CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements*

Maria Grazia Buscemi[1] and Ugo Montanari[2]

[1] IMT Lucca Institute for Advanced Studies, Italy
marzia.buscemi@imtlucca.it
[2] Dipartimento di Informatica, University of Pisa, Italy
ugo@di.unipi.it

Abstract. Service Level Agreements are a key issue in Service Oriented Computing. SLA contracts specify client requirements and service guarantees, with emphasis on Quality of Service (cost, performance, availability, etc.). In this work we propose a simple model of contracts for QoS and SLAs that also allows to study mechanisms for resource allocation and for joining different SLA requirements. Our language combines two basic programming paradigms: name-passing calculi and concurrent constraint programming (cc programming). Specifically, we extend cc programming by adding synchronous communication and by providing a treatment of names in terms of restriction and structural axioms closer to nominal calculi than to variables with existential quantification. In the resulting framework, SLA requirements are constraints that can be generated either by a single party or by the synchronisation of two agents. Moreover, restricting the scope of names allows for local stores of constraints, which may become global as a consequence of synchronisations. Our approach relies on a system of *named* constraints that equip classical constraints with a suitable algebraic structure providing a richer mechanism of constraint combination. We give reduction-preserving translations of both cc programming and the calculus of explicit fusions.

1 Introduction

An important aspect of web services concerns client requirements and service guarantees with emphasis on Quality of Service, such as cost, performance, availability. These are commonly referred to as Service Level Agreements. SLAs between organisations are used in several areas of IT services, like hosting and communication services. The terms and conditions appearing in a SLA contract can be negotiated among the contracting parties prior to service execution.

In this paper we present a simple calculus, called *cc-pi calculus*, for modeling processes able to specify QoS requirements and to conclude SLA contracts. The proposed language is also equipped with mechanisms for resource allocation and for joining different SLA requirements. Our approach combines basic features of name-passing calculi and of concurrent constraint (cc) programming.

* Research supported by the EU IST-FP6 16004 Integrated Project SENSORIA

Name-passing calculi, such as the pi-calculus [7], are a key paradigm of computation whose interaction mechanism may dynamically change the communication topology. Since the introduction of name-passing calculi, the notion of *names* has been recognised as crucial in theories for concurrency and mobility.

The name-passing calculus we start with is the pi-F calculus [19]. The pi-F calculus is a variant of the pi-calculus [7], whose synchronisation mechanism is global and, instead of binding formal names to actual names, it yields *explicit fusions*, i.e. simple constraints expressing name equalities. For example, consider two processes $\overline{u}\langle v\rangle.P$ and $u\langle x\rangle.Q$, that are ready to make an output and an input on u, respectively. The interaction between these processes results in the explicit fusion of v and x. This fusion will also affect any further process R running in parallel: $R\,|\,\overline{u}\langle v\rangle.P\,|\,u\langle x\rangle.Q \rightarrow R\,|\,P\,|\,Q\,|\,x = v$. The restriction operator (x) can be used to limit the scope of a fusion, e.g.: $R\,|\,(x)(\overline{u}\langle v\rangle.P\,|\,u\langle x\rangle.Q) \rightarrow R\,|\,(x)(P\,|\,Q\,|\,x = v)$.

The cc-pi calculus extends the pi-F calculus by generalising explicit fusions like $x = v$ to *named constraints* and by adding primitives for handling such constraints. While the informal concept of constraint is widely used in a variety of different fields, a very general, formal notion of constraint system has been introduced in the `cc` programming paradigm [15]. Actually, `cc` programming is a simple and powerful computing model based on a shared store of constraints that provides partial information about possible values that variables can take. Concurrent agents can act on this store by performing either a `tell` action (for adding a constraint, if the resulting store is *consistent*) or an `ask` action (for checking if a constraint is *entailed* by the store). As computation proceeds, more and more information are accumulated, thus the store is *monotonically refined*.

Of the classical `cc` programming paradigm we keep the `ask` and `tell` constructs, but we extend/modify several other aspects. Maybe the most radical change is to give up the monotonicity requirement. While non-monotonicity was already present in the so-called *linear* `cc` programming [14], the introduction in our calculus of a `retract` construct, whose effect is to erase a previously told constraint, is strongly suggested by the need of allocating a resource and of deallocating *the same* resource. Of course monotonicity is the basis of several properties of `cc` programming, which thus do not hold in our framework. However whenever retracts are forbidden, or their usage is limited, some of the useful properties could be reinstated. We also introduce a `check` operation for verifying if a constraint is consistent with the store of constraints.

Another important difference with respect to [15] is that we adopt a different concept of general, abstract constraint system. While the classical notion is equipped with an operation of entailment and a predicate of consistency, being based on Dana Scott's information systems, we employ constraints forming c-semirings [2]. Roughly, a c-semiring consists of a set equipped with two binary operations, the sum $+$ and the product $\times$, such that $+$ is associative, commutative and idempotent, $\times$ is associative and commutative and $\times$ distributes over $+$. A c-semiring is automatically equipped with a partial ordering $a \leq b$, which means that a is more constrained than b, or, more interestingly, that a entails b, $a \vdash b$. The sum $a+b$ chooses the worst constraint better than a and b, while the product $a \times b$ combines two constraints. The simplest c-semiring consists of the booleans with $\vee$ as $+$ and $\wedge$ as $\times$.

Our c-semirings enjoy two kinds of nice properties. On the one hand they are very stable, since cartesian products, functional spaces and powerdomains of c-semirings are c-semirings. On the other hand c-semirings are quite adequate for modeling the so-called *soft* constraints, i.e. constraints which do not return only true or false, but more informative values instead. In fact it is easy to define c-semirings expressing fuzzy, hierarchical, or probabilistic values. Also, optimization algorithms work on the c-semiring consisting of the reals plus infinity with the operations of sum as $\times$ and min as $+$. Several efficient algorithms defined for ordinary, crisp constraints, like local propagation or dynamic programming, can be generalized to c-semirings.

The former kind of properties is used in the paper to model networks of constraints for defining constraint satisfaction problems (CSPs) [8]. In fact, a single constraint, or even a network of constraints, is a function which, given an assignment of the variables to some domain D, returns a boolean, or rather a value in a generic c-semiring in the soft case. CSPs are a well-established formalism, especially studied in the artificial intelligence area, adequate to specify many kinds of real-life problems. In this paper we do not fully explore the latter aspect of c-semirings. However we consider it as extremely valuable and we plan to further exploit it in the future. In fact, we believe that a lot of non-functional requirements of QoS can be adequately modeled using c-semirings.

