Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 200 (2008) 133–154
www.elsevier.com/locate/entcs

Service Discovery and Negotiation With COWS [1]

Alessandro Lapadula[2] Rosario Pugliese[3] Francesco Tiezzi[4]

Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
50134 Firenze, Italy

Abstract

To provide formal foundations to current (web) services technologies, we put forward using COWS, a process calculus for specifying, combining and analysing services, as a uniform formalism for modelling all the relevant phases of the life cycle of service-oriented applications, such as publication, discovery, negotiation, deployment and execution. In this paper, we show that constraints and operations on them can be smoothly incorporated in COWS, and propose a disciplined way to model multisets of constraints and to manipulate them through appropriate interaction protocols. Therefore, we demonstrate that also QoS requirement specifications and SLA achievements, and the phases of dynamic service discovery and negotiation can be comfortably modelled in COWS. We illustrate our approach through a scenario for a service-based web hosting provider.

Keywords: Service discovery, negotiation, service level agreement, service orchestration, process calculi.

1 Introduction

In recent years, the increasing success of e-business, e-learning, e-government, and other similar emerging models, has led the World Wide Web, initially thought of as a system for human use, to evolve towards a *service-oriented architecture* (SOA) supporting automated use. SOAs advocate the use of 'services', to be understood as autonomous, platform-independent, computational entities that can be described, published, discovered, and assembled, as the basic blocks for building applications. In an SOA, services can play essentially three different roles: the provider, the requestor and the registry. Providers offer functionalities and publish machine-readable service descriptions on registries to enable automated discover and invocation by requestors. In addition to the function that the service performs, service

[1] This work has been supported by the EU project SENSORIA, IST-2 005-016004.
[2] Email: lapadula@dsi.unifi.it
[3] Email: pugliese@dsi.unifi.it
[4] Email: tiezzi@dsi.unifi.it

doi:10.1016/j.entcs.2008.04.097

descriptions should also include non-functional properties, such as e.g., response time, availability, reliability, security, and performance, that jointly represent the *quality of the service* (QoS). Some of these properties could depend on the current run-time configuration of the system (e.g. the maximum allowed bandwidth might depend on the actual load of the server), thus a *dynamic discovery* process is often needed to find a provider that meets the requestors' requirements. Moreover, since services are often developed and run by different organizations, a key issue of the discovery process is to define a flexible *negotiation* mechanism that allows two or more parties to reach a joint agreement about cost and quality of a service, *prior to service execution*. This mechanism ranges from simple forms where a two-phase negotiation is sufficient (one of the two parties exposes a contract template that the other party can fill in with values in a given range) to more sophisticated forms where the parties use complex strategies and interact repeatedly. For example, if the involved parties fail to reach an agreement, their strategies can weaken the requirements and retry, or just give up the negotiation.

The outcome of the negotiation phase is a *Service Level Agreement* (SLA), i.e. a contract among the involved parties (service requestor and provider and, possibly, some third parties) that sets out both type and bounds on various performance metrics of the service to be provided, and the remedial actions to be performed if these are not met. For example, an SLA among a Web hosting provider and its customers may specify the annual cost of the service and the guaranteed bandwidth that will be provided, and the penalties to be imposed if the service fails to fulfill the guaranteed bandwidth. After the agreement has been achieved, trustworthy measurement services can possibly be used by each party to dynamically monitor that the contract is respected by the other parties.

A successful instantiation of SOA are the so-called *web services*, namely sets of operations that can be invoked through the Web via XML messages complying with given standard formats. The expansion of web services has caused the development of several new languages and technologies, among which we mention those for supporting the phases of discovery, negotiation, agreement and monitoring, like e.g. WSLA [31,23] and WS-Agreement [1], that permit specifying and managing SLAs, WS-Negotiation [22], that permits implementing automated negotiation, and [41,40], that exploit the ontology languages DAML-S and OWL-S to enable semantic matching of service capabilities.

