UNIVERSITÀ DI PISA
DIPARTIMENTO DI INFORMATICA

TECHNICAL REPORT: TR-11-05

Constraints for Service Contracts

Maria Grazia Buscemi Mario Coppo
Mariangiola Dezani-Ciancaglini Ugo Montanari

March 5, 2011

ADDRESS: Largo B. Pontecorvo 3, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Constraints for Service Contracts

Maria Grazia Buscemi[1], Mario Coppo[2],
Mariangiola Dezani-Ciancaglini[2], and Ugo Montanari[3]

[1] IMT Institute for Advanced Studies, Lucca, Italy
[2] Dipartimento di Informatica, Università di Torino, Italy
[3] Dipartimento di Informatica, Università di Pisa, Italy

Abstract. This paper focuses on client-service interactions distinguishing between three phases: negotiate, commit and execute. The participants negotiate their behaviours, and if an agreement is reached they commit and start an execution which is guaranteed to respect the interaction scheme agreed upon. These ideas are materialised through a calculus of contracts enriched with semiring-based constraints, which allow clients to choose services and to interact with them in a safe way. A concrete representation of these constraints with logic programs and logic program combinations is straightforward, thus reducing constraint solution (and consequently the establishment of a contract) to the execution of a logic program.

1 Introduction

Formal methods are fundamental tools to guarantee that the programs we develop behave as desired. The classical approach is based on static analysis: programs and program development methods are analysed in terms of their semantics and of the properties which have to be proved. Recently, however, network based computing has added important open endness requirements, which often make static analysis less than meaningful.

An alternative approach is NCE, Negotiate, Commit, Execute, where agents negotiate certain desired behaviours, but without any guarantee of success. However, if and when an agreement is reached (commit), under certain conditions a coordinated computation of the involved agents can start, which is guaranteed to have the properties agreed upon in the negotiation phase.

In the paper, we focus on the property of stuck-freedom [15] for two-parties sessions, where if an agent is ready to have an interaction, it cannot be stuck forever. Existing work on session types [9] and behavioural contracts [8] mainly focuses on a static analysis phase, which has a yes-no answer: either the session can take place with the desired properties, or not at all. Instead in our approach a negotiation phase between the client and the services guarantees the *best* interaction.

Here we present a simple source calculus with client and service processes and with an LTS semantics: the clients are recursive and can place nested service calls, while services are permanent and nonrecursive. The calculus is nondeterministic, with external choices, and a client-service behaviour which allows for more choices is considered better, provided they are stuck-free. A target calculus is then defined, where clients

and saservices are compiled to, augmenting them with *named constraint semirings* [5], which encode their behaviour.

Constraint semirings are semirings with an idempotent additive operation and a commutative multiplicative operation. They generalise boolean algebra and are equipped with a partial ordering with 1 as maximal and 0 as minimal element. Semiring values model constraints: larger values are less constraining, multiplication means combination of constraints, and addition returns the lub, namely the smaller value larger than both arguments. Among the most relevant semirings we can mention the fuzzy semiring, the tropical or optimisation semiring, the powerdomain semiring. *Named* constraint semirings are equipped in addition with name permutations, interpreted as free names, which imply a notion of support, and with name restriction, understood in the usual way. Named constraint semirings inherit from ordinary constraints on the boolean semiring interesting properties and efficient algorithms, like constraint propagation and dynamic programming.

In the paper we take advantage of a quite simple and standard named constraint semiring: the one employed by logic programming, where the Herbrand signature contains as many unary operations as actions and a constant to model termination. The semiring values are sets of assignments with Herbrand terms to all the names. However, only the assignments to the tuple of free names forming the support are relevant. Addition is union and multiplication is intersection. Restriction adds all the assignments where the restricted variable is assigned in all possible ways. Value 1 is the set of all assignments, and thus its support is empty, while 0 is the empty set. The correspondence with logic programming is quite simple: given a set of clauses and a goal $P(x_1,\ldots,x_n)$, its semantics in terms of our constraint system is the set of all the ground instantiations of the goal which satisfy the clauses. The support is the set $\{x_1,\ldots,x_n\}$ or smaller. Goal composition is multiplication, while multiple clauses for the same goal model addition. Variables appearing in the body but not in the head of a clause represent restriction. In our notation, the effect of recursive clauses is obtained by an explicit fixpoint operator.

To understand our approach, let us first consider the case of a client without nested calls and with a single service. The source client is then compiled into a target client combined with a constraint with just one support name, say x. The constraint is thus just a set of traces, representing the behaviour of the client. The compiled code is very similar to the source, except that its choices are guarded by *check* constructs, similar to the *ask* constructs of concurrent constraint programming, which enable the corresponding continuations only if the global constraint allows it. The source service is also compiled, yielding a constraint on y which represents its behaviour but no check guards are included. The negotiation phase consists of multiplying the two constraints, and their result with the constraint[1] $x = y$. The resulting constraint contains exactly all the executions of the source client-service system which are not stuck. If the constraint is not 0, i.e. if it is not the empty set, the commit takes place, and the execution phase can start. Thanks to the check guards, the traces possible for the target client-service system are exactly those in the constraint. Notice that while the client (service) compilation is static, and thus it does not fit in the NCE scheme, it does not depend on the particular

[1] Constraint $x = y$ has support $\{x,y\}$ and contains all assignments with the same term for x and y. In logic programming it is specified by the goal $eq(x,y)$ with the clause eq(x,x):-

service (client) partner it will be matched to. Thus the open endness requirement is here satisfied by the possibility of introducing into the system new services (clients) without the need of any further modification.

For instance, let $T = \overline{\Box}.\overline{\alpha}_1.\overline{\boxtimes}.\mathbf{0} + \overline{\alpha}_2.\overline{\boxtimes}.\mathbf{0}$ be a client and $S = \Box.\alpha_1.\boxtimes + \alpha_3.\boxtimes$ be a service, where $\overline{\Box}$ is service call, $\overline{\boxtimes}$ is call end, α ranges over actions and the co-symbols have the expected meaning. The interactions offered by T and S are represented, respectively, by the constraints $c = eq(x,\alpha_1(end)) \oplus eq(x,\alpha_2(end))$ and $d = eq(y,\alpha_1(end)) \oplus eq(y,\alpha_3(end))$ (noting that constraints disregard co-actions). It holds that $c \otimes d \otimes eq(x,y) = eq(x,\alpha_1(end)) \otimes eq(y,\alpha_1(end)) \otimes eq(x,y) \neq 0$, which reflects the fact that the only successful interaction between T and S is (over) α_1.

Let us now consider the general case of a client with nested calls and several services. Services are compiled in the usual way, but their behaviours are added up. The behaviour of the client, instead, must be represented by a constraint with several names in the support. In fact, different service calls may not be independent: imagine that the client in an inner call makes a choice which must be matched by the corresponding service. Then the client returns to the outer level and makes another choice which must be matched this time by the service corresponding to the outer call. The two choices may be dependent, and this requirement is represented by a constraint with a two-name support. Thus the ability of the constraint system of representing sets of tuples of traces (and not only tuples of sets of traces) allows us to guarantee stuck-freedom for complex client-service pairs, which at the best of our knowledge have not been considered in the literature by now.

An example of this kind of clients is $\overline{\Box}.\overline{\alpha}.\overline{\Box\beta}.(\overline{\gamma}.\overline{\boxtimes}.\overline{\delta}.\overline{\boxtimes}.\mathbf{0} + \overline{\mu}.\overline{\boxtimes}.\overline{\rho}.\overline{\boxtimes}.\mathbf{0})$, where the choices made by the two nested calls depend on each other. Such a dependency is not at all unusual: for instance it can arise when modelling a traveller who asks both for two-ways flies to an airline company and for one room to an hotel. The resulting constraint $eq(x_1,\alpha(\delta(end))) \otimes eq(x_2,\beta(\gamma(end))) \oplus eq(x_1,\alpha(\rho(end))) \otimes eq(x_2,\beta(\mu(end)))$, with support $\{x_1,x_2\}$, obliges the run of each service call to be coherent with the run of the other.

In the paper, the source and the target calculus are the content of Sections 2 and 3, respectively. Section 4 presents the compilation from the first to the second calculus and states its soundness and completeness. Section 5 shows how to use the constraints in order to get most liberal interactions. Lastly Sections 6 and 7 discuss related papers and some future developments. Proofs are only sketched.

2 Source Calculus

In choosing the source calculus we were guided by the requirement of having a minimal number of process operators to represent one client with (nested) service calls and a set of available services offering finite interactions. For this reason communications are atomic actions, all choices are external, client processes are recursive and can do nested calls, services are permanent, non recursive and each instance of a service can accept exactly one call.