The last, important difference with respect to [15] is that we handle variables, or rather names, in a very different way. In ordinary `cc` programming, constraints involving variables are seen as relations, in the style of Tarski's cylindrical algebras. This interpretation is particularly visible in the axioms for hiding (written as $\exists$) and variable equality. Instead, in our *named* constraints we regard variables as ordinary names in the pi-calculus style. More precisely, names are introduced, as for pi-calculus agents [9], by means of permutation algebras. Operations of permutation algebras are permutations of names. A key concept of permutation algebras is the *support* of a value, that specifies the set of names such that the permutations which do not affect them do not modify the value. Thus, equipping a c-semiring with a permutation algebra structure allows to characterise the set of relevant, i.e. *free*, names of a constraint c as the support of c. Since the treatment of names is the same, we can handle constraints as processes, making both syntax and semantics of our calculus simpler and more natural.

Besides `ask`, `tell`, `retract` and `check` there is another way in which agents can interact with the constraints existing in the system. In fact, synchronization of processes works like a global `ask` and `tell` construct. Two agents trying to perform an output $\overline{x}\langle y\rangle$ and an input $x'\langle y'\rangle$ action can synchronize only if the constraint $x = x'$ is entailed by the store. The result of the synchronization is a new constraint $y = y'$ which is told to the store. Fusion $y = y'$ can modify deeply the store, depending on the actual constraint system. For instance it can allow two local constraints to interact, establishing a SLA between the two partners. However, if the resulting constraint is inconsistent, the synchronization is forbidden. It can become possible at some later time if some other agent performs a `retract` action which makes the store less constrained.

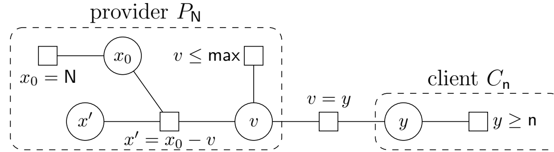
The special role of fusions in the control mechanisms of our calculus requires their presence in all constraint systems. Thus, we propose named c-semirings with name fusions, or equalities, as the underlying data model of cc-pi calculus.

In the paper, we show the generality of our approach by proposing three examples of named c-semirings, i.e. name equalities, Herbrand constraint systems, and soft CSPs, and we prove that they are effectively named c-semirings. We also show how our model can be applied in specifying and monitoring SLAs. Finally, we explore the expressiveness of our calculus by giving reduction-preserving translations of Pi-F and cc programming into cc-pi.

A motivating example Consider a service offering computing resources (e.g. units of CPUs of a given power) and suppose the service provider and a client want to reach a SLA. The provider P_{N}, with N available resources and the client C_{n} requiring at least n resources can be specified in our framework as follows, being max the maximum number of resources that can be allocated to each client:

$$\begin{aligned} P_{\mathsf{N}} &= (x_0)\,(\texttt{tell}\ (x_0 = \mathsf{N}).Q(x_0)) \\ Q(x) &= (v)\,(x')\,(\texttt{tell}\ (x' = x - v).\texttt{tell}\ (v \leq \mathsf{max}).c\langle v\rangle.Q(x')). \\ C_{\mathsf{n}} &= (y)\,(\texttt{tell}\ (y \geq n).\overline{c}\langle y\rangle.\tau.\texttt{retract}\ (y \geq n).\texttt{tell}\ (y = 0)\,). \end{aligned}$$

In words, P_{N} first sets the initial number of resources to N and evolves to Q. Process Q creates a name v representing the resources available to a client and a non-negative name x' counting the resources left after concluding a contract with the client; Q then adds the constraints $x' = x - v$ for setting the value of x' and $v \leq \mathsf{max}$ for imposing the bound max on v. Finally, Q signs the contract, i.e. it synchronises on a channel c with a client and, if the synchronisation succeeds, Q becomes ready to accept a new request. On the other side, C_{n} initially creates a local name y and places the constraint $y \geq n$. Next, C_{n} tries to synchronise on a public port c with a server. In case of success, C_{n} makes some calculation involving the obtained resources, which is modelled as a silent action τ. Then, C_{n} releases the allocated resources by removing the above constraint on y ($\texttt{retract}\ (y \geq n)$). Hence, a negotiation between P_{N} and C_{n} begins with the two parties placing their constraints. P_{N} and C_{n} can then synchronise (thus yielding the fusion of names v and y), if the resulting constraint system is consistent, i.e. if $\mathsf{n} \leq min(N, \mathsf{max})$, as shown by the graph representation below.



Related work Bacciu *et al.* [1] also propose a framework for specifying client requirements and provider guarantees on the offered services, along with negotiation mechanisms. Unlike our model, their approach relies on fuzzy sets rather than on c-semirings. The process calculus introduced in [4] focuses on controlling and coordinating distributed process interactions respecting QoS parameters expressed as c-semiring values, but the model does not cover negotiations. The ρ-calculus [11] is a concurrent calculus with first-order constraints and high-order procedural abstraction. Akin to our approach, the ρ-calculus is parametric to a certain constraint system. In [18] the ρ-calculus has been encoded into the Fusion Calculus [13]. Thus, we expect that ρ can

also be encoded into cc-pi. The pi$^+$-calculus [5] is an extension of the pi-calculus with constraint agents that can perform `tell` and `ask` actions. In contrast to our model, the constraint systems are first-order theories rather than algebraic structures and they do not support local stores. However, to our knowledge, none of the above languages has been applied for specifying SLA contracts. SLAng [17] and WSLA [6] are XML-based languages for defining SLAs at a lower level of abstractions. The elements of SLAng are also constraints on the behaviour of associated services and service clients, but their are specified in OCL. WSLA provides the ability to create new SLAs as functions over existing metrics. This is useful to formalise requirements that are expressed in terms of multiple QoS parameters. The semantics for expressions over metrics is not formally defined, though.

2 Background

2.1 C-semirings

We give here the basic definitions and properties concerning c-semirings. We refer to [2] for a more detailed treatment.

Definition 1 (c-semiring). *A* constraint semiring (c-semiring) *is a tuple* $\langle A,+,\times,0,1\rangle$ *such that: (i) A is a set and* $0,1 \in A$*; (ii)* $+$ *is commutative, associative, idempotent,* 0 *is its unit element and* 1 *is its absorbing element; (iii)* $\times$ *is associative, commutative and distributes over* $+$.

Let us consider the relation $\leq$ over A such that $a \leq b$ iff $a+b=b$. Then, it is possible to prove that (see [2]): (i) $\leq$ is a partial order; (ii) $+$ and $\times$ are monotone on $\leq$; (iii) $\times$ is intensive on $\leq$: $a \times b \leq a,b$; (iv) 0 is its minimum and 1 its maximum; (v) $\langle A,\leq\rangle$ is a complete lattice and, for all $a,b \in A$, $+$ is the least upper bound operator, that is, $a+b = lub(a,b)$. Moreover, if $\times$ is idempotent, then: $+$ distributes over $\times$; $\langle A,\leq\rangle$ is a distributive lattice and $\times$ is its greatest lower bound. Informally, the relation $\leq$ gives us a way to compare semiring values and constraints.