To provide formal foundations to current (web) services technologies, in [28] we have introduced COWS as a formalism for specifying and orchestrating services while modelling their dynamic behaviour. COWS, in fact, falls within a main line of research (see e.g. [9,10,24,27,21,7,11,12,25]) that aims at developing process calculi capable of capturing the basic aspects of service-oriented systems and, possibly, of supporting the analysis of qualitative and quantitative properties of services. While the proposals in the literature address one specific aspect or another of currently available SOA technologies, in this paper we demonstrate that COWS instead can model all the phases of the life cycle of service-oriented applications, such as publication, dynamic discovery, negotiation, deployment and execution. We are not

affirming that whoever programs service-oriented applications should use COWS as the sole language. First of all, forcing to use only one language would be unrealistic and in neat contrast with the 'open-endedness' of the SOA paradigm. Moreover, COWS is a lower level modelling language rather than a full-fledged programming language. We are instead putting forward that COWS can be a common and convenient basis to enable analysis of service-oriented applications by means of translation from higher level languages. Indeed, it is widely recognized (see e.g. [32,42]) that a major benefit of using process calculi for modelling SOA systems and applications is that they enjoy a rich repertoire of elegant meta-theories, proof techniques and analytical tools that can be likely tailored to the needs of service-oriented applications. Therefore, the type system introduced in [29] to check data confidentiality properties, the stochastic extension defined in [37] to enable quantitative reasoning on service behaviours, and the logic and model checker presented in [16] to express and check functional properties of service behaviours, are certainly an important added value of using COWS for modelling services.

Technically, we exploit the fact that COWS language definition abstracts from a few sets of objects (e.g., the set of expressions) and appropriately specialize these parameters of the language so that services can specify and conclude SLAs. We follow the approach put forward in cc-pi [9], a language that combines basic features of name-passing calculi with concurrent constraint programming [38]. Specifically, we show that constraints and operations on them can be smoothly incorporated in COWS, and propose a disciplined way to model multisets of constraints and manipulate them through appropriate interaction protocols. This way, SLA requirements are expressed as constraints that can be dynamically generated and composed, and that can be used by the involved parties both for service publication and discovery (on the Web), and for the SLA negotiation process. Consistency of the set of constraints resulting from negotiation means that the agreement has been reached.

The rest of the paper is organized as follows. Section 2 presents the syntax of COWS and its informal operational semantics. Section 3 shows how COWS can be used for concurrent constraint programming. Section 4 describes some simple communication protocols that allow two parties to generate constraints through synchronization. Section 5 presents the specification of a web hosting scenario, that is, in a simplified form, one of the typical SOA scenarios where SLA among organizations are largely employed. Section 6 introduces some variants of the concurrent constraint programming constructs presented in Section 3. Finally, Section 7 concludes the paper with a few observations.

2 A Process Calculus for Service-Oriented Systems

COWS (*Calculus for Orchestration of Web Services*, [28]) is a recently defined process calculus for specifying, combining and analysing services, while modelling their dynamic behaviour. The design of the calculus has been influenced by the web services orchestration language WS-BPEL [36] and by existing process calculi. As a result, COWS integrates such features as e.g. not binding receive activities, asyn-

s	::=	$\mathbf{kill}(k) \mid u \bullet u'!\bar{e}$		(kill, invoke)
	$\mid$	$\sum_{i=0}^{l} p_i \bullet o_i?\bar{w}_i.s_i \mid s \mid s$		(receive-guarded choice, parallel)
	$\mid$	$\{\!\lvert s \rvert\!\} \mid [d]\, s \mid * s$		(protection, delimitation, replication)

Table 1
COWS syntax

chronous communication, polyadic synchronization, pattern matching, protection, and delimited killing activities. Due to space limitation, we only present COWS syntax and a glimpse of the semantics, and refer the interested reader to [26] for a full account of the operational semantics, for many examples illustrating COWS peculiarities and expressiveness, and for comparisons with other process-based and orchestration formalisms.

The syntax of COWS is presented in Table 1. It is parameterized by three countable and pairwise disjoint sets: the set of *(killer) labels* (ranged over by $k, k', \ldots$), the set of *values* (ranged over by v, v', ...) and the set of 'write once' *variables* (ranged over by x, y, ...). The set of values is left unspecified; however, we assume that it includes the set of *names*, ranged over by n, m, o, p, ..., mainly used to represent partners and operations. The language is also parameterized by a set of *expressions*, ranged over by e, whose exact syntax is deliberately omitted. We just assume that expressions contain, at least, values and variables, but do not include killer labels (that, hence, are *not* communicable values). We use w to range over values and variables, u to range over names and variables, and d to range over killer labels, names and variables.

Notation $\bar{\cdot}$ stands for tuples of objects, e.g. $\bar{x}$ is a compact notation for denoting the tuple of variables $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$). We assume that variables in the same tuple are pairwise distinct, and that tuples can be arbitrarily nested. Tuples can be constructed using a concatenation operator defined as $\langle a_1, \ldots, a_n \rangle : \langle b_1, \ldots, b_m \rangle = \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$. To single out an element of a tuple, we will write $(\bar{a}, c, \bar{b})$ to denote the tuple $\langle a_1, \ldots, a_n, c, b_1, \ldots, b_m \rangle$, where $\bar{a}$ or $\bar{b}$ might not be present.