Let $\mathcal{N}$ be an infinite set of names (ranged over by $\alpha,\beta,\ldots$) representing actions and $\overline{\mathcal{N}}$ be an infinite set of disjoint co-names (ranged over by $\overline{\alpha},\overline{\beta},\ldots$) representing co-

Source client processes

$$P := \overline{\Box}.P \;\Big|\; \overline{\alpha}.P \;\Big|\; P+P \;\Big|\; rec\,p.P \;\Big|\; \overline{\boxtimes}.P \;\Big|\; p \;\Big|\; \mathbf{0}$$

Source service processes

$$Q := \alpha.Q \;\Big|\; Q+Q \;\Big|\; \boxtimes$$

Table 1. Syntax of source client and service processes.

actions, with as usual $\overline{\overline{\alpha}} = \alpha$. The syntax of client and service processes is the content of Table 1. For the sake of readability we assume that client processes offer co-actions and service processes offer actions, $+$ is the external choice, $\overline{\Box}$ is service call and $\overline{\boxtimes}, \boxtimes$ are end of call for client and service processes, respectively.

Recursive processes cannot have nested calls (namely, service calls cannot occur under the scope of the *rec* operator). We will consider recursive processes modulo fold/unfold, i.e. we identify the processes $rec\,p.P$ and $P[p := rec\,p.P]$.

A *client* is a client process of the shape $\overline{\Box}.P$. A client is *balanced* if each $\overline{\Box}$ is followed by a corresponding $\overline{\boxtimes}$ in each branch. We use T to range over balanced clients.

A *service* is a service process prefixed by $\Box$ (representing call acceptance), i.e. it has the shape $\Box.Q$. We use S to range over services.

We consider *systems* formed by one balanced client and a set $\mathbb{S}_s$ of available services. The interaction between a client and a service is represented by boxing the parallel composition of the client process which follows the call with the interacting service process (see rule $(s\text{-}call)$ in Table 2). Therefore by reducing a balanced client we will get a box containing either the parallel composition of a client process and a service process or the parallel composition of a box and a service process. We get then the following syntax for systems V:

$$V ::= T \;\Big|\; [U] \;\Big|\; \mathbf{0} \qquad\qquad U ::= P \mid Q \;\Big|\; [U] \mid Q$$

Since we want to compare reductions in the source and in the target calculus we give the operational semantics of both calculi via labelled transition systems. We say that the process Q *exhibits the service end* (notation $Q \downarrow_{\boxtimes}$) if either Q is $\boxtimes$ or Q is the sum of two processes one of which exhibits the service end.
Table 2 gives the reduction rules for the source calculus, where W denotes a client process, a service process, or a box, $\lambda \in \mathcal{N} \cup \overline{\mathcal{N}}$, $\phi \in \{[, \overline{\boxtimes}\} \cup \mathcal{N} \cup \overline{\mathcal{N}}$ and $\psi \in \{[,]\} \cup \mathcal{N}$. The symmetric rule with respect to $+$ has been omitted. Rule $(s\text{-}up)$ terminates the execution of a call leaving the client process which follows this call one nesting level up.
Let σ be a strings on $\{[,]\} \cup \mathcal{N}$. We define $V \stackrel{\sigma}{\Longrightarrow} V'$ if

- either $\sigma = \varepsilon$ and $V = V'$
- or $\sigma = \psi\sigma'$ and $V \xrightarrow{\psi} \stackrel{\sigma'}{\Longrightarrow} V'$.

We write $V \Downarrow^{may}$ if $\exists \sigma$ such that $V \stackrel{\sigma}{\Longrightarrow} \mathbf{0}$.

$$\frac{\Box.Q \in \mathbb{S}_s}{\overline{\Box}.P \xrightarrow{\lfloor} [P \mid Q]}\ (s\text{-}call) \qquad \lambda.W \xrightarrow{\lambda} W\ (s\text{-}action) \qquad \overline{\boxtimes}.P \xrightarrow{\overline{\boxtimes}} P\ (s\text{-}end)$$

$$\frac{W \xrightarrow{\phi} W'}{W + W_1 \xrightarrow{\phi} W'}\ (s\text{-}choice) \qquad \frac{P \xrightarrow{\bar{\alpha}} P' \quad Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P' \mid Q'}\ (s\text{-}interaction)$$

$$\frac{P \xrightarrow{\overline{\boxtimes}} P' \quad Q \downarrow_{\boxtimes}}{[P \mid Q] \xrightarrow{\rfloor} P'}\ (s\text{-}up) \qquad \frac{U \xrightarrow{\Psi} U'}{[U] \xrightarrow{\Psi} [U']}\ (s\text{-}box) \qquad \frac{V \xrightarrow{\Psi} V'}{V \mid Q \xrightarrow{\Psi} V' \mid Q}\ (s\text{-}parallel)$$

Table 2. LTS for the source calculus

3 Target Calculus

The target calculus enriches the source calculus by introducing constraints, which are meant to prevent clients and services from initiating interactions that will eventually lead to deadlocks. The constraints we adopt coincide with those used in logic programming and form a *named constraint semiring* [5], which is a constraint semiring [2] along with a notion of relevant names that allows plugging constraints into languages having an explicit concept of names. Hereafter, we use the term 'variable' rather to 'name' in conformance with the standard notation of logic programming. Below we introduce the formal definitions which are used throughout the paper. We refer to [5, 2, 1] for a more detailed treatment.

3.1 Constraints

Let $\mathcal{V}$ be a set of variables, ranged over by $x, y, z, \ldots$. Assume a signature Σ consisting of monadic functions corresponding to actions plus a constant *end* (which expresses the successful end of a process):

$$\Sigma = \{\alpha(\_), end \mid \alpha \in \mathcal{N}\}$$

A term is any expression that can be obtained from $\mathcal{V}$ and Σ; a ground term is a term that does not contain variables. The Herbrand Universe $\mathcal{H}$, ranged over by h, is the set of ground terms over Σ. A ground assignment is a total function s that maps variables to ground terms, namely $s : \mathcal{V} \to \mathcal{H}$.

A *constraint* is a set of ground assignments and we let $\mathcal{C}$, ranged over by c, be the set of all constraints. The restriction operator $\exists x$ over a constraint c makes a variable x local in c, and is defined as $\exists x.c = \{s[h/x] \mid h \in \mathcal{H}, s \in c\}$. Of course, the ordering of $\exists x$'s in a constraints is irrelevant. Thus, we can conveniently write $\exists X$ in place of $\exists x_1, \ldots, \exists x_n$, for $X = \{x_1, \ldots, x_n\}$. The *support* of a constraint $\mathrm{supp}(c)$ is the minimum set $\mathcal{W}$ of variables such that $x \in \mathcal{W}$ implies that $\exists x.c \neq c$. Intuitively, the support of a constraint c contains all the variables which are *relevant* for c. The notion of support corresponds with the concept of set of free variables in process calculi. We abbreviate the notation for constraints by disregarding variables which are not in the support. Hence, by $eq(x, y)$ we write the constraint c with support $\{x, y\}$ that contains all assignments with the same

ground term for x and y, namely $c = \{s_1, s_2, \ldots\}$ such that $s_i = [h_i/x, h_i/y, h_i'/z \ldots]$ with $z \notin \mathrm{supp}(c)$ and $z \neq x, y$ for all $z \in \mathcal{V}$. Similarly, by $eq(x, end)$ we denote the constraint consisting of all the assignments that map x to end. The set of constraints $\mathcal{C}$ can be proved to be a named constraint semiring [5] by taking product $\otimes$, sum $\oplus$, 0, and 1 as follows:

- $c \oplus d = c \cup d$;
- $c \otimes d = c \cap d$;
- The bottom element 0 is the empyset;
- The top element 1 is the constraint with empty support.

We also define a recursive operator $rec_x \mathsf{c}$ over constraints. Recursive constraint variables c_y contain (except that in the binding construct) an extra variable that is used when unfolding the constraint. All occurrences of a c_y in c are bound by $rec_x \mathsf{c}$ in $rec_x \mathsf{c}.c$, independently of the variable appearing in the subscript. Recursive constraints are folded/unfolded using the following equation:

$$rec_x\, \mathsf{c}.\exists X\, c = \exists X\, .c[{}^{c_i}/_{\mathsf{c}_{y_i}} \mid i \in I]$$

with $c_i = rec_{y_i}\, \mathsf{c}.(X)c[{}^{y_i}/_{x}]$ and y_i for $i \in I$ the variables occurring as indexes of c in c. The (solution of the) recursive constraint $rec_x \mathsf{c}.c$ can be defined as a least fixpoint, which exists because the operations on constraints are continuous and are defined over a domain that is a lattice.

It can be easily proved that constraints coincide with the ground semantics of logic programs [12]. As an example, consider a recursive constraint that amounts to assigning to a variable z an arbitrary number of α_2's followed by α_2 and, subsequently, by (end), namely:

$$\begin{aligned} c = rec_z \mathsf{c}.\exists x_1, x_2, y_1, y_2.(eq(z, x_1) \oplus eq(z, y_1)) \otimes eq(x_1, \alpha_1(x_2)) \otimes \\ eq(x_2, end) \otimes eq(y_1, \alpha_2(y_2)) \otimes \mathsf{c}_{y_2} \end{aligned}$$

The logic program corresponding to c is given by the goal `F(z)` along with the following clauses:

```
F(x1) :- G1(x1)
F(y1) :- H1(y1)
G1(α1(x2)) :- G2(x2)
G2(end).
H1(α2(y2)) :- H2(y2)
H2(y2):- F(y2)
```

3.2 Syntax and Semantics

The syntax of target calculus (see Table 3) is analogous to the syntax of the source calculus apart for the introduction in the target processes of constraints $c \in \mathcal{C}$, as defined in §3.1, over possibly restricted variables. A service calls $\overline{\Box}\langle x \rangle$ includes the 'root' variable x of the constraint representing the interaction offered by the client to the invoked service. The process $\texttt{check}\ \ c_1.\mathsf{P}_1, \texttt{check}\ \ c_2.\mathsf{P}_2$ generalises the external choice by evolving to the process P_i if $c_i \otimes c \neq 0$, with c the current constraint store and $i = 1, 2$.