Typical examples are the c-semiring for classical CSPs $\langle\{\mathsf{False},\mathsf{True}\},\vee,\wedge,\mathsf{False},\mathsf{True}\rangle$, the c-semiring for fuzzy CSPs $\langle[0,1],max,min,0,1\rangle$, and the c-semiring for probabilistic CSPs $\langle[0,1],max,\cdot,0,1\rangle$. Since the Cartesian product of two c-semirings is still a c-semiring, it is also possible to model multicriteria optimization in this framework.

2.2 Permutation Algebras

We denote by $\mathcal{N}$ the infinite, countable, totally ordered set of *names* and we use $x,y,z\ldots$ to denote names. We write $\widetilde{x}$ for the tuple of names $\langle x_1,\ldots,x_n\rangle$. A *substitution* is a function $\sigma : \mathcal{N} \to \mathcal{N}$. We denote by $[y_1/x_1,\cdots,y_n/x_n]$ the substitution that maps x_i into y_i for $i=1,\ldots,n$ and which is the identity on the other names. The *identity substitution* is denoted by id. A *permutation* is a bijective name substitution. We let ρ range over permutations. The *kernel*, $K(\rho)$ of a permutation ρ is the set of the names that are changed by the permutation. A *permutation algebra* is defined by a carrier set and by a function defining how states are transformed by the finite-kernel permutations.

An interesting example is given by the permutation algebra for the pi-calculus [9]. In that case, the carrier contains all the processes, up to structural congruence, and the interpretation of a permutation is the associated name substitution.

The carrier of a permutation algebra can be partitioned into *orbits*, where two elements are in the same orbit if one can be obtained from the other by applying some permutation. To every element a a *symmetry* $\mathrm{sym}(a)$ can be associated, i.e. the group of all permutations ρ such that $a = \rho(a)$. The *support* $\mathrm{supp}(a)$ of an element a is the smallest set of names such that all the permutations that do not modify them are in $\mathrm{sym}(a)$. Intuitively, the names in $\mathrm{supp}(a)$ are the free names of a, the permutations which do not modify them are obviously not influent on a. Indeed, the permutations exchanging names in $\mathrm{supp}(a)$ with names not in $\mathrm{supp}(a)$ are renamings of the free names and do not belong to $\mathrm{sym}(a)$, while the permutations in $\mathrm{sym}(a)$ which modify only names in the support are genuine self-transformations of the element. A permutation algebra is *finite-support* if each element of its carrier has finite support.

3 Named Constraints

In this section, we propose a definition of *named constraints* that relies on the notion of *named c-semirings*. Essentially, a named c-semiring is a c-semiring enriched with a notion of name fusions, a permutation algebra A and a hiding operator $(\nu x.)$. In particular, A allows to characterise the finite set of *relevant names* of each element of the c-semiring as the support $\mathrm{supp}(c)$ of c in A, and $\nu x. c$ makes a name x local in c, in the style of process calculi. A named constraint is an element of a named c-semiring with an associated support.

Definition 2. *We define* (name) fusions *as total equivalence relations on $\mathcal{N}$ with only finitely many non-singular equivalence classes. By $x{=}y$ we denote the fusion with a unique non-singular equivalence class containing x and y.*

Definition 3. *A* named c-semiring $\mathcal{C} = \langle C, +, \times, \nu x., \rho, 0, 1\rangle$ *is a tuple where: (i) $x{=}y \in C$ for all x and y in $\mathcal{N}$; (ii) $\langle C, +, \times, 0, 1\rangle$ is a c-semiring; (iii) $\langle C, \rho\rangle$ is a finite-support permutation algebra; (iv) $\nu x. : C \to C$, for each name x, is a unary operation; (v) for all $c, d \in C$ and for all ρ the following axioms hold.*

$$
\begin{array}{ll}
(\text{FUSE}) & x{=}y \times c \;=\; x{=}y \times [y/x]\, c \\
(\text{HIDE}) & \nu x.\, 1 = 1 \quad \nu x. \nu y. c = \nu y. \nu x. c \quad \nu x.(c \times d) = c \times \nu x. d \text{ if } x \notin \mathrm{supp}(c) \\
 & \nu x.(c + d) = c + \nu x. d \;\text{ if } x \notin \mathrm{supp}(c) \quad \nu x.\, c = \nu y. [y/x]\, c \text{ if } y \notin \mathrm{supp}(c) \\
(\text{PERM}) & \rho 0 = 0 \quad \rho 1 = 1 \quad \rho(c \times d) = \rho c \times \rho d \quad \rho(c+d) = \rho c + \rho d \\
 & \rho(\nu x. c) = \nu x. (\rho c) \text{ if } x \notin K(\rho)
\end{array}
$$

The (FUS) axiom accounts for combining fusions and generic elements of c-semirings: $x{=}y \times c$ is equivalent to the product $x{=}y \times [y/x]\, c$ where y is replaced by x in c. The (HIDE) and (PERM) axioms rule how the ν and ρ operations, respectively, interact with the operations of the c-semiring. The axioms (HIDE) are inspired by the analogous structural congruence axioms for restriction in process calculi. Roughly, the c-semiring product $\times$ corresponds to the parallel composition of processes and constraint hiding is the

counterpart of restriction on processes. The notion of support $\text{supp}(c)$ associated with permutation algebras recalls the concept of free names in process calculi. According to the (PERM) axioms, ρ distributes with respect to $\times$ and $+$, and ρ is inactive on 0 and 1. Finally, the order of ρ and ν can be changed if x is not affected by ρ.

We propose below three examples of named c-semirings aimed at showing the generality of our approach. Specifically, we consider c-semirings for name equalities, for Herbrand constraint systems and for soft CSPs. Note that these named c-semirings can be suitably composed to model more complex constraint systems.

Example 1 (Name Equalitites). Let $\mathcal{R}$ be the set of all equivalence relation on $\mathcal{N}$. We define $\mathcal{C}_E$ as the tuple $\mathcal{C}_E = \langle C, +, \times, \nu x., \rho, 0, 1\rangle$ such that: (i) $C = \mathcal{R}$; (ii) $R_1 + R_2 = R_1 \cap R_2$; (iii) $R_1 \times R_2 = (R_1 \cup R_2)^\star$, i.e. $R_1 \times R_2$ is the reflexive, transitive, and symmetric closure of $R_1 \cup R_2$; (iv) $\nu x.R = R + \{(y,z) \,|\, y,z \neq x \text{ or } y = z = x\}$, i.e. $\nu x.R$ is obtained from R by replacing the equivalence class of x with the singleton $\{x\}$; (v) $\rho R = \{(\rho(x), \rho(y)) \,|\, (x,y) \in R\}$; (vi) $0 = (\mathcal{N} \times \mathcal{N})$ and $1 = \{(x,x) \,|\, x \in \mathcal{N}\}$.