Notably, tuples can be used to represent XML documents, the standard format of messages exchanged among web services, by adopting the convention that the first field of each tuple acts as a 'tag' [1] (like originally proposed in the coordination language Linda [19]). For example, the following XML message representing a paper

[1] Element attributes could be rendered in a similar way.

reference:

```
<paper>
  <title>Service discovery and negotiation with COWS</title>
  <authors>
    <author>Lapadula</author>
    <author>Pugliese</author>
    <author>Tiezzi</author>
  </authors>
  <year>2008</year>
</paper>
```

could be rendered through the following COWS tuple:

$$\langle \texttt{paper}, \langle \texttt{title}, \text{Service discovery and negotiation with COWS} \rangle, \langle \texttt{authors}, \langle \texttt{author}, \text{Lapadula} \rangle, \langle \texttt{author}, \text{Pugliese} \rangle, \langle \texttt{author}, \text{Tiezzi} \rangle \rangle \langle \texttt{year}, 2008 \rangle \rangle$$

Thus, to extract the title and the name of the second author of the paper above, one can use the following pattern (as argument of a receive activity):

$$\langle \texttt{paper}, \langle \texttt{title}, x_{title} \rangle, \langle \texttt{authors}, \_, \langle \texttt{author}, x_{secName} \rangle, \_ \rangle, \_ \rangle$$

where, for simplicity sake, we assume that the 'don't care' symbol _ matches any value/tuple.

COWS computational entities (ranged over by s, s', ...) are structured activities built from basic activities, i.e. kill, invoke and receive, by means of choice among receive prefixed terms, parallel composition, protection, delimitation and replication. In the sequel, we shall use $\mathbf{0}$ to denote empty choice and $+$ to abbreviate binary choice. We will omit trailing occurrences of $\mathbf{0}$, writing e.g. $p \bullet o?\bar{w}$ instead of $p \bullet o?\bar{w}.\mathbf{0}$, and write $[d_1, \ldots, d_n]\, s$ in place of $[d_1] \ldots [d_n]\, s$.

Invoke and *receive* are the basic communication activities provided by COWS. An invoke can proceed as soon as evaluation of the expressions in its argument returns the corresponding values. A receive offers an invocable operation along a given partner name and its execution permits to take a decision among alternative behaviours. Besides input parameters and sent values, both activities indicate a communication *endpoint*, i.e. a pair composed of a partner name p and of an operation name o, through which communication should occur. The naming mechanism used to identify endpoints is very flexible. For example, it allows the same service to be identified by means of different logic names (i.e. to play more than one partner role as in WS-BPEL). Thus, the term $p_{slow} \bullet o?\bar{w}.s_{slow} + p_{fast} \bullet o?\bar{w}.s_{fast}$ accepts requests for the same operation o (with parameters $\bar{w}$) through different partners with distinct access modalities: the continuation s_{slow} implements a 'slower service' provided when the request is processed through the partner p_{slow}, while s_{fast} implements a 'faster service' provided when the request arrives through p_{fast}. Additionally, the naming mechanism allows the names composing an endpoint to be dealt with separately, as in a request-response interaction, where usually the service provider knows the name of the response operation, but not the partner name of the service it has to reply to. For example, the term $p \bullet o_{req}?\langle x \rangle.x \bullet o_{res}!\langle \text{"I live"} \rangle$ behaves as a sort of 'ping' service that will know at run-time the partner name for the reply activity, i.e. the service which will be bound to x. In fact, partner and

operation names are dealt with as values and, as such, can be exchanged in communication (although dynamically received names cannot form the endpoints used to receive further invocations). This enables easily modelling many service interaction and reconfiguration patterns.

An inter-service communication takes place when the pattern argument of a receive matches a tuple of values sent along the same endpoint by an invoke and gives rise to substitutions for replacing the variables argument of the receive with the corresponding values argument of the invoke. The substitution for a variable is applied only when the whole scope of the variable is determined and to the term resulting from removing the delimitation. In fact, to enable parallel terms to share the state (or part of it), receive activities in COWS do *not* bind variables. The range of application of the substitution generated by a communication is then regulated by the *delimitation* operator (namely, $[d]\,s$ binds d in the scope s), that is the *only* binder of the calculus. Delimitation is also used to generate fresh names (as the restriction operator of the π-calculus [33]) and to delimit the field of action of kill activities. Execution of a *kill activity* $\mathbf{kill}(k)$ triggers termination of all unprotected parallel terms inside an enclosing $[k]$ (that stops the killing effect). Indeed, critical code can be protected from the effect of a forced termination by using the *protection* operator. Notably, the scope of names and variables can be extended (by using 'structural laws' similar to those dealing with restricted names in the π-calculus), that of killer labels cannot (in fact, they are not communicable values).