Akin to recursive constraints, recursive processes are also decorated with a source process recursion variable p. All occurrences of p_x^p in R are bound by $rec_y \mathsf{p}^p$ in $rec_y \mathsf{p}^p.\mathsf{R}$. We identify recursive processes via the following fold/unfold equation

$$rec_x\, \mathsf{p}^p.(X)c \mid \mathsf{R} = (X)c \mid \mathsf{R}[{}^{\mathsf{P}_i}/_{\mathsf{p}_{y_i}^p} \mid i \in I]$$

where $\mathsf{P}_i = rec_{y_i}\, p.(X)c[{}^{y_i}/_x] \mid \mathsf{R}[{}^{y_i}/_x]$ and y_i for $i \in I$ are all the variables occurring as indexes of p^p in R.

Target client processes

$$\mathsf{P} := \overline{\Box}\langle x\rangle.\mathsf{P} \mid \overline{\alpha}.\mathsf{P} \mid (\texttt{check}\ \ c_1.\mathsf{P}_1, \texttt{check}\ \ c_2.\mathsf{P}_2) \mid \mathsf{p}_x^p \mid rec_x \mathsf{p}^p.\mathsf{R} \mid \overline{\boxtimes}.\mathsf{P} \mid \mathbf{0} \qquad \mathsf{R} := (X)c \mid \mathsf{P}$$

Target service processes

$$\mathsf{Q} := \alpha.\mathsf{Q} \mid \mathsf{Q} + \mathsf{Q} \mid \boxtimes$$

Table 3. Syntax of target client and service processes.

$$\frac{(X)\,d \mid \Box\langle x\rangle.\mathsf{Q} \in \mathbb{S}_t \quad c \otimes d \neq 0}{c \mid \overline{\Box}\langle x\rangle.\mathsf{P} \xrightarrow{[} (X)[c \otimes d \mid \mathsf{P} \mid \mathsf{Q}]}\ (t\text{-}call) \qquad c \mid \overline{\boxtimes}.\mathsf{P} \xrightarrow{\overline{\boxtimes}} c \mid \mathsf{P}\ \ (t\text{-}end)$$

$$c \mid \overline{\alpha}.\mathsf{P} \xrightarrow{\overline{\alpha}} c \mid \mathsf{P}\ \ (t\text{-}action_C) \qquad \alpha.\mathsf{Q} \xrightarrow{\alpha} \mathsf{Q}\ \ (t\text{-}action_S)$$

$$\frac{c \mid \mathsf{P}_i \xrightarrow{\varphi} \mathsf{W} \qquad c \otimes c_i \neq 0}{c \mid \texttt{check}\ \ c_1.\mathsf{P}_1, \texttt{check}\ \ c_2.\mathsf{P}_2 \xrightarrow{\varphi} \mathsf{W}}\ (t\text{-}check) \qquad \frac{\mathsf{Q} \xrightarrow{\alpha} \mathsf{Q}'}{\mathsf{Q} + \mathsf{Q}_1 \xrightarrow{\alpha} \mathsf{Q}'}\ (t\text{-}choice)$$

$$\frac{c \mid \mathsf{P} \xrightarrow{\overline{\alpha}} c \mid \mathsf{P}' \quad \mathsf{Q} \xrightarrow{\alpha} \mathsf{Q}'}{c \mid \mathsf{P} \mid \mathsf{Q} \xrightarrow{\alpha} c \mid \mathsf{P}' \mid \mathsf{Q}'}\ (t\text{-}interaction) \qquad \frac{c \mid \mathsf{P} \xrightarrow{\overline{\boxtimes}} c \mid \mathsf{P}' \quad \mathsf{Q} \downarrow_{\boxtimes}}{[c \mid \mathsf{P} \mid \mathsf{Q}] \xrightarrow{]} c \mid \mathsf{P}'}\ (t\text{-}up)$$

$$\frac{\mathsf{U} \xrightarrow{\psi} \mathsf{U}'}{[\mathsf{U}] \xrightarrow{\psi} [\mathsf{U}']}\ (t\text{-}box) \qquad \frac{\mathsf{V} \xrightarrow{\psi} \mathsf{V}'}{\mathsf{V} \mid \mathsf{Q} \xrightarrow{\psi} \mathsf{V}' \mid \mathsf{Q}}\ (t\text{-}parallel) \qquad \frac{\mathsf{V} \xrightarrow{\psi} \mathsf{V}'}{(X)\mathsf{V} \xrightarrow{\psi} (X)\mathsf{V}'}\ (t\text{-}restr)$$

Table 4. LTS for the target calculus

A *target client* has the shape $(X)c \mid \overline{\Box}\langle x\rangle.\mathsf{P}$. We use T to range over target clients. A *target service* has the shape $(X)c \mid \Box\langle x\rangle.\mathsf{Q}$. We use S to range over services. *Target systems* V are the same as source systems apart for the fact that the parallel composition of a client and a service includes a constraint and that a set X of variables can be restricted in a box:

$$\mathsf{U} ::= c \mid \mathsf{P} \mid \mathsf{Q} \ \Big|\ (X)[\mathsf{U}] \mid \mathsf{Q} \qquad\qquad \mathsf{V} ::= \mathsf{T} \ \Big|\ (X)[\mathsf{U}] \ \Big|\ (X)c \mid \mathbf{0}$$

Assume a set of available target services $\mathbb{S}_t$. The labelled transition system is reported in Table 4, where $\varphi \in \{[, \overline{\boxtimes}\} \cup \overline{\mathcal{N}}$, $\psi \in \{[,]\} \cup \mathcal{N}$, $\mathsf{Q} \downarrow_{\boxtimes}$ is defined as for source

processes, W stands for $c \mid \mathsf{P}$ or $(X)[\mathsf{U}]$, and the symmetric rule with respect to $+$ has been omitted. The rules are the same as their homologous ones in the source target, taking into account that systems contain constraints. Rule $(t\text{-}check)$ activates a process continuation P_i only if the corresponding guard is consistent with the constraint store (condition $c \otimes c_i \neq 0$).

Below we characterise the shape of the initial state of the LTS, which plays a key role in our theory as the existence of the initial state is meant to guarantee absence of deadlocks. To this purpose, we first define a function $I(\_)$ that applied to a target client gives a set Y of sets of variables which start the service calls. Each set of Y contains the variables corresponding to a different execution path. Note that Y cannot be infinite because recursive clients cannot have nested calls. For Y a collection of sets, we let $\{x\} \uplus Y = \{\{x\} \cup Z \mid Z \in Y\}$.

$$I(\overline{\square}\langle x\rangle.\mathsf{P}) = \{x\} \uplus I(\mathsf{P}) \qquad I(\overline{\alpha}.\mathsf{P}) = I(\mathsf{P}) \qquad I(\mathbf{0}) = \{\emptyset\} \qquad I(\overline{\boxtimes}.\mathsf{P}) = I(\mathsf{P})$$
$$I((\texttt{check } c_1.\mathsf{P}_1, \texttt{check } c_2.\mathsf{P}_2)) = I(\mathsf{P}_1) \cup I(\mathsf{P}_2) \quad I(\mathsf{p}_x^p) = \{\emptyset\} \quad I(rec_x \mathsf{p}^p.\mathsf{R}) = \{\emptyset\}$$

Assume a client T and a set of available services $\mathbb{S}_t$. Suppose

$$\mathsf{T} = (X)c_0 \mid \overline{\square}\langle x\rangle.\mathsf{P} \qquad \mathbb{S}_t = \{(X_i)\, c_i \mid \square\langle x_i\rangle.\mathsf{Q}_i \mid i \in I\}$$
$$I(\overline{\square}\langle x\rangle.\mathsf{P}) = Y \qquad d_Y = \bigoplus_{Z \in Y} \bigotimes_{z \in Z} \bigoplus_{i \in I} (X_i)\, c_i \otimes eq(z, x_i)$$

If $c_0 \otimes d_Y \neq 0$ then $\texttt{start}(\mathsf{T}, \mathbb{S}_t) = (X \cup \bigcup_{Z \in Y} Z) c_0 \otimes d_Y \mid \overline{\square}\langle x\rangle.\mathsf{P}$ is the *initial state* of the labelled transition system in Table 4.

The consistency check $c_0 \otimes d_Y$ amounts to say that for at least a set $Z \in Y$, every service call corresponding to a $z \in Z$ can successfully interact with a service in $\mathbb{S}_t$. Remark that rule $(t\text{-}call)$ is applied with c being the product of c_0, the constraint of the client, and the constraints d_Y of services interacting with service calls. Hence, the condition $c \otimes d \neq 0$ not only ensures that the interaction with the actual service will be successful, but also that any subsequent service call will be able not to lead to a stuck state.