Proposition 1. *$\mathcal{C}_E$ is a named c-semiring with idempotent product $\times$.*

Example 2 (Herbrand Constraint System). Given a signature Σ, let $=_E$ be an equational theory on $T_\Sigma(\mathcal{N})$, plus the additional axioms:

$$\frac{f(t_1,\ldots,t_n) =_E f(t'_1,\ldots,t'_n)}{t_i =_E t'_i} \quad i = 1,\ldots,n \qquad\qquad \frac{x =_E t \quad t_1 =_E t_2}{[t/x]t_1 =_E [t/x]t_2}$$

and with the restrictions: $x \neq_E t(x)$ and $f(t_1,\ldots,t_n) \neq_E g(t_1,\ldots,t_m)$, where $t(x)$ is any term different than x which contains x and $f \neq g$. We define $\mathcal{C}_H$ as the tuple $\mathcal{C}_H = \langle C, +, \times, \nu x., \rho, 0, 1\rangle$ where: (i) C is the set of the above-defined equational theories plus a bottom element $\bot$; (ii) $E_1 + E_2 = E_1 \cap E_2$; (iii) $E_1 \times E_2$ is the unification of E_1 and E_2, i.e. it is the smallest equational theory largest than or equal to $E_1 \cup E_2$, if it exists, otherwise $\bot$; (iv) $\nu x.\, E = E \cap \bar{E}$, where $t_1 =_{\bar{E}} t_2$ iff $t_1 = t_2$ or x does not occur in t_1, t_2; (v) $\rho t_1 =_{\rho E} \rho t_2$ iff $t_1 =_E t_2$; (vi) $0 = \bot$ and $1 = \{(t,t) \,|\, t \in T_\Sigma(\mathcal{N})\}$.

Proposition 2. *$\mathcal{C}_H$ is a named c-semiring with idempotent product $\times$.*

Example 3 (soft CSPs). Given a domain D of interpretation for $\mathcal{N}$, and a c-semiring $S = \langle A, +, \times, 0, 1\rangle$, a *soft* constraint c can be represented as a function $c = (\mathcal{N} \to D) \to A$ associating to each variable assignment $\eta = \mathcal{N} \to D$ a value of A. We define $\mathcal{C}_{\text{soft}}$ as the tuple $\mathcal{C}_{\text{soft}} = \langle C, +', \times', \nu x., \rho, 0', 1'\rangle$ such that: (i) C is the set of all soft constraints over $\mathcal{N}$, D and S; (ii) fusions $x{=}y$ are defined as $(x = y)\eta = 1$ if $\eta(x) = \eta(y)$, $(x = y)\eta = 0$ otherwise; (iii) $(c_1 +' c_2)\eta = c_1\eta + c_2\eta$; (iv) $(c_1 \times' c_2)\eta = c_1\eta \times c_2\eta$; (v) $(\nu x.c)\eta = \sum_{d \in D}(c\eta[^d/_x])$, where the assignment $\eta[^d/_x]$ is defined, as usual, as $\eta[^d/_x](y) = d$ if $x = y$, $\eta(y)$ otherwise; (vi) $(\rho c)\eta = c\overline{\eta}$ with $\overline{\eta}(x) = \eta(\rho(x))$; (vii) $0'\eta = 0$ and $1'\eta = 1$ for all η.

Proposition 3. *$\mathcal{C}_{\text{soft}}$ is a named c-semiring with idempotent product $\times$.*

Note that the support $\text{supp}(c)$ of an element of $\mathcal{C}_{\text{soft}}$ coincides with the support of a functional constraint c as defined in [3].

Definition 4 ((named) constraint). *Given a named c-semiring with equalities* $\langle A,+,\times,\rho,\nu x.,0,1\rangle$*, a* (named) constraint c *is an element of* A*.*

We define here the notions of *consistency* and *entailment* of constraints. They are analogous to the corresponding definitions given by Saraswat and Rinard [15]. Below we abbreviate by $(\times C)$ the product $c_1 \times \ldots \times c_n$ with $C = \{c_1,\ldots,c_n\}$.

Definition 5. *Let* $\langle A,+,\times,\rho,\nu x.,0,1\rangle$ *be a named c-semiring and* $C \subseteq A$ *be a set of constraints.* C *is* consistent *if* $(\times C) \neq 0$*. Moreover, given a constraint* $c \in A$*, we say that* C *entails* c*, written* $C \vdash c$*, if* $(\times C) \leq c$*.*

4 The cc-pi Calculus

4.1 Syntax

We assume the countable set of names $\mathcal{N}$ and a set of process identifiers, ranged over by D. We let c range over constraints of an arbitrary named c-semiring $\mathcal{C}$.

Definition 6. *The sets of* prefixes *and* cc-pi processes *are defined as follows:*

PREFIXES	$\pi ::= \tau \mid \overline{x}\langle\tilde{y}\rangle \mid x\langle\tilde{y}\rangle \mid \texttt{tell}\ c \mid \texttt{ask}\ c \mid \texttt{retract}\ c \mid \texttt{check}\ c$
UNCONSTRAINED PROCESSES	$U ::= \mathbf{0} \mid U\vert U \mid \sum_i \pi_i.U_i \mid (x)U \mid D(\tilde{y})$
CONSTRAINED PROCESSES	$P ::= U \mid c \mid P\vert P \mid (x)P$

The τ prefix stands for a silent action, the output prefix $\overline{x}\langle\tilde{y}\rangle$ for emitting over the port x the message $\tilde{y}$ and the input prefix $x\langle\tilde{y}\rangle$ for receiving over x a message and binding it to $\tilde{y}$. 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 is not enabled 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. Prefix `check` c is enabled if c is consistent with the set of constraints in parallel. *Unconstrained processes* U are essentially processes that can only contain constraints c in prefixes `tell` c, `ask` c, `retract` c, and `check` c. As usual, 0 stands for the inert process and $U \mid U$ for the parallel composition. $\sum_i \pi_i.U_i$ denotes an external 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 identifier D is of the form $D(\tilde{x}) \stackrel{\text{def}}{=} U$ with $|\tilde{x}| = |\tilde{y}|$. *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.