Execution of *parallel* terms is interleaved, but when a communication or a kill activity can be performed. Indeed, the former must ensure that, if more than one matching receive is ready to process a given invoke, only the receive with greater priority progresses (i.e. the receive that generates the substitution with 'smaller' domain), while the latter must be executed *eagerly*. Finally, the *replication* operator permits to spawn in parallel as many copies of its argument term as necessary thus, for example, enabling creation of concurrent service instances.

We end this section with a simple example aimed at clarifying some peculiarities of COWS. Consider the following term:

$$[x,y,k]\,(\,p \bullet o_1?\langle x,y\rangle + p \bullet o_2?\langle x\rangle.\mathbf{kill}(k) \;\mid\; \{\!|p' \bullet o_3!\langle x\rangle|\!\} \;\mid\; p' \bullet o_4!\langle x,y\rangle\,) \;\mid\; [n]\,p \bullet o_2!\langle n\rangle$$

Communication of private names exploits scope extension as in the π-calculus. Receive and invoke activities can interact only if both are in the scopes of the delimitations that bind the variables argument of the receive. Hence, to enable communication of private name n, besides the scope of n, we must also extend the scope of variable x as follows:

$$[n,x]\,(\,[y,k]\,(\,p \bullet o_1?\langle x,y\rangle + p \bullet o_2?\langle x\rangle.\mathbf{kill}(k) \;\mid\; \{\!|p' \bullet o_3!\langle x\rangle|\!\} \;\mid\; p' \bullet o_4!\langle x,y\rangle\,) \;\mid\; p \bullet o_2!\langle n\rangle\,)$$

Now, taking place of the communication discards the receive along $p \bullet o_1$ and causes application of the substitution $\{x \mapsto n\}$ to all terms delimited by $[x]$, not only to the continuation of the receive. We thus obtain the term

$$[n,y,k]\,(\,\mathbf{kill}(k) \;\mid\; \{\!|p' \bullet o_3!\langle n\rangle|\!\} \;\mid\; p' \bullet o_4!\langle n,y\rangle\,)$$

Finally, execution of the kill activity only retains the protected invoke.

$$[n]\ \{|p' \bullet o_3!\langle n\rangle|\}$$

In the rest of the paper, we will write $Z_{\bar{v}} \triangleq W$ to assign a symbolic name $Z_{\bar{v}}$ to the term W and to indicate the values $\bar{v}$ occurring within W. We will use $\hat{n}$ to stand for the endpoint $n_p \bullet n_o$ or for the tuple $\langle n_p, n_o\rangle$ and rely on the context to resolve any ambiguity.

3 Using COWS for concurrent constraint programming

We now tailor COWS for specifying Service Level Agreements. We take advantage of the fact that its syntax and operational semantics are parametrically defined with respect to the set of *values*, the syntax of *expressions* that operate on values and, therefore, the definition of the *pattern-matching* function. We show that, by specializing these parameters, we can obtain a dialect that properly integrates the principle of 'computing with partial information', or constraints, that is at the basis of the concurrent constraint programming paradigm [38].

We first provide some insights on the constraint system used. In COWS, a *constraint* is a relation among a specified set of variables which gives some information on the set of possible values that these variables may assume. Such information is usually not complete since a constraint may be satisfied by several assignments of values to the variables. For example, we can employ constraints such as

$$\mathtt{cost} \geqslant 350 \qquad \mathtt{cost} = \mathtt{bw} \cdot 0.05 \qquad \mathtt{z} = \frac{1}{1 + |\mathtt{x} - \mathtt{y}|}$$

In practice, we do not take a definite standing on which of the many kind of constraints to use. From time to time, the appropriate kind of constraints to work with should be chosen depending on what one intends to model.