We define $\mathsf{V} \stackrel{\sigma}{\Longrightarrow}$ as expected. We say that V is *satisfied* if it is $(X)c \mid \mathbf{0}$ for some X, c. We define $\mathsf{V} \Downarrow^{must}$ if either V is satisfied or for all σ, V' such that $\mathsf{V} \stackrel{\sigma}{\Longrightarrow} \mathsf{V}'$ we have $\mathsf{V}' \Downarrow^{must}$.

4 Compilation

We map the source calculus into the target calculus by adding constraints which take into account the interactions offered by the processes. This allows us to model the negotiation phase which precedes the choice of a service.

The compilation of services (Table 5) is simple, since the syntax of source and target processes is the same, when forgetting constraints and restrictions. This compilation adds an appropriate constraint for each process constructor by introducing a fresh variable x which is equated to *end* for $\boxtimes$, to $\alpha\langle y\rangle$ for $\alpha.Q$ (where y is the variable introduced by the compilation of Q), to x_1 or x_2 for $Q_1 + Q_2$ (where x_1, x_2 are the variables introduced by the compilation of Q_1, Q_2, respectively). Lastly the compilation of a service $\square.Q$ uses the variable x introduced by the compilation of Q to get $\square\langle x\rangle.\mathsf{Q}$, where Q

$$\{[\boxtimes]\}_x = eq(x,end) \mid \boxtimes$$

$$\{[Q]\}_y = (X)c \mid \mathsf{Q} \text{ implies } \{[\alpha.Q]\}_x = (X \cup \{y\})c \otimes eq(x,\alpha(y)) \mid \alpha.\mathsf{Q}$$

$$\{[Q_1]\}_{x_1} = (X_1)c_1 \mid \mathsf{Q}_1 \text{ and } \{[Q_2]\}_{x_2} = (X_2)c_2 \mid \mathsf{Q}_2 \text{ imply } \{[Q_1+Q_2]\}_x =$$
$$(X_1 \cup X_2 \cup \{x_1\} \cup \{x_2\})c_1 \otimes c_2 \otimes (eq(x,x_1) \oplus eq(x,x_2)) \mid \mathsf{Q}_1 + \mathsf{Q}_2$$

$$\{[Q]\}_x = (X)c \mid \mathsf{Q} \text{ implies } \{[\Box.Q]\} = (X \cup \{x\})c \mid \Box\langle x\rangle.\mathsf{Q}$$

Table 5. Compilation of services

is the process obtained by compiling Q. All variables occurring in constraints are restricted in the resulting target service.

$$[\![\mathbf{0}]\!]_{\mathtt{nil}} = \mathbf{0}$$

$$[\![p]\!]_{\mathtt{cons}(x,\star)} = \mathsf{c}_x^p \mid \mathsf{p}_x^p$$

$$[\![P]\!]_\ell = (X)c \mid \mathsf{P} \text{ implies } [\![\overline{\boxtimes}.P]\!]_{\mathtt{cons}(x,\ell)} = (X)c \otimes eq(x,end) \mid \overline{\boxtimes}.\mathsf{P}$$

$$[\![P]\!]_{\mathtt{cons}(y,\ell)} = (X)c \mid \mathsf{P} \text{ implies } [\![\overline{\alpha}.P]\!]_{\mathtt{cons}(x,\ell)} = (X \cup \{y\})c \otimes eq(x,\alpha(y)) \mid \overline{\alpha}.\mathsf{P}$$

$$[\![P]\!]_{\mathtt{cons}(x,\ell)} = (X)c \mid \mathsf{P} \text{ implies}$$
$$[\![rec\,p.P]\!]_{\mathtt{cons}(x,\ell)} = rec_x\,\mathsf{c}^p.((X)c) \mid rec_x\mathsf{p}^p.((X)c[{}^1\!/_{\mathsf{c}_{y_i}^p} \mid i \in I] \mid \mathsf{P})$$
where y_i for $i \in I$ are all the variables occurring as indexes of c^p in c.

$$[\![P_1]\!]_{\ell_1} = (X_1)c_1 \mid \mathsf{P}_1 \text{ and } [\![P_2]\!]_{\ell_2} = (X_2)c_2 \mid \mathsf{P}_2 \text{ imply}$$
$$[\![P_1+P_2]\!]_\ell =$$
$$(X_1 \cup X_2 \cup \mathcal{X}(\ell_1) \cup \mathcal{X}(\ell_2)\})c_1 \otimes c_2 \otimes (EQ(\ell,\ell_1) \oplus EQ(\ell,\ell_2)) \mid$$
$$(\mathtt{check}\ EQ(\ell,\ell_1).\mathsf{P}_1, \mathtt{check}\ EQ(\ell,\ell_2).\mathsf{P}_2)$$
where $\ell = \mathcal{F}(\ell_1,\ell_2)$

$$[\![P]\!]_{\mathtt{cons}(x,\ell)} = (X)c \mid \mathsf{P} \text{ implies } [\![\overline{\Box}.P]\!]_\ell = (X)c \mid \overline{\Box}\langle x\rangle.\mathsf{P}$$

Table 6. Compilation of balanced clients

The compilation of clients (Table 6) is more complex, since client processes have nested calls, recursion and check expressions.

In order to deal with nested service calls we use a stack, represented as a list (*nesting list*), recording the variables which are the roots of the constraints in the encoding of the nested service calls which are still open and suspended (and then need to be closed); the head of this list contains the variable corresponding to the constraint root of the current call. We denote by `nil` the empty list, by ℓ an arbitrary list of variables or $\star$ and by $\mathtt{cons}(x,\ell)$ the list with head x and tail ℓ. The nesting list of $\mathbf{0}$ is `nil`. For an interaction $\overline{\alpha}.P$ the nesting list is obtained just replacing the fresh variable x to the variable y which

$$\mathcal{F}(\ell_1,\ell_2)=\begin{cases}\mathtt{cons}(x,\mathcal{F}(\ell_1',\ell_2')) & \text{if } \ell_i=\mathtt{cons}(x_i,\ell_i') \quad i=1,2\\ \mathtt{cons}(x,\mathcal{F}(\ell_1',\star)) & \text{if } \ell_1=\mathtt{cons}(x_1,\ell_1') \text{ and } \ell_2=\star\\ \mathtt{cons}(x,\mathcal{F}(\ell_2',\star)) & \text{if } \ell_2=\mathtt{cons}(x_2,\ell_2') \text{ and } \ell_1=\star\\ \mathtt{nil} & \text{if } \ell_1=\ell_2=\mathtt{nil} \text{ or } \ell_1=\mathtt{nil} \text{ and } \ell_2=\star \quad \text{where } x \text{ is fresh}\\ & \text{ or } \ell_2=\mathtt{nil} \text{ and } \ell_1=\star\\ \star & \text{if } \ell_1=\ell_2=\star\\ \text{undefined} & \text{otherwise.}\end{cases}$$

$$EQ(\ell_1,\ell_2)=\begin{cases}eq(x_1,x_2)\otimes EQ(\ell_1',\ell_2') & \text{if } \ell_i=\mathtt{cons}(x_i,\ell_i') \quad i=1,2\\ 1 & \text{if } \ell_1=\ell_2=\mathtt{nil} \text{ or } \ell_1=\star \text{ or } \ell_2=\star,\\ \text{undefined} & \text{otherwise.}\end{cases}$$

$$\begin{aligned}\mathcal{X}(\mathtt{nil}) &= \emptyset\\ \mathcal{X}(\star) &= \emptyset\\ \mathcal{X}(\mathtt{cons}(x,\ell)) &= \{x\}\cup\mathcal{X}(\ell)\end{aligned}$$

Table 7. Auxiliary functions for the compilation of client choices

is the head of the nesting list in the compilation of process P. The compilation of the end $\overline{\boxtimes}$ of a service call corresponds (being the compilation bottom-up) to the resumption of the interaction with a service after the end of the nested interaction. Then we push a new fresh variable on the nesting list corresponding to the last action of the nested call.

When compiling a recursion variable the list of nested service is not yet known (being the compilation one-step) and so is replaced by the placeholder $\star$. Going on in the compilation process, however, we check (via the function EQ defined in in Table 7) that the nesting lists of the suspended services in all branches of the recursive client are consistent (recall that no new service can be opened inside a recursive processes).

In the compilation of a choice we must assure that the suspended services call will be resumed safely in the different branches (which could require different choices also in the paired services). This check is performed using the function $\mathcal{F}$ (see Table 7) which creates a new fresh nesting list starting from the nesting lists in the compilation of the two branches. The application of $\mathcal{F}$ to the lists ℓ_1,ℓ_2 is defined only if either ℓ_1,ℓ_2 have the same length or the shorter list terminates with $\star$. When defined the value of $\mathcal{F}(\ell_1,\ell_2)$ is a list of n fresh variables, where n is the maximum of the lengths of ℓ_1,ℓ_2, terminating with $\star$ if both ℓ_1,ℓ_2 terminate with $\star$ and with $\mathtt{nil}$ otherwise. Note that in a recursive process no $\overline{\boxtimes}$ can occur along a path ending in a recursion variable.