We extend the usual notion of *free names* of a process by stating that the set of free names of a constraint c is the support $\mathrm{supp}(c)$ defined in the previous section. Formally,

the set $\mathrm{fn}(P)$ is inductively defined as follows:

$$\mathrm{fn}(\mathbf{0}) = \emptyset \quad \mathrm{fn}(\tau.U) = \mathrm{fn}(U) \quad \mathrm{fn}(\overline{x}\langle\tilde{y}\rangle.U) = \{x,y\} \cup \mathrm{fn}(U) \quad \mathrm{fn}(x\langle\tilde{y}\rangle.U) = \{x,y\} \cup \mathrm{fn}(U)$$
$$\mathrm{fn}(\pi.U) = \mathrm{supp}(c) \cup \mathrm{fn}(U) \quad \text{if } \pi = \texttt{tell}\ c, \texttt{ask}\ c, \texttt{retract}\ c, \texttt{check}\ c$$
$$\mathrm{fn}(\textstyle\sum_i \pi_i.U_i) = \cup_i \mathrm{fn}(\pi_i.U_i) \quad \mathrm{fn}(D(\tilde{x})) = \mathrm{fn}(U) \quad \text{if } D(\tilde{x}) \stackrel{\text{def}}{=} U$$
$$\mathrm{fn}(c) = \mathrm{supp}(c) \quad \mathrm{fn}(P\,|\,Q) = \mathrm{fn}(P) \cup \mathrm{fn}(Q) \quad \mathrm{fn}((x)\,P) = \mathrm{fn}(P) \setminus \{x\}$$

We write $\mathrm{n}(P)$ for the set of *names* of a process P and $\mathrm{bn}(P) = \mathrm{n}(P) \setminus \mathrm{fn}(P)$ for the set of *bound names*; the usual notion of α-conversion on bound names holds. By σP we denote the process obtained from P by simultaneously substituting each free occurrence of z in P by $\sigma(z)$, possibly α-converting bound names.

4.2 Operational Semantics

The reduction semantics, as usual, is given in two steps: the definition of a *structural congruence*, which rearranges processes into adjacent positions, and a notion of *reduction relation* that captures computations.

Definition 7. *We let* structural congruence, $\equiv$, *be the least congruence over processes closed with respect to α-conversion and satisfying the following rules.*

$$(\textsc{Ax-Par}) \quad P|0 \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R)$$
$$(\textsc{Ax-Res}) \quad (x)0 \equiv 0 \quad (x)(y)P \equiv (y)(x)P \quad P|(x)Q \equiv (x)(P|Q) \quad \textit{if } x \notin \mathrm{fn}(P)$$
$$(\textsc{Ax-Rec}) \quad D(\tilde{y}) \equiv [\tilde{y}/\tilde{x}]U \quad \textit{if } D(\tilde{x}) \stackrel{\text{def}}{=} U$$

These axioms can be applied for reducing every process P into a normal form $(x_1)\ldots(x_n)\,(C\,|\,U)$, where C is a parallel composition of constraints and U is an unconstrained process. Specifically, the axioms are applied from left to right in the following order: (Ax-Res) for moving forward restrictions, and (Ax-Par) for grouping constraints together, and (Ax-Rec).

Definition 8. *The* reduction relation *over processes $\rightarrow$ is the least relation satisfying the following inference rules. We use the following notations: 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 $(c_1 \times \ldots \times c_n) \leq c$; $C - c$ 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.*

$$(\textsc{tau})\ C\,|\,\tau.U \rightarrow C\,|\,U \qquad (\textsc{tell})\ C\,|\,\texttt{tell}\ c.U \rightarrow C\,|\,c\,|\,U \ \ \textit{if } C\,|\,c \textit{ consistent}$$

$$(\textsc{ask})\ C\,|\,\texttt{ask}\ c.U \rightarrow C\,|\,U \quad \textit{if } C \vdash c \quad (\textsc{retract})\ C\,|\,\texttt{retract}\ c.U \rightarrow (C - c)\,|\,U$$

$$(\textsc{check})\ C\,|\,\texttt{check}\ c.U \rightarrow C\,|\,U \ \ \textit{if } C\,|\,c \textit{ consistent}$$

$$(\textsc{com})\ C\,|\,(\overline{x}\langle\tilde{y}\rangle.U + \textstyle\sum_i \pi_i.U_i)\,|\,(z\langle\widetilde{w}\rangle.V + \sum_j \pi'_j.V_j) \ \longrightarrow\ C\,|\,\{\widetilde{y} = \widetilde{w}\}\,|\,U\,|\,V$$
$$\textit{if } |\widetilde{y}| = |\widetilde{w}|,\ C\,|\,\widetilde{y} = \widetilde{w} \textit{ consistent and } C \vdash x = z$$

$$(\textsc{sum})\ \frac{C\,|\,\pi_i.U_i \rightarrow P}{C\,|\,\sum_i \pi_i.U_i \rightarrow P} \qquad (\textsc{par})\ \frac{P \rightarrow P'}{P\,|\,U \rightarrow P'\,|\,U}$$

$$(\textsc{res})\ \frac{P \rightarrow P'}{(x)\,P \rightarrow (x)\,P'} \qquad (\textsc{struct})\ \frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}$$

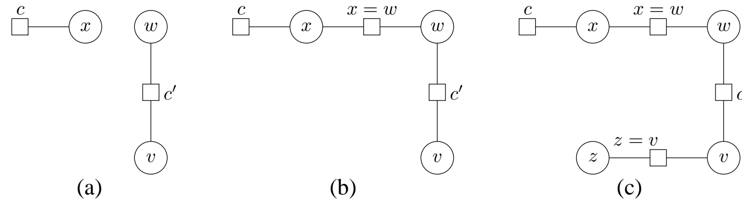
The idea behind this reduction relation is to proceed as follows. First, rearranging processes into the normal form $(x_1)\ldots(x_n)(C\,|\,U)$ by means of rule (STRUCT). Next, applying the rules (TELL), (ASK), (RETRACT), and (CHECK) for primitives on constraints and the rule (COM) for synchronising 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 is stuck otherwise. Rules (ASK) and (CHECK) specify that a process starting with an $\mathtt{ask}\ c$ or, respectively, $\mathtt{check}\ c$ prefix evolves to its continuation if c is entailed by C or, respectively, if $c\,|\,C$ is consistent, and that the process is stuck otherwise. By rule (RETRACT) a process can remove c if c is among the syntactic constraints in C; e.g., the process $x{=}y\,|\,y=z\,|\,\mathtt{retract}\ x=z.U$ does not affect $x{=}y\,|\,y=z$. In rules (COM), 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 if the equality of the names x and z is entailed by C and if the parallel composition $C\,|\,\widetilde{y}=\widetilde{w}$ is consistent. Note that it is legal to treat fusions as constraints c over $\mathcal{C}$, because we only consider named c-semiring with fusions, as noted in § 3. Rule (PAR) allows for closure with respect to unconstrained processes in parallel. This rule imposes to take into account all constraints in parallel when applying the rules for constraints and synchronisation.

