarrow} & (r)(p)(bw)(cost)(bw')(cost') \\ & bw \geq \mathsf{4Gb} \mid bw = bw' \mid bw' \leq \mathsf{6Gb} \mid bw' * \$20 = cost' \mid cost \leq \$100 \mid U \mid U' \\ & \mid \mathsf{Provider}_{\mathsf{5Gb,\$30}}\end{aligned}$$

and

$$\mathsf{S}' \overset{\tau}{\Rightarrow} (r)(p)(bw)(cost)(bw')(cost')\ Q \mid Q' \mid \mathsf{Provider}_{\mathsf{6Gb,\$20}}$$

The first one describes the successful negotiation between the provider $\mathsf{Provider}_{\mathsf{6Gb,\$20}}$ and the client, while the the second one describes the failed negotiation between the client and $\mathsf{Provider}_{\mathsf{5Gb,\$30}}$.

The remaining of this section is devoted to show that the small- and the big-step semantics coincide for shallow processes. Next propositions are auxiliary results that will be used for proving the main theorem.

Proposition 1. *If $P_S \xrightarrow{\alpha} P$ and $\alpha = x\langle\tilde{y}\rangle, \overline{x}\langle\tilde{y}\rangle$, then P is stable.*

Proof. By rule induction (using the fact that transactions occur only under τ prefixes in shallow processes).

The following result assures that a derivation from a stable process P_S that reduces to a non stable process, which is able to perform an input, an output, or an abort action, can be rewritten as a computation that executes the respective action first, and then starts all the transactions.

Proposition 2. *Let P_S be a stable process. If $P_S \xrightarrow{\tau}{}^{*} (\tilde{x})R_S|T$ and $R_S \xrightarrow{\alpha} R'$ for $\alpha = y\langle\tilde{z}\rangle, \overline{y}\langle\tilde{z}\rangle, \mathtt{abr}$, then, there exists a stable process T_S s.t. $P_S \xrightarrow{\tau}{}^{*} (\tilde{x})R_S|T_S \xrightarrow{\alpha} (\tilde{x})R'|T_S \xrightarrow{\tau}{}^{*} (\tilde{x})R'|T$.*

Proof (sketch). Proof follows by induction on the length of the derivation $P_S \xrightarrow{\tau}{}^{n} (\tilde{x})R_S|T$. The base case ($n = 0$) is immediate by considering $T_S \equiv T$ (note that $P_S \equiv (\tilde{x})R_S|T$ and, hence, T is stable). Inductive step follows by considering $P_S \xrightarrow{\tau} P$. There are two main possibilities: if P is stable, then the proof is immediate by inductive hypothesis. If P is not stable, the only possibility for $P_S \xrightarrow{\tau} P$ is $P_S \equiv (\tilde{z})\tau.[Q : Q'].U|O_S$ (by shallowness, transactions occur only under τ prefixes). Consequently, $P \equiv (\tilde{z})[Q : Q'].U|O_S$. There are three possibilities: (i) when $O_S \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} O'_S$, then the proof follows by using inductive hypothesis; (ii) when $[Q : Q'].U \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} O$, then α occurs after the commit of the

transaction and $U \xrightarrow{\tau}^{*} \xrightarrow{\alpha} O$, it follows by inductive hypothesis (since U is stable); (iii) $[Q : Q'].U|O_S \xrightarrow{\tau}^{*} \xrightarrow{\alpha} O$ by applying at least once rule (TR-COMM). Also in this case α may occur only after the commit of all joint transactions, which releases only stable processes. Hence, the proof follows by inductive hypothesis.

Next proposition assures that all the possible states reached by the execution of a pair of transactions that have no active subtransactions can be obtained by computations that never merge transactions containing subtransactions.

Proposition 3. *For any two shallow non nested transactions $[P_{S_1} : Q].U$ and $[P_{S_2} : Q'].U'$, the following holds*

$$C|[P_{S_1} : Q].U|[P_{S_2} : Q'].U' \xrightarrow{\tau}^{*} C|[P_1 : Q].U|[P_2 : Q'].U' \xrightarrow{\tau} C|[P : Q|Q'].(U|U')$$

implies

$$C|[P_{S_1} : Q].U|[P_{S_2} : Q'].U' \rightarrowtail^{\tau}{}^{*} C|[P'_{S_1} : Q].U|[P'_{S_2} : Q'].U' \rightarrowtail^{\tau} C|[P_S : Q|Q'].(U|U') \rightarrowtail^{\tau}{}^{*} C|[P : Q|Q'].(U|U')$$