The compilation of a service call $\overline{\square}$ uses the head variable of the nesting list to compile the call and continues with the tail of the nesting list going then one level up in the call nesting. Note the compilation of a balanced client has $\mathtt{nil}$ as final nesting list.

The process resulting from the compilation of a recursion variable is the parallel of a constraint variable and a process variable. This is so since in compiling recursive processes we need to generate both a recursive "global" constraint (taking into account all possible unfoldings of the process) and a recursive process including a "local" copy of its constraint (associated to each specific unfolding). The global recursive constraint characterises the process behaviour and is used in searching for a contract with the set

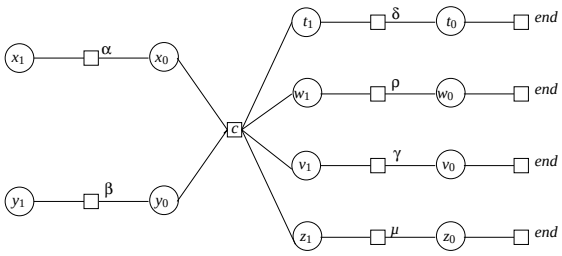

Fig. 1. Constraint store of Example 1

of considered services. The local copy of the constraints in each recursive call is used instead to keep track of the overall solution in the `check` branches. In the local copy of the constraint the occurrences of the recursion constraint variable (which are free) can be replaced by 1 since the consistency with the global solution is assured by the replacements of the process variables.

The compilation of a choice needs to check if one or both of the two branches agree with the interactions offered by services. This is accomplished by requiring that the list $\mathcal{F}(\ell_1,\ell_2)$ can be equated to ℓ_1 or to ℓ_2 considering that $\star$ can be equated to any list (see the definition of EQ in Table 7).

All fresh variables which are introduced by compilation are restricted in the resulting process, but for the variables which occur in the service calls. In order to restrict the variables of nesting lists in the compilation of choices we use the function $\mathcal{X}$ defined in Table 7. We show below two examples of compiled clients.

Example 1. The compilation of the source client $\overline{\square}.\overline{\alpha}.\overline{\square}\overline{\beta}.(\overline{\gamma}.\overline{\boxtimes}.\overline{\delta}.\overline{\boxtimes}.\mathbf{0} + \overline{\mu}.\overline{\boxtimes}.\overline{\rho}.\overline{\boxtimes}.\mathbf{0})$ is the following target client:

$$
\begin{array}{l}
(x_0,y_0,v_1,v_0,t_1,t_0,z_1,z_0,w_1,w_0)eq(x_1,\alpha(x_0))\otimes eq(y_1,\beta(y_0))\otimes \\
(eq(x_0,t_1)\otimes eq(y_0,v_1)\oplus eq(x_0,w_1)\otimes eq(y_0,z_1))\otimes eq(v_1,\gamma(v_0))\otimes eq(v_0,end)\otimes \\
eq(t_1,\delta(t_0))\otimes eq(t_0,end)\otimes eq(z_1,\mu(z_0))\otimes eq(z_0,end)\otimes eq(w_1,\rho(w_0))\otimes eq(w_0,end)\ \mid \\
\overline{\square}\langle x_1\rangle.\overline{\alpha}.\overline{\square}\langle y_1\rangle.\overline{\beta}.\texttt{check}\ eq(x_0,t_1)\otimes eq(y_0,y_1).\overline{\gamma}.\overline{\boxtimes}.\overline{\delta}.\overline{\boxtimes}.\mathbf{0}, \\
\qquad\qquad \texttt{check}\ eq(x_0,w_1)\otimes eq(y_0,z_1).\overline{\mu}.\overline{\boxtimes}.\overline{\rho}.\overline{\boxtimes}.\mathbf{0}
\end{array}
$$

which can be simplified to

$$
\begin{array}{l}
(x_1,y_1)eq(x_1,\alpha(x_0))\otimes eq(y_1,\beta(y_0))\otimes \\
(eq(x_0,\delta(end))\otimes eq(y_0,\gamma(end))\oplus eq(x_0,\rho(end))\otimes eq(y_0,\mu(end)))\ \mid \\
\overline{\square}\langle x_0\rangle.\overline{\alpha}.\overline{\square}\langle y_0\rangle.\overline{\beta}.\texttt{check}\ eq(x_0,\delta(end))\otimes eq(y_0,\gamma(end)).\overline{\gamma}.\overline{\boxtimes}.\overline{\delta}.\overline{\boxtimes}.\mathbf{0}, \\
\qquad\qquad \texttt{check}\ eq(x_0,\rho(end))\otimes eq(y_0,\mu(end)).\overline{\mu}.\overline{\boxtimes}.\overline{\rho}.\overline{\boxtimes}.\mathbf{0}
\end{array}
$$

In Figure 1 we give the graph representation of the constraint store of this client. Each node represents a variable and each constraint is modelled by a hyperedge connecting the variables involved in the constraint. By c we abbreviate the constraint $c = eq(x_0,t_1)\otimes eq(y_0,v_1)\oplus eq(x_0,w_1)\otimes eq(y_0,z_1)$. The remaining constraints are equalities: by placing a name on the right of an hyperedge we abbreviate the constraint that

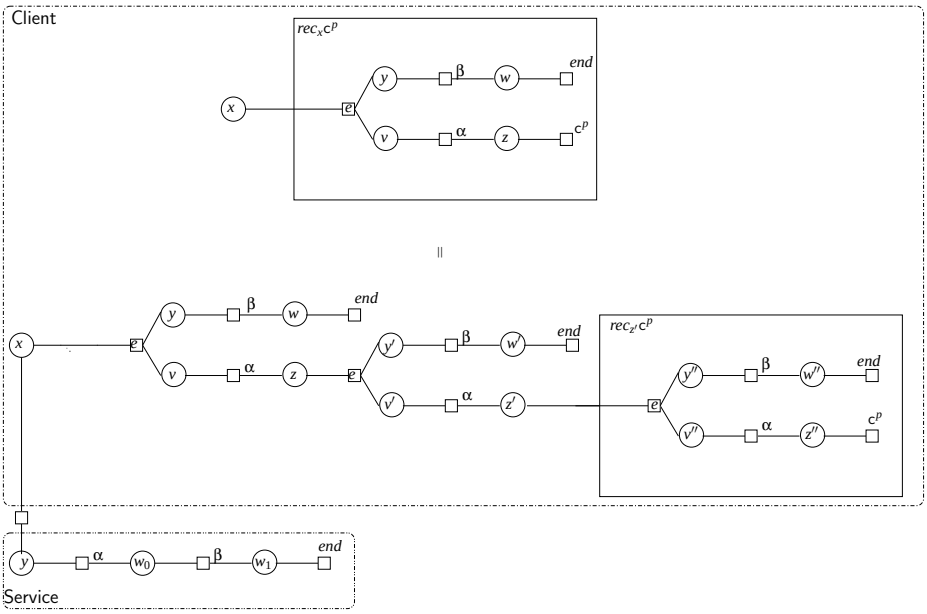

Fig. 2. Constraint store of Example 2

fuses the variable on the left to a function corresponding to that name, possibly applied to the variable on the right. For instance, name δ stands for the constraint $eq(t_1, \delta(t_0))$; similarly, the top-most occurrence of *end* denotes the constraint $eq(t_0, end)$.

Note that this client can safely interact, for instance, with a set of (source) services including $\{\Box.\alpha.\delta.\boxtimes,\ \Box.\beta.(\gamma.\boxtimes + \nu.\boxtimes)\}$.

Example 2. Compiling the recursive process $\overline{\Box}.rec\,p.(\overline{\alpha}.p + \overline{\beta}.\overline{\boxtimes}.\mathbf{0})$ we get, after some simplifications:

$$\begin{array}{l} rec_x \mathsf{c}^p.(z)(eq(x,\alpha(z)) \oplus eq(x,\beta(end))) \otimes \mathsf{c}^p_z \mid \\ \overline{\Box}.\langle x\rangle.rec_x \mathsf{p}^p.(z)(eq(x,\alpha(z)) \oplus eq(x,\beta(end))) \\ \qquad (\mathtt{check}\ eq(x,\alpha(z)).\overline{\alpha}.\mathsf{p}^p_z, \mathtt{check}\ eq(x,\beta(end)).\overline{\beta}.\mathbf{0}) \end{array}$$

The recursive constraint after two unfoldings and some simplifications becomes:

$$\begin{array}{c} c = (z,z')(eq(x,\alpha(z)) \oplus eq(x,\beta(end))) \otimes (eq(z,\alpha(z')) \oplus eq(z,\beta(end))) \otimes \\ rec_{z'} \mathsf{c}^p.(z'')((eq(z',\alpha(z'')) \oplus eq(z',\beta(end))) \otimes \mathsf{c}^p_{z''}) \end{array}$$

and the corresponding process:

$$\begin{array}{c} \overline{\Box}\langle x\rangle(z)(eq(x,\alpha(z)) \oplus eq(x,\beta(end))) \mid \\ (\mathtt{check}\ eq(x,\alpha(z)).\overline{\alpha}. \\ (z')(eq(z,\alpha(z')) \oplus eq(z,\beta(end))) \mid \\ (\mathtt{check}\ eq(z,\alpha(z')).\overline{\alpha}.rec_{z''} \mathsf{p}^p ....., \mathtt{check}\ eq(z,\beta(end)).\overline{\beta}.\mathbf{0}), \\ \mathtt{check}\ eq(x,\beta(end)).\overline{\beta}.\mathbf{0}) \end{array}$$

If the service $(y)d \mid \Box\langle y\rangle.\alpha.\beta.\boxtimes$, where

$$d = (w_0, w_1)\ eq(y, \alpha(w_0)) \otimes eq(w_0, \beta(w_1)) \otimes eq(w_1, end)$$

is available, then the constraint $\exists y.c \otimes d \otimes eq(x,y)$ is equal to $eq(x, \alpha(\beta(end)))$.