The present semantics does not specify how to solve at each step the constraint system given by the parallel composition of constraints C. However, in [10] it is shown how to apply dynamic programming to solve a CSP by solving its subproblems and then by combining solutions to obtain the solution of the whole problem. A visual representation of the problem is given by considering a graph where names are represented as nodes and constraints as arcs connecting the names involved in each constraint.

Example 4. Let P and Q be the following two processes (we write $c(x_1,\ldots,x_n)$ for a constraint c with support $\mathrm{supp}(c)=\{x_1,\ldots,x_n\}$):

$$P \equiv (x)(z)\,\mathtt{tell}\ c(x).\overline{y}\langle x\rangle.x\langle z\rangle.\mathbf{0} \qquad Q \equiv (w)\,\mathtt{tell}\ c'(w,v).y\langle w\rangle.\overline{w}\langle v\rangle.\mathbf{0}$$

First, P and Q make their respective $\mathtt{tell}$ actions, which necessarily succeed as the constraint system is initially empty and the constraints c and c' have different support. The graph representation of the resulting store of constraints is depicted in Fig. (a) below. Next, the two processes try to synchronise on port y and, according to rule (COM), the synchronisation takes place if the constraint combination $c\times c'\times x=w$ has a solution (Fig. (b)). Finally, the processes synchronise on port x, which is identified to w, thus yielding the fusion $z=v$ (Fig. (c)).



Remark on $\mathtt{retract}$ We have chosen to introduce the $\mathtt{retract}$ operation in the calculus in order to model non-monotonic constraint systems. For instance, an agent can

perform a `retract` action for removing from a store a constraint that it had previously placed, thus enabling a `tell` operation which would be stuck otherwise or for releasing some resources after using them. Nevertheless, we can consider a version of the cc-pi calculus not including the `retract` primitive. For this fragment of the calculus the following additional axioms for relating parallel composition with product and restriction with hiding hold: $c_1 \,|\, c_2 \equiv c_1 \times c_2$ and $(x)\,c \equiv \nu x.c$. These axioms cannot be included in the original cc-pi calculus. In fact, a constraint c can be removed only if c is syntactically present in the store of constraints, while by applying product or hiding we generate new syntactic constraints. Note that the axioms for structural congruence in Def. 7 along with the above ones lead to processes into a normal form $(\tilde{x})\,(c\,|\,U)$, where U does not contain restrictions.

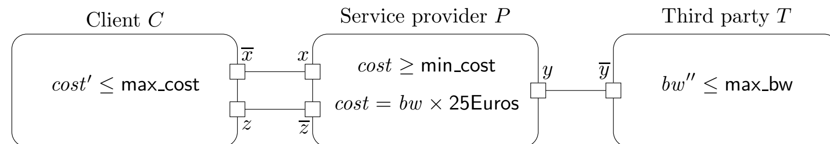
5 Specifying Service Level Agreements

In this section we show how to model within our framework SLA contracts. The idea is to specify each SLA parameter as a variable and each SLA requirement or guarantee as a constraint that connects the involved variables. The parties are modelled as communicating processes. A constraint can be generated either by a single process or by the synchronisation of two processes that induces the identification of the communicated values. Note that our constraint-based approach allows to specify not only negotiations to reach a SLA contract, but also run-time checks that the contracts is not violated by the involved parties.

Here we consider two examples that show how to apply our approach in modelling the SLA management system. The first example is centered around the basic mechanism for reaching and validating a contract. The second example extends the example given in the introduction with three clients. For simplicity, in both examples we take the constraint system to be a CSP by instantiating cc-pi with the named c-semiring $\mathcal{C}_{\text{soft}}$, defined in Example 3, over the c-semiring $S = \langle\{\mathsf{False}, \mathsf{True}\}, \vee, \wedge, \mathsf{False}, \mathsf{True}\rangle$. This choice leads to solutions consisting of the set of tuples of legal domain values. We could generalise such constraint system with soft constraints by replacing S with an arbitrary c-semiring.

5.1 A Web Hosting Service

Consider a service that offers different web hosting solutions, varying in cost and in bandwidth. Let P be the service provider and C be a client. Suppose that P obtains its bandwidth resources from a third party T. Before the execution of the service, P and C want to sign a SLA contract. The success of such an agreement also depends on the resources provided by T. This scenario is depicted below.

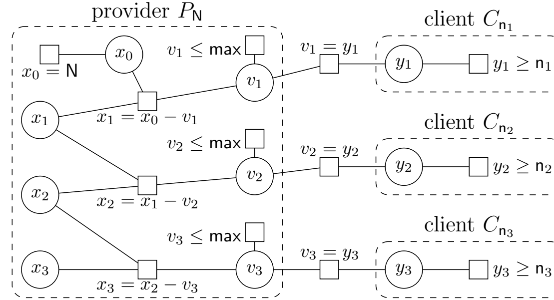


The interaction protocol is as follows. Each party imposes its SLA requirements or guarantees: P specifies the minimum cost $\mathsf{min_cost}$ for the service and the cost per unit of bandwidth $cost = bw \cdot \mathsf{25_Euros}$, C imposes a maximum cost $\mathsf{max_cost}$ it can pay for the service, and T fixes the maximum bandwidth $\mathsf{max_bw}$ that it can supply. Next, P communicates with C and with T on ports x and y, respectively. If the above constraints are consistent with each other, i.e. if $c = ((\mathsf{min_cost} \leq cost \leq \mathsf{max_cost}) \times (bw \leq \mathsf{max_bw}) \times (cost = bw \cdot \mathsf{25_Euros}) \times (cost = cost') \times (bw = bw' = bw'')) \neq 0$, P and C can sign this contract c, by synchronising on z. Then, the service is executed and, assuming P provides C with a certain bandwidth $\mathsf{act_bw}$ and the corresponding cost, the two parties validate the contract by performing `check` operations on their respective parameters. Note that the semantics of `check` enables this validation while not modifying the constraints of the contract. The specification of P, C, and T in cc-pi is as below and the whole system is represented by the parallel composition of the three parties.