Proof (sketch). The reduction step $C|[P_1 : Q].U|[P_2 : Q'].U' \xrightarrow{\tau} C|[P : Q|Q'].(U|U')$ implies that $\exists x, z$ s.t. $P_1 \xrightarrow{x\langle\tilde{y}\rangle} P'_1$, $P_2 \xrightarrow{\bar{z}\langle\tilde{w}\rangle} P'_2$, $C \vdash x = z$, and $P = P'_1|P'_2|\tilde{y} = \tilde{w}$. Note that $P_1 \xrightarrow{x\langle\tilde{y}\rangle} P'_1$ implies $P_1 \equiv (\tilde{v})R_{S_1}|T_1$ and $R_{S_1} \xrightarrow{x\langle\tilde{y}\rangle} R'_{S_1}$. By Proposition 2, there exists a stable process P'_{S_1} s.t. $P_{S_1} \xrightarrow{\tau}^{*} P'_{S_1} \xrightarrow{x\langle\tilde{y}\rangle} P''_1 \xrightarrow{\tau}^{*} P'_1$. By Proposition 1, P''_1 is stable. Similarly, for P_{S_2}. Hence, both transactions can be merged, obtaining $C|[(\tilde{v})P''_1|P''_2|\tilde{y} = \tilde{w} : Q|Q'].(U|U')$. Note $P''_1|P''_2$ is stable and therefore the proof follows by taking $P_S \equiv (\tilde{v})P''_1|P''_2|\tilde{y} = \tilde{w}$.

Lemma 1. *$P_S \xrightarrow{\tau}^{+} P'_S$ implies $P_S \rightarrowtail^{\tau} P'_S$.*

Proof (sketch). Follows by induction on the length of the derivation $P_S \xrightarrow{\tau}^{n} P'_S$. Base case ($n = 1$) is immediate by rule analysis. Inductive step ($n = k$) considers $P_S \xrightarrow{\tau} P \xrightarrow{\tau}^{k} P'_S$. If P is stable, the proof is completed by applying inductive hypothesis. Otherwise, the only possibility is $P \equiv (\tilde{x})[Q_S : Q'_S].U_S|P_{S_1}$ and $P_S \rightarrowtail^{\tau} P$ (proved by structural induction over P_S). Since $P \xrightarrow{\tau}^{k} P'_S$, there are three possibilities for completing the computation:

1. The transaction commits by itself, then $Q_S \xrightarrow{\tau}^{*} (\tilde{y})C$. By inductive hypothesis $Q_S \rightarrowtail^{\tau} (\tilde{y})C$. By rule (UP) $Q_S \overset{\tau}{\Rightarrow} (\tilde{y})C$, by (TRANS') $[Q_S : Q'_S].U_S \rightarrowtail^{\tau} [(\tilde{y})C : Q'_S].U_S$, by (COMMIT) $[(\tilde{y})C : Q'_S].U_S \rightarrowtail^{\tau} (\tilde{y})C|U_S$, by (TR-PAR) and (PAR) $[Q_S : Q'_S].U_S|P_{S_1} \rightarrowtail^{\tau} (\tilde{y})C|U_S|P_{S_1}$, and finally by (RES) $(\tilde{x})[Q_S : Q'_S].U_S|P_{S_1} \rightarrowtail^{\tau} (\tilde{x})(\tilde{y})C|U_S|P_{S_1}$. The proof is completed by inductive hypothesis on $(\tilde{x})(\tilde{y})C|U_S|P_{S_1} \xrightarrow{\tau} P'_S$ and rule (STEP).
2. The transaction aborts by itself, then $Q_S \xrightarrow{\tau}^{*} Q \xrightarrow{\mathtt{abr}} Q'$. Then, by Proposition 2 there exists Q'' and Q''' stable s.t. $Q_S \xrightarrow{\tau}^{*} Q'' \xrightarrow{\mathtt{abr}} Q'''$. By, inductive hypothesis $Q_S \rightarrowtail^{\tau}{}^{*} Q''$. By (TRANS') $[Q_S : Q'_S].U_S \rightarrowtail^{\tau} [Q'' : Q'_S].U_S$. By structural induction we can prove that $Q'' \xrightarrow{\mathtt{abr}} Q'''$ implies $Q'' \rightarrowtail^{\mathtt{abr}} Q'''$, and then by (TR-COMP) $[Q'' : Q'_S].U_S \rightarrowtail^{\tau} Q'_S$. The proof is completed as in the previous case.

3. The transaction merges with some transaction activated by P_{S_1}. The proof follows by using repeatedly Proposition 3 for proving that merge of transactions can be done with non nested transactions, and inductive hypothesis for proving that reductions inside transactions from stable to stable processes correspond to $\rightarrowtail$ reductions.