This solution is determined only by the (proper) unfolding of the recursive constraint of the client and the constraint of the service. Once the solution is determined the constraint of each unfolding is used only to assure the correct choices in the `check` branches. These constraints are propagated via the unfolding of the recursive process and so no unfolding of the outermost recursive constraint is needed in the process reduction.

Figure 2 depicts the constraint store representing a commit between the recursive client and the service. We represent recursive constraints as boxed constraints with a connection to the variable that is the index of the recursive operator. By e we abbreviate the constraints of the shape $e = eq(u_1,u_2) \oplus eq(u_1,u_3)$, where u_1 is the variable on the left node and u_2, u_3 are the variables on the right nodes.

It is easy to verify that our compilation is successful for all source clients and processes, but for the case of unbalanced clients.

Theorem 1. *Each balanced client and each service can be compiled.*

More interesting (and more complex to prove) is the fact that given a compiled client and a set of available services $\mathbb{S}_t$, the labelled transition system in Table 4 either is empty or always terminates. We start with some definitions.

Definition 1. *1. A source system V is* reachable *if $T \stackrel{\sigma}{\Longrightarrow} V$ for some σ and some source client T.*

2. A target system V is reachable *if $\mathtt{start}(\mathsf{T}, \mathbb{S}_t) \stackrel{\sigma}{\Longrightarrow} \mathsf{V}$ for some σ and some T obtained by compiling a source client.*

Note that each source system V which is not a client or satisfied has the shape $V = [\ldots[[P \mid Q_0]|Q_1]\ldots Q_n]$ for some $n \geq 0$. Similarly each target system V which is obtained by reducing the compilation of a client and it is not satisfied has the shape $\mathsf{V} = (X_n)[(X_{n-1})\ldots[(X_0)[c \mid \mathsf{P} \mid \mathsf{Q}_0]|\mathsf{Q}_1]\ldots\mathsf{Q}_n]$. This justifies the mappings defined in Table 8.

Lemma 1. *If V is reachable and not satisfied, $\mathtt{constraint}(\mathsf{V}) = c$, $\mathtt{client}(\mathsf{V}) = \mathsf{P}$, $\mathtt{depth}(\mathsf{V}) = n$, $\mathtt{service}(i, \mathsf{V}) = \mathsf{Q}_i$ for $1 \leq i \leq n$, then there are P, Q_i, x_i, y_i such that $\mathsf{P} = [\![P]\!]_{\mathtt{cons}(x_0,\ldots\mathtt{cons}(x_n,\mathtt{nil}))}$, $\mathsf{Q} = \{[Q_i]\}_{y_i}$, and $c \leq x_i = y_i$ for $1 \leq i \leq n$.*

Proof. By induction on reductions.

Lemma 2. *If V is reachable and not satisfied, $\mathtt{constraint}(\mathsf{V}) = c$, $\mathtt{client}(\mathsf{V}) = \mathsf{P}$ and $\mathtt{service}(0, \mathsf{V}) = \mathsf{Q}$, then*

- *$\mathsf{P} = \overline{\Box}\langle x\rangle.\mathsf{P}'$ implies that there is $(X)\,d \mid \Box\langle x\rangle.\mathsf{Q} \in \mathbb{S}_t$ such that $c \otimes d \neq 0$;*
- *$\mathsf{P} = \overline{\alpha}.\mathsf{P}'$ implies $\mathsf{Q} \stackrel{\alpha}{\longrightarrow} \mathsf{Q}'$ for some Q';*

$$
\begin{array}{lll}
\texttt{constraint}((X)c \mid \mathbf{0}) & = & c \\
\texttt{constraint}(c \mid \mathsf{P} \mid \mathsf{Q}) & = & c \\
\texttt{constraint}((X)[\mathsf{U}] \mid \mathsf{Q}) & = & \texttt{constraint}(\mathsf{U}) \\
\texttt{constraint}((X)[\mathsf{U}]) & = & \texttt{constraint}(\mathsf{U})
\end{array}
$$

$$
\begin{array}{lll}
\texttt{client}(c \mid \mathsf{P} \mid \mathsf{Q}) & = & \mathsf{P} \\
\texttt{client}((X)[\mathsf{U}] \mid \mathsf{Q}) & = & \texttt{client}(\mathsf{U}) \\
\texttt{client}((X)[\mathsf{U}]) & = & \texttt{client}(\mathsf{U})
\end{array}
$$

$$
\begin{array}{lll}
\texttt{depth}(c \mid \mathsf{P} \mid \mathsf{Q}) & = & 0 \\
\texttt{depth}((X)[\mathsf{U}] \mid \mathsf{Q}) & = & \texttt{depth}(\mathsf{U}) + 1 \\
\texttt{depth}((X)[\mathsf{U}]) & = & \texttt{depth}(\mathsf{U})
\end{array}
$$

$$
\begin{array}{lll}
\texttt{service}(0, c \mid \mathsf{P} \mid \mathsf{Q}) & = & \mathsf{Q} \\
\texttt{service}(i, (X)[\mathsf{U}] \mid \mathsf{Q}) & = & \begin{cases} \mathsf{Q} & \text{if } i = \texttt{depth}((X)[\mathsf{U}]), \\ \texttt{service}(i, \mathsf{U}) & \text{otherwise.} \end{cases} \\
\texttt{service}(i, (X)[\mathsf{U}]) & = & \texttt{service}(i, \mathsf{U})
\end{array}
$$

Table 8. Auxiliary mappings

- $\mathsf{P} = \overline{\boxtimes}.\mathsf{P}'$ *implies* $\mathsf{Q} \xrightarrow{\boxtimes}$*;*
- $\mathsf{P} = \texttt{check}\ c_1.\mathsf{P}_1, \texttt{check}\ c_2.\mathsf{P}_2$ *implies that* $c \otimes c_i \neq 0$ *for some* i *and:*
  - *either* $\mathsf{P}_i \xrightarrow{\lceil} \mathsf{V}'$ *for some* V'*;*
  - *or* $\mathsf{P}_i \xrightarrow{\overline{\alpha}} \mathsf{P}'$ *and* $\mathsf{Q} \xrightarrow{\alpha} \mathsf{Q}'$ *for some* $\alpha, \mathsf{P}', \mathsf{Q}'$*;*
  - *or* $\mathsf{P}_i \xrightarrow{\overline{\boxtimes}} \mathsf{P}'$ *and* $\mathsf{Q} \downarrow_{\boxtimes}$ *for some* P'.

Proof. By Lemma 1 we have $\mathsf{P} = [\![P]\!]_{\texttt{cons}(x,\ell)}$, $\mathsf{Q} = \{[Q]\}_y$ and $c \leq x = y$ for some P, x, ℓ, Q, y. The proof follows by looking at the compilation.

Theorem 2. *If* T *is obtained by compiling a source client and* $\texttt{start}(\mathsf{T}, \mathbb{S}_t) \stackrel{\sigma}{\Longrightarrow} \mathsf{V}$*, then* $\mathsf{V} \Downarrow^{must}$.

Proof. First of all notice that there are no infinite computations, since services are not recursive, and the constraint of a client can be satisfied only unfolding its recursive calls a finite number of times. Therefore we only need to show that V is not stuck. The proof is by cases on $\texttt{client}(\mathsf{V})$ using Lemmas 1 and 2.

Comparing the reduction rules in Tables 2 and 4 it is easy to verify that the reductions of a target client obtained by compiling a source client correspond to reductions of the source client itself. We denote by $|\mathsf{V}|$ the source system obtained from the target system V by erasing all constraints and restrictions and by replacing checks by sums, see Table 9, where W denotes a compiled client or a compiled service.

Theorem 3. *(Soundness) If* $\mathbb{S}_t$ *is obtained by compiling the source services of* $\mathbb{S}_s$ *and* $\texttt{start}([\![T]\!], \mathbb{S}_t) \stackrel{\sigma}{\Longrightarrow} \mathsf{V}$*, then* $T \stackrel{\sigma}{\Longrightarrow} |\mathsf{V}|$.