$$
\begin{array}{rcl}
P_{\mathsf{min_cost}} & \equiv & (bw)(cost)(\mathtt{tell}\ ((cost \geq \mathsf{min_cost}) \times (cost = bw \cdot \mathsf{25_Euros})). \\
 & & x\langle bw, cost\rangle.\overline{y}\langle bw\rangle.\overline{z}\langle\rangle.P'_{\mathsf{act_bw}}) \\
P'_{\mathsf{act_bw}} & \equiv & \overline{x}\langle \mathsf{act_bw}, \mathsf{act_bw} \cdot \mathsf{25_Euros}\rangle.\mathtt{check}((\mathsf{act_bw} \cdot \mathsf{25_Euros} = cost) \times \\
 & & (\mathsf{act_bw} = bw)).z\langle\rangle \\
C_{\mathsf{max_cost}} & \equiv & (bw')(cost')(\mathtt{tell}\ (cost' \leq \mathsf{max_cost}).\overline{x}\langle bw', cost'\rangle.z\langle\rangle.C') \\
C' & \equiv & (b', c')(x\langle b', c'\rangle.\mathtt{check}((b' = bw') \times (c' = cost')).\overline{z}\langle\rangle) \\
T_{\mathsf{max_bw}} & \equiv & (bw'')(\mathtt{tell}\ (bw'' \leq \mathsf{max_bw}).y\langle bw''\rangle)
\end{array}
$$

5.2 Resource Allocation

We consider a slightly more complex scenario of the example given in the introduction with one provider P_{N} and three clients C_{n_1}, C_{n_2}, and C_{n_3}. The graph representation of the constraint system resulting from the negotiation among the parties is depicted below. Each node represents a variable, and each constraint is modelled by a hyperedge connecting the variables involved in the constraint.



Suppose that P_{N} has allocated the resources y_1 and y_2, with $y_i \geq n_i$ for $i = 1,2$, to C_{n_1} and C_{n_2}, respectively. If C_{n_3} makes a request $y_3 \geq n_3$ that P_{N} is not able to satisfy because $n_1 + n_2 + n_3 \geq \mathsf{N}$, the synchronisation between P_{N} and C_{n_3} cannot take place until some resources y_i, with $y_i \geq n_3$, are released.

6 Expressiveness Results

Encoding Pi-F calculus We start by recalling the Pi-F calculus. For better relating the calculus with cc-pi, we present the Pi-F in the standard pi-calculus fashion rather than in the 'commitment' style [19].

Definition 9. *The syntax of Pi-F processes is the same as the one given in Definition 6 minus summation,* `tell`*,* `ask`*,* `retract`*, and* `check` *and where constraints c are taken over the named c-semiring of equalities* $\mathcal{C}_E$ *defined in Example 1. The structural congruence* $\equiv_F$ *is as in Def. 7 plus the axioms below:*

$$x{=}x \equiv_F \mathbf{0} \qquad x{=}y \equiv_F y{=}x \qquad x{=}y \,|\, y{=}z \equiv_F x{=}z \,|\, y{=}z \qquad (x)\,(x{=}y) \equiv_F \mathbf{0}$$
$$x{=}y \,|\, x\langle z\rangle.P \equiv_F x{=}y \,|\, y\langle z\rangle.P \qquad x{=}y \,|\, \overline{x}\langle z\rangle.P \equiv_F x{=}y \,|\, \overline{y}\langle z\rangle.P$$
$$z{=}y \,|\, x\langle z\rangle.P \equiv_F z{=}y \,|\, z\langle y\rangle.P \qquad z{=}y \,|\, \overline{x}\langle z\rangle.P \equiv_F z{=}y \,|\, \overline{x}\langle y\rangle.P$$

The reduction relation $\rightarrow_F$ *between processes is the smallest relation closed with respect to* $_ \,|\, _$*,* $(x)\,_$ *and* $_ \equiv_F _$*, which satisfies:*

$$\overline{x}\langle\widetilde{z}\rangle.P \,|\, x\langle\widetilde{w}\rangle.Q \rightarrow_F P \,|\, Q \,|\, \{\widetilde{z} = \widetilde{w}\} \quad \textit{if } |\widetilde{z}| = |\widetilde{w}|$$

Note that this syntax rules out processes containing name fusions under prefixes. This choice follows the analogous restriction applied in cc-pi, which avoids that two processes synchronise and, simultaneously, add some constrains to the store, thus possibly yielding an inconsistency.

Definition 10. *The translation* $[\![_]\!]_F$ *of pi-F processes into cc-pi processes is trivial: it maps pi-F constructs on their homonymous versions in cc-pi.*

Theorem 1. *1. If* $P \rightarrow_F Q$ *then* $[\![P]\!]_F \rightarrow [\![Q]\!]_F$*. 2. If* $[\![P]\!]_F \rightarrow Q'$ *then* $P \rightarrow_F Q$ *and* $Q' \equiv [\![Q]\!]_F$*.*

By exploiting Theorem 1 and similar results proved in [19], it is also possible to give reduction-preserving translations of pi-calculus and Fusion [13] in cc-pi.

Encoding `cc` *programming* First, we briefly recall `cc` programming [15]. For the purpose of a more straightforward translation into cc-pi, we present a slightly modified version of the language. The basic ingredients of the `cc` programming constraint system are a set D of *primitive constraints* or *tokens* and a reflexive and transitive *entailment* relation $\vdash$. A constraint c in a constraint system $\langle \mathcal{P}(D), \vdash \rangle$ is an element of $\mathcal{P}(D)^\star$, i.e. the closure of the powerset $\mathcal{P}(D)$ under entailment. The notion of *consistency* is given by identifying a set of *inconsistent* constraints I. The existential operator on constraints $\exists x\, c$ is formalised in terms of cylindric algebras.

Definition 11. *The syntax of* `cc` *programming is defined as follows:*

$$\begin{array}{lll} \textsc{Prefixes} & \pi ::= & \texttt{tell}\; c \;\big|\; \texttt{ask}\; c \\ \textsc{Processes} & A ::= & \texttt{success} \;\big|\; \pi.A \;\big|\; A\,|\,A \;\big|\; \sum_i \pi_i.A_i \;\big|\; \exists_{x,c} A \;\big|\; p(\tilde{y}) \end{array}$$

Following standard lines [16, 12], we replace the classical hiding operator $\exists_x A$ with $\exists_{x,c} A$ that represents the evolution of a process of the form $\exists_x A'$, where c is the local constraint produced during the evolution. Moreover, $p(\tilde{y})$ is a *procedure call*, where p is the name of the procedure and $\tilde{y}$ is the tuple of actual parameters. The meaning of a process is given with respect to a set of procedure declarations of the form $p(\tilde{x}) := A$. An *instantiation* of $p(\tilde{x}) := A$ is an object of the form $p(\tilde{y}) := [\tilde{y}/\tilde{x}]A$. A *configuration* is a pair $\langle c, A\rangle$ with a constraint c representing the store and a process A. The reduction relation $\rightarrow_C$ over configurations is the smallest relation given by the following rules.