Theorem 1. *$P_S \xrightarrow{\tau}^{+} P'_S$ implies $P_S \overset{\tau}{\Rightarrow} P'_S$.*

Lemma 2. *$P \overset{\tau}{\rightarrowtail} P'$ implies $P \xrightarrow{\tau}^{+} P'$.*

Proof (sketch). Proof follows by rule induction. Rules (TAU), (TELL), (ASK), (RETRACT), are immediate. Cases (PAR), (SUM), (RES), (TR-PAR) follows by inductive hypothesis. Cases (COMM) and (TR-COMM) follow by proving using rule induction that $P \overset{\alpha}{\rightarrowtail} P'$ for $\alpha = x\langle\tilde{y}\rangle, \bar{x}\langle\tilde{y}\rangle$ implies $P \xrightarrow{\alpha} P'$. If the last applied rule is (TRANS), then $P \equiv [P_S : Q_S].U_S$. Consequently, the proof has the following shape:

$$\dfrac{\dfrac{P_S \overset{\tau}{\rightarrowtail} P'_S}{P_S \overset{\tau}{\Rightarrow} P'_S}\ (\text{UP})}{[P_S : Q_S].U_S \overset{\tau}{\rightarrowtail} [P_S : Q_S].U_S}\ (\text{TRANS})$$

By inductive hypothesis on $P_S \overset{\tau}{\rightarrowtail} P'_S$ we have that $P_S \xrightarrow{\tau}^{+} P'_S$. Then, it can be proved by induction on the length of the derivation that $P_S \xrightarrow{\tau}^{+} P'_S$ implies $[P_S : Q_S].U_S \xrightarrow{\tau}^{+} [P'_S : Q_S].U_S$

Theorem 2. *$P \overset{\tau}{\Rightarrow} P'$ implies $P \xrightarrow{\tau}^{+} P'$.*

Theorem 3. *$P \overset{\tau}{\Rightarrow} P'$ iff $P \xrightarrow{\tau}^{+} P'$.*

Proof. Immediate by Theorems 1 and 2.

6 Concluding Remarks

We have presented a constraint-based model of transactional SLAs. In our language, the mutual responsibilities of service providers and clients are expressed in terms of constraints, which are placed by each party during the negotiation. If the combination of such constraints is consistent, then they form the SLA contract. On the contrary, if the negotiation fails, each party can activate a programmable compensation aimed e.g. at relaxing client requirements or increasing service guarantees.

The proposed approach seems promising for studying more complex negotiation scenarios that, for instance, include third parties applications or feature arbitrarily nested transactions. We also plan to investigate different compensation mechanisms in which, e.g., the constraints placed until the failure are not discarded when the transaction aborts and allowing the compensating process to take advantage of them. Similarly, it would be interesting to consider an optimistic approach to transactions along the lines of [1]. This could be achieved by relaxing the well-formedness assumption and by allowing global constraints to be copied inside transactional scopes upon transaction activation.

Acknowledgments. We thank Roberto Bruni and Ugo Montanari for fruitful discussions.

References

1. L. Acciai, M. Boreale, and S. dal Zilio. A concurrent calculus with atomic transactions. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lect. Notes in Comput. Sci.* Springer Verlag, 2007.
2. D. Bacciu, A. Botta, and H. Melgratti. A fuzzy approach for negotiating quality of services. In *Proc. of 2nd Symposium on Trustworthy Global Computing (TGC 2006)*, To appear as *Lect. Notes in Comput. Sci.* Springer Verlag, 2007.
3. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.
4. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *Proc. of 6th IFIP International Conference on Formal Methods for Open-Object Based Distributed Systems (FMOODS'03)*, volume 2884 of *Lect. Notes in Comput. Sci.*, pages 124–138. Springer Verlag, 2003.
5. R. Bruni, H. Melgratti, and U. Montanari. Nested commits for mobile calculi: extending Join. In *Proc. of the 3rd IFIP-TCS 2004, 3rd IFIP Intl. Conference on Theoretical Computer Science*, pages 569–582. Kluwer Academic Publishers, 2004.
6. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220. ACM Press, 2005.
7. M. G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In *Proc. of the 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lect. Notes in Comput. Sci.* Springer Verlag, 2007.
8. M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In A. Abdallah and J. Sanders, editors, *Proceedings of 25 Years of CSP*, 2004.
9. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the Join calculus. In *Proc. of 23rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385. ACM Press, 1996.
10. A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Jour. Net. and Sys. Manag.*, 11(1):57–81, 2003.
11. C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS'05)*, volume 3441 of *Lect. Notes in Comput. Sci.*, pages 282–298. Springer Verlag, 2005.
12. R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77, 1992.
13. V. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the 17th Symposium on Principles of programming languages (POPL'90)*. ACM Press, 1990.
14. J. Skene, D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proc. of the 26th International Conference on Software Engineering (ICSE'04)*, 2004.
15. L. Wischik and P. Gardner. Explicit fusions. *Theoret. Comput. Sci.*, 340(3):606–630, 2005.