The aim of the compilation is to avoid deadlocks, but also to preserve all successful interactions. This is the content of the following theorem, whose proof requires a precise analysis of the relations between the reduct of a target system and the compilation of the reduct of a source system.

$$
\begin{array}{ll}
|\overline{\Box}\langle x\rangle.\mathsf{P}| & = \overline{\Box}.|\mathsf{P}| \\
|\lambda.\mathsf{W}| & = \lambda.|\mathsf{W}| \\
|(\texttt{check } c_1.\mathsf{P}_1, \texttt{check } c_2.\mathsf{P}_2)| & = |\mathsf{P}_1| + |\mathsf{P}_2| \\
|\ \mathsf{p}^p_x| & = p \\
|rec_x \mathsf{p}^p.\mathsf{R}| & = rec\, p.|\mathsf{R}| \\
|\overline{\boxtimes}.\mathsf{P}| & = \overline{\boxtimes}.|\mathsf{P}| \\
|\mathbf{0}| & = \mathbf{0} \\
|\boxtimes| & = \boxtimes \\
|(X)c \mid \mathsf{P}| & = |\mathsf{P}| \\
|\mathsf{Q}_1 + \mathsf{Q}_1| & = |\mathsf{Q}_1| + |\mathsf{Q}_2| \\
|\Box\langle x\rangle.\mathsf{Q}| & = \Box.|\mathsf{Q}| \\
|c \mid \mathsf{P} \mid \mathsf{Q}| & = |\mathsf{P}| \mid |\mathsf{Q}| \\
|(X)[\mathsf{U}] \mid \mathsf{Q}| & = |(X)[\mathsf{U}]| \mid |\mathsf{Q}| \\
|(X)[\mathsf{U}]| & = [|\mathsf{U}|]
\end{array}
$$

Table 9. The "forgetting" map $|\ |$

Theorem 4. *(Completeness) If $\mathbb{S}_t$ is obtained by compiling the source services of $\mathbb{S}_s$ and $T \stackrel{\sigma}{\Longrightarrow} V$ and $V \Downarrow^{may}$, then there is V such that $\texttt{start}([\![T]\!], \mathbb{S}_t) \stackrel{\sigma}{\Longrightarrow} \mathsf{V}$ and $|\mathsf{V}| = V$.*

The remaining of this section is devoted to the completeness proof. Table 10 defines a translation from source systems to target systems.

$$
\begin{array}{l}
\{\{P \mid Q\}\}_\ell \ = \ (X_1 \cup X_2)\ c_1 \otimes c_2 \otimes eq(x,y) \otimes d_{I(\mathsf{P})} \mid \mathsf{P} \mid \mathsf{Q} \\
\quad \text{where } [\![P]\!]_{\texttt{cons}(x,\ell)} = (X_1)c_1 \mid \mathsf{P} \text{ and } \{[Q]\}_y = (X_2)c_2 \mid \mathsf{Q} \\
\{\{[U] \mid Q\}\}_\ell \ = \ (X_1 \cup X_2)\ c_1 \otimes c_2 \otimes eq(x,y) \mid [\mathsf{U}] \mid \mathsf{Q} \\
\quad \text{where } \{\{U\}\}_{\texttt{cons}(x,\ell)} = (X_1)c_1 \mid \mathsf{U} \text{ and } \{[Q]\}_y = (X_2)c_2 \mid \mathsf{Q} \\
\{\{T\}\}_{\texttt{nil}} = \texttt{start}(\mathsf{T}, \mathbb{S}_t) \text{ where } \mathsf{T} = [\![T]\!]_{\texttt{nil}} \\
\{\{0\}\}_{\texttt{nil}} = 1 \mid 0
\end{array}
$$

Table 10. From source systems to target systems

Lemma 3. *If V is reachable, $\mathsf{V} \stackrel{\psi}{\longrightarrow} \mathsf{V}'$ and $\texttt{constraint}(\mathsf{V}) = c$, then $\texttt{constraint}(\mathsf{V}') = \exists Z.c \otimes c'$ for some Z, c' where $c' \neq 0$.*

Proof. By cases on ψ.

Lemma 4. *Let $V \Downarrow^{may}$. Then $\texttt{constraint}(\{\{V\}\}_{\texttt{nil}}) \neq 0$.*

Proof. If $V \Downarrow^{may}$ then $\exists\sigma$ such that $V \stackrel{\sigma}{\Longrightarrow} \mathbf{0}$. The proof is by induction on σ using Lemma 3. Note that $\texttt{constraint}(1 \mid \mathbf{0}) = 1$.

We define the equivalence $\approx$ between target systems as the minimal congruence generate by the following axioms:

$$[(X)[\mathsf{U}] \mid \mathsf{Q}] \approx (X)[[\mathsf{U}] \mid \mathsf{Q}]$$
$$(X)[\mathsf{U}] \approx (X \cup X')[\mathsf{U}] \text{ if } X' \cap FV(\mathsf{U}) = \emptyset$$
$$c \mid \mathsf{P} \approx \exists Z.c \mid \mathsf{P} \text{ if } Z \cap FV(\mathsf{P}) = \emptyset$$

Note that $\mathsf{V} \approx \mathsf{V}'$ implies $\mathtt{constraint}(\mathsf{V}) \neq 0$ if and only if $\mathtt{constraint}(\mathsf{V}') \neq 0$.

Lemma 5. *1.* $\mathsf{V} \approx \{\{V\}\}_{\mathtt{nil}}$ *implies* $|\mathsf{V}| = V$.
2. $\mathsf{V} \approx \mathsf{V}'$ *and* $\mathsf{V} \xrightarrow{\psi} \mathsf{V}_1$ *imply* $\mathsf{V}' \xrightarrow{\psi} \mathsf{V}'_1$ *and* $\mathsf{V}_1 \approx \mathsf{V}'_1$.

Lemma 6. *If* $V \xrightarrow{\psi} V'$, $\mathtt{constraint}(\{\{V'\}\}_{\mathtt{nil}}) \neq 0$ *and* $\mathtt{constraint}(\{\{V\}\}_{\mathtt{nil}}) \neq 0$, *then* $\{\{V\}\}_{\mathtt{nil}} \xrightarrow{\psi} \mathsf{V}'$ *and* $\{\{V'\}\}_{\mathtt{nil}} \approx \mathsf{V}'$.

Proof. By cases on ψ using Lemma 2.

We prove a slightly different version of the completeness theorem, which immediately implies the original version by Lemma 5(1).

Theorem 5. *(Completeness) If* $\mathbb{S}_t$ *is obtained by compiling the source services of* $\mathbb{S}_s$ *and* $T \xRightarrow{\sigma} V$ *and* $V \Downarrow^{may}$, *then there is* V *such that* $\mathtt{start}([\![T]\!], \mathbb{S}_t) \xRightarrow{\sigma} \mathsf{V}$ *and* $\mathsf{V} \approx \{\{V\}\}_{\mathtt{nil}}$.

Proof. By induction on σ. The base step is straightforward.
As for the induction step assume $T \xRightarrow{\sigma} V \xrightarrow{\psi} V'$ and $V' \Downarrow^{may}$. Then also $V \Downarrow^{may}$. By induction hypothesis $\mathtt{start}([\![T]\!], \mathbb{S}_t) \xRightarrow{\sigma} \mathsf{V}$ where $\mathsf{V} \approx \{\{V\}\}_{\mathtt{nil}}$ is reachable. By Lemma 4 $\mathtt{constraint}(\{\{V\}\}_{\mathtt{nil}}) \neq 0$ and $\mathtt{constraint}(\{\{V'\}\}_{\mathtt{nil}}) \neq 0$. Therefore we can apply Lemma 6 to get $\{\{V\}\}_{\mathtt{nil}} \xrightarrow{\psi} \mathsf{V}''$ where $\mathsf{V}'' \approx \{\{V'\}\}_{\mathtt{nil}}$. Then we conclude using Lemma 5(2).

5 Optimised semantics

In the previous Section we have shown that the semantics of the target calculus ensures stuck-freedom provided that for each service call of a client there is a service which is able to complete an interaction with the client successfully. Nevertheless, during a client-service negotiation, it is desirable to have a mean to select the services that offer more choices to the client (considering all choices equally satisfactory), among all services which can successfully complete an interaction. This property holds for the target LTS only if there is a single service. By contrast, in the general case in which there is more than a 'complying' service available, the target LTS gives no guarantee on this respect. For instance, let $T = \overline{\square}.(\overline{\alpha}.\overline{\boxtimes}.\mathbf{0} + \overline{\beta}.\overline{\boxtimes}.\mathbf{0})$ be a client and $S_1 = \square.(\alpha.\boxtimes + \beta.\boxtimes)$ and $S_2 = \square.(\alpha.\boxtimes + \gamma.\boxtimes)$ be two services. According to the target semantics both S_1 and S_2 can be selected, as none of them would lead to a deadlock. Nevertheless, S_1 is somehow preferable as it additionally allows T to exhibit action $\overline{\beta}$. Clearly for the client $\overline{\square}.(\overline{\alpha}.\overline{\boxtimes}.\mathbf{0} + \overline{\gamma}.\overline{\boxtimes}.\mathbf{0})$ the service S_2 is better.