$$(\text{TELL})\ \langle c, \texttt{tell}\ c'.A\rangle \mapsto \langle (c \cup c')^\star, A\rangle \qquad \text{if } (c \cup c')^\star \text{ consistent}$$

$$(\text{ASK})\ \langle c, \texttt{ask}\ c'.A\rangle \mapsto \langle c, A\rangle \qquad \text{if } c \vdash c'$$

$$(\text{SUM})\ \frac{\langle c, \pi_i.A_i\rangle \mapsto \langle c', A'\rangle}{\langle c, \sum_i \pi_i.A_i\rangle \mapsto \langle c', A'\rangle} \qquad (\text{PAR})\ \frac{\langle c, A\rangle \mapsto \langle c', A'\rangle}{\langle c, A \,|\, B\rangle \mapsto \langle c', A' \,|\, B\rangle}$$

$$(\text{PAR}')\ \frac{\langle c, A\rangle \mapsto \langle c', A'\rangle}{\langle c, B \,|\, A\rangle \mapsto \langle c', B \,|\, A'\rangle} \qquad (\text{HIDE})\ \frac{\langle c \cup \exists x\, d, A\rangle \mapsto \langle c' \cup \exists x\, d, A'\rangle}{\langle d, \exists_{x,c} A\rangle \mapsto \langle d, \exists_{x,c'} A'\rangle}$$

$$(\text{PROC})\ \frac{\langle c, [\tilde{y}/\tilde{x}]A\rangle \mapsto \langle c', A'\rangle \quad \text{and} \quad p(\tilde{x}) := A}{\langle c, p(\tilde{y})\rangle \mapsto \langle c', A'\rangle}$$

Definition 12. *The translation* $[\![_]\!]_F$ *of* `cc` *processes in cc-pi is trivial:*

$$[\![\texttt{success}]\!]_C = \mathbf{0} \quad [\![\texttt{ask}\ c.A]\!]_C = \texttt{ask}\ c.[\![A]\!]_C \quad [\![\texttt{tell}\ c.A]\!]_C = \texttt{tell}\ c.[\![A]\!]_C$$
$$[\![A \,|\, B]\!]_C = [\![A]\!]_C \,|\, [\![B]\!]_C \quad [\![\exists_{x,c} A]\!]_C = (x)\,(c \,|\, [\![A]\!]_C) \quad [\![\textstyle\sum_i \pi_i.A_i]\!]_C = \textstyle\sum_i [\![\pi_i.A_i]\!]_C$$
$$[\![p(\tilde{y})]\!]_C = D_p(\tilde{y})$$

where for each `cc` *procedure declaration* $p(\tilde{x}) := A$ *we give a defining equation* $D_p(\tilde{x}) = Q$ *with* $[\![A]\!]_C = Q$.

Lemma 1. *A constraint system* $\langle \mathcal{P}(D), \vdash\rangle$ *can be represented as a named c-semiring* $\langle C, +, \times, \nu x., \rho, 0, 1\rangle$ *with: (i)* $C = \overline{\mathcal{P}(D)} \cup \bot$*, where* $\bot$ *corresponds to the set* I*; (ii)* $c_1 + c_2 = (c_1 \cap c_2)$*; (iii)* $c_1 \times c_2 = (c_1 \cup c_2)^\star$*; (iv)* $\nu x. c = \exists x c$ *and* $\rho c = \rho\, c$*; (v)* $0 = \bot$ *and* $1 = C$.

Theorem 2. *1. If* $\langle c, A\rangle \mapsto \langle c', A'\rangle$ *then* $c \,|\, [\![A]\!]_C \rightarrow c' \,|\, [\![A']\!]_C$*. 2. If* $c \,|\, [\![A]\!]_C \rightarrow P$ *then* $\langle c, A\rangle \mapsto \langle c', A'\rangle$ *and* $P \equiv c' \,|\, [\![A']\!]_C$.

Note that `cc` programming handles the evolution of local stores of constraints through the rule (HIDE), while cc-pi (without `retract`) obtains the same effect by reducing processes into a normal form in which names are conveniently α-converted. This fact plays a crucial role in the proof of Theorem 2.

7 Concluding Remarks

This paper is mainly focused on presenting the cc-pi calculus and on showing its flexibility as a constrained-based model for specifying SLA contracts and resource allocation. We foresee several directions for future work. We plan to consider a distributed

version of the calculus by equipping, e.g., processes with locations and by limiting the synchronous behavior of processes and constraints to a single locality. It would also be interesting to study suitable mechanisms for assuring transactional and security properties of process executions, e.g. by enforcing that only the process which has told a constraint can retract it. We also intend to further study the ability of c-semirings to model soft constraints to express nonfunctional properties of SLAs.

Acknowledgments We thank the anonymous referees for helpful comments.

References

1. A. Bacciu, A. Botta, and H. Melgratti. A fuzzy approach for negotiating quality of services. In *Proc. TGC '06*. To appear.
2. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.
3. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Logic*, 7(3):563–589, 2006.
4. R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A process calculus for QoS-aware applications. In *Proc. COORDINATION'05*, volume 3454 of *Lect. Notes in Comput. Sci*. Springer, 2005.
5. J. F. Diaz, C. Rueda, and F. Valencia. A calculus for concurrent processes with constraints. *CLEI Electronic Journal*, 1(2), 1998.
6. 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.
7. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Inform. and Comput*, 100(1):1–40,41–77, 1992.
8. U. Montanari. Networks of constraints: fundamental properties and application to picture processing. *Information Science*, 7:95–132, 1974.
9. U. Montanari and Pistore M. Structured coalgebras and minimal hd-automata for the pi-calculus. *Theoret. Comput. Sci*, 340(3):539–576, 2005.
10. U. Montanari and F. Rossi. Constraint relaxation may be perfect. *Artif. Intell.*, 48(2):143–170, 1991.
11. J. Niehren and M. Mueller. Constraints for free in concurrent computation. In *Proc. Asian '95*, volume 1023 of *Lect. Notes in Comput. Sci*. Springer, 1995.
12. M. Nielsen, C. Palamidessi, and F. Valencia. On the expressive power of temporal concurrent constraint programming languages. In *Proc. PPDP'02*. ACM, 2002.
13. J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proc. LICS'98*. IEEE, Computer Society Press, 1998.
14. V. Saraswat and P. Lincoln. Higher-order linear concurrent constraint programming, 1992. Technical Report, Xerox Parc.
15. V. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. POPL'90*. ACM Press, 1990.
16. V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. POPL'91*. ACM Press, 1991.
17. J. Skene, D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proc. ICSE'04*, 2004.
18. B. Victor and J. Parrow. Constraints as processes. In *Proc. CONCUR'96*, volume 1119 of *Lect. Notes in Comput. Sci*. Springer, 1996.
19. L. Wischik and P. Gardner. Explicit fusions. *Theoret. Comput. Sci*, 340(3):606–630, 2005.