We define an *optimised* Labelled Transition System that is obtained from the LTS given in Table 4 by replacing rule $(t\text{-}call)$ by the following rule:

$$\frac{(X)\,d \mid \Box\langle x\rangle.\mathsf{Q} \in \mathbb{S}_t \quad \begin{array}{l} c\otimes d \neq 0 \text{ and } \nexists (X')\,d' \mid \Box\langle x\rangle.\mathsf{Q}' \in \mathbb{S}_t \text{ s.t.} \\ c\otimes d' \neq 0 \text{ and } c\otimes d < c\otimes d' \end{array}}{c \mid \overline{\Box}\langle x\rangle.\mathsf{P} \xrightarrow{\lceil} (X)[c\otimes d \mid \mathsf{P} \mid \mathsf{Q}]}\ (t\text{-}call')$$

We denote by $\stackrel{\sigma}{\Longmapsto}$ reductions in the optimised LTS.

Rule $(t\text{-}call')$ ensures that a service S is selected if it is *one of the best* possible services, namely such that there is no other service which offers the *same* successful interaction paths of S and some more. Clearly in general we can get incomparable constraints, like for example for the client $T' = \overline{\Box}.(\overline{\alpha}.(\overline{\mu}\overline{\boxtimes}.\mathbf{0} + \overline{\rho}\overline{\boxtimes}.\mathbf{0}) + \overline{\beta}.\overline{\boxtimes}.\mathbf{0} + \overline{\gamma}.\overline{\boxtimes}.\mathbf{0})$ and the services $S_3 = \Box.(\alpha.\mu.\boxtimes + \beta.\boxtimes + \gamma.\boxtimes)$ and $S_4 = \Box.\alpha.(\mu\boxtimes + \rho\boxtimes)$. Note that the client T' has 3 successful paths choosing service S_3 and only 2 successful paths choosing service S_4, but these paths are incomparable. In such cases rule $(t\text{-}call')$ chooses in a non deterministic way.

We prove that the new target semantics is *optimal*, in the sense that by choosing one of best services at each service call we obtain a set of services that guarantee the same interactions with more choices to a client. For simplicity we consider only *unambiguous* source processes, i.e. processes whose sums start with two different actions (modulo commutativity and associativity of sums).

Theorem 6. *If* T *is obtained by compiling a source client and* $\mathtt{start}(\mathsf{T},\mathbb{S}_t) \stackrel{\sigma}{\Longmapsto} \mathsf{V}$, *then there is no* V' *such that* $\mathtt{start}(\mathsf{T},\mathbb{S}_t) \stackrel{\sigma}{\Longrightarrow} \mathsf{V}'$ *and* $\mathtt{constraint}(\mathsf{V}) < \mathtt{constraint}(\mathsf{V}')$.

Proof. Remark that the condition of non ambiguity assures that σ uniquely determines the choices performed by the applications of rule $(t\text{-}check)$. So the only non determinism comes from the applications of rule $(t\text{-}call)$ and $(t\text{-}call')$, which are also the only rules which change the constraint. The result then follows from the observation that rule $(t\text{-}call')$ always maximises the constraint of the resulting system.

6 Related Papers

Our notion of stuck-freedom is taken from [15]: there are no messages waiting forever to be send or sent messages which are never received. This property is crucial in communication centred programming.

The most common calculi used to model communicating processes are session types and behavioural contracts.

Sessions and session types (first introduced in [16]) are built on π-calculus: the key idea is that channels can be used to send and receive messages of different types following a fixed communication protocol. We refer to [9] and [17] for overviews.

Behavioural contracts are CCS-like processes which describe the global communications between clients and services. Many recent papers focus on the compatibility between clients and services and the safe replacements of services. The necessary control on communications is achieved by explicit interfaces [11], filtering [8], orchestration [14]. A companion research line develops choreographies for service composition [13].

To the best of our knowledge, in a client-server context the idea of using constraints for negotiating which interactions to choose, hence avoiding deadlocks, is novel.

Named constraint semirings have beed originally proposed as the underlying structure of the cc-pi calculus [5], a process calculus for modelling agreements on non-functional parameters in a service oriented scenario. The target calculus we have introduced in this paper is close in spirit to the cc-pi calculus, except for the fact that cc-pi adopts a communication mechanism *à la* pi-calculus while the primitives of our target calculus are meant to model two-party sessions. [6] presents a variant of the cc-pi calculus in which the non-deterministic choice is replaced by an operation that allows selecting an action if the corresponding constraint has a priority over the constraints of the alternative branches. Though priorities are assigned following different criteria, the optimised semantics we have proposed is inspired by the prioritised cc-pi calculus.

Named constraint semirings have been defined as an extension of constraint semirings with a notion of relevant names. C-semirings [2] allow defining soft constraints, namely constraints which do not return only true or false, but more informative values instead (e.g., degree of preference, cost), thus extending paradigms like Constraint Logic Programming or Concurrent Constraint Programming. In [4] a version of the Soft CCP has been used for specifying SLA negotiations, basically with the same goal of the cc-pi calculus.

7 Conclusion and Future Work

In the paper we have augmented a client-service calculus with suitable constraints. A run time combination (multiplication in the simple cases) of client and service constraints guarantees that all and only the stuck-free interactions are possible. The constraints are exactly those of logic programming, and in fact a concrete representation of our constraints with logic programs and logic program combinations is straightforward. This property is comfortable from a theoretical point of view, since logic programs are well understood, but it is now appropriate to ask if it might be useful also from a practical point of view. We did not study the issue, which is outside the scope of this paper, but we can say that asking for satisfaction of the combined client-service constraint would be perfectly possible in any logic programming implementation, which would return an example of stuck-free interaction. Efficiency might depend on the exact way in which clauses are listed and parallel goals expanded. However it might be possible to devise an efficient algorithm (e.g. factorizing constraints and matching them top down breadth first) and to build a metainterpreter implementing the algorithm.

In the future, we plan to generalise the current target calculus by exploiting the formalism of Soft Constraint Logic Programming by Bistarelli, Montanari and Rossi [3]. In that paper, the ground semantics of a logic program (a goal and set of clauses) is not a set of ground assignments of the free variables of its goal, but rather a function from ground assignments to values of (another) constraint semiring. These values could give a measure of how acceptable the assignments are. Such functions, computed pointwise, form again a constraint semiring, and thus the formal treatment turns out simple and elegant. In particular the three semantics of logic programming (operational, denotational and model theoretical) can be defined also for soft constraint logic programming and

proved equivalent. Specifically, in the context of the present paper a particular client-service computation would not be only possible or impossible, but it could be assigned an acceptance weight, which might itself be structured by measuring the quality of service obtained in the interaction. For instance these weights could be taken into account in the reduction rules $(t\text{-}call)$ and $(t\text{-}choice)$ in order to allow only executions which maximise client's satisfaction.

In a different direction we will extend both source and target calculi with internal choices in order to model interactions in which one participant takes one branch independently from what is offered by the other participant. Clearly to avoid deadlocks in presence of internal choices we need to rethink the compilation of clients and services.

Lastly we are interested in considering how to apply our approach to model interactions of more than two participants, taking inspiration from [10] and [7].

References

1. S. Bistarelli and F. Gadducci. Enhancing constraints manipulation in semiring-based formalisms. In *ECAI'06*, volume 141 of *Frontiers in Artif. Intel. and Applic.*, pages 63–67. IOS Press, 2006.
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. Semiring-based constraint logic programming: syntax and semantics. *ACM Transactions on Programming Languages and Systems*, 23(1):1–29, 2001.
4. S. Bistarelli and F. Santini. A nonmonotonic soft concurrent constraint language for SLA negotiation. In *CILC'08*, volume 236 of *ENTCS*, pages 147–162. Elsevier, 2009.
5. M. G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In *ESOP'07*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.
6. M. G. Buscemi and U. Montanari. Qos negotiation in service composition. *Journal of Logic and Algebraic Programming*, 80(1):13–24, 2011.
7. L. Caires and H. T. Vieira. Conversation types. *Theoretical Computer Science*, 411(51-52):4399–4440, 11 2010.
8. G. Castagna, N. Gesbert, and L. Padovani. A Theory of Contracts for Web Services. *ACM Transactions on Programming Languages and Systems*, 31, 2009. article n.19, pages 51.
9. M. Dezani-Ciancaglini and U. de' Liguoro. Sessions and session types: an overview. In *WSFM'09*, volume 6194 of *LNCS*, pages 1–28. Springer, 2010.
10. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
11. C. Laneve and L. Padovani. The must preorder revisited: An algebraic theory for web services contracts. In *CONCUR'07*, volume 4703 of *LNCS*, pages 212–225. Springer, 2007.
12. J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
13. M.Bravetti and G. Zavattaro. A theory of contracts for strong service compliance. *Mathematical Structures in Computer Science*, 19:601–638, 2009.
14. L. Padovani. Contract-directed synthesis of simple orchestrators. In *CONCUR'08*, volume 5201 of *LNCS*, pages 131–146. Springer, 2008.
15. S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *CAV'02*, volume 2402 of *LNCS*, pages 166–179. Springer, 2002.
16. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.
17. V. T. Vasconcelos. Sessions, from types to programming languages. *EATCS Bulletin*, 2011. to appear.