Formally a constraint c is represented as a function $c : (V \to D) \to \{\mathbf{true}, \mathbf{false}\}$, where V is the set of constraint variables (that, as explained in the sequel, is included in the set of COWS names), and D is the domain of interpretation of V, i.e. the domain of values that the variables may assume. If we let $\eta : V \to D$ be an assignment of domain elements to variables, then a constraint is a function that, given an assignment η, returns a truth value indicating if the constraint is satisfied by η. For instance, the assignment $\{\mathtt{cost} \mapsto 500\}$ satisfies the first constraint, while $\{\mathtt{cost} \mapsto 500, \mathtt{bw} \mapsto 8000\}$ does not satisfy the second constraint, that is, instead, satisfied by $\{\mathtt{cost} \mapsto 400, \mathtt{bw} \mapsto 8000\}$. An assignment that satisfies a constraint is called a *solution*.

The constraints we have presented are called *crisp* in the literature, because they can only be satisfied or violated. In fact, we can also use more general constraints called *soft constraints* [18,6]. These constraints, given an assignment for the variables, return an element of an arbitrary constraint semiring (*c-semiring*, [5]), namely a partially ordered set of 'preference' values equipped with two suitable operations for combination ($\times$) and comparison ($+$) of (tuples of) values and constraints. Formally, a c-semiring is an algebraic structure $\langle A, +, \times, 0, 1\rangle$ such that:

A is a set and $0, 1 \in A$; $+$ is a binary operation on A that is commutative, associative, idempotent, 0 is its unit element and 1 is its absorbing element; $\times$ is a binary operation on A that is commutative, associative, distributes over $+$, 1 is its unit element and 0 is its absorbing element. Operation $+$ induces a partial order $\leq$ on A defined by $a \leq b$ iff $a + b = b$, which means that a is more constrained than b. The minimal element is thus 0 and the maximal 1. For example, crisp constraints can be understood as soft constraints on the c-semiring $\langle\{\mathbf{true}, \mathbf{false}\}, \vee, \wedge, \mathbf{false}, \mathbf{true}\rangle$ of the boolean values.

The COWS dialect we work with in the rest of the paper specializes expressions to also include *constraints*, ranged over by c, and *constraint multisets*, ranged over by C, and to be formed by using the following operators.

- *Consistency check:* predicate $isCons(C)$ takes a constraint multiset C and holds true if C is consistent. Formally, $isCons(\{c_1, \ldots, c_n\})$ holds true if there exists an assignment η such that $c_1\eta \wedge \ldots \wedge c_n\eta \neq \mathbf{false}$, i.e. if the combination of all constraints has at least a solution[2]. The predicate $isCons(\_)$ is defined for crisp constraints. However, we can generalize its definition to soft constraints by requiring that it is satisfied if there exists an assignment η such that $c_1\eta \times \ldots \times c_n\eta \neq 0$.
- *Entailment check:* predicate $C \vdash c$ takes a constraint multiset C and a constraint c and holds true if c is entailed by C. Formally, $\{c_1, \ldots, c_n\} \vdash c$ holds true if there exists an assignment η such that $c_1\eta \wedge \ldots \wedge c_n\eta \leq_B c\eta$, where $\leq_B$ is the partial ordering over booleans, defined by $b_1 \leq_B b_2$ iff $b_1 \vee b_2 = b_2$. Also this predicate can be generalized to soft constraints by requiring that $\{c_1, \ldots, c_n\} \vdash c$ holds true if there exists an assignment η such that $c_1\eta \times \ldots \times c_n\eta \leq c\eta$.
- *Retraction:* operation $C - c$ takes a constraint multiset C and a constraint c and returns the multiset $C \backslash \{c\}$ if $c \in C$, otherwise returns C.
- *Multiset union:* binary operator $\uplus$ is the standard union operator between multisets.

Since constraints and constraint multisets are expressions, they need to be evaluated. The (expression) evaluation function $[\![\_]\!]$ acts on constraints and constraint multisets as the identity, except for constraints containing COWS variables, for which the function is undefined. Therefore, evaluated constraints and constraint multisets are values that can be communicated by means of synchronization of invoke and receive activities and can replace variables by means of application of substitutions to terms. *Substitutions* (ranged over by σ) map variables to values and are written as collections of pairs of the form $x \mapsto v$.

To efficiently implement the primitives of the concurrent constraint programming paradigm, we tailor the rules defining the pattern-matching function $\mathcal{M}(\_, \_)$ to deal with constraints and operations on them, as shown in Table 2. The original matching rules (shown in the upper part of the table) are still valid and state that variables match any value (thus, e.g., $\mathcal{M}(x, C) = \{x \mapsto C\}$), two values match only

[2] We do not consider here the well-studied problem of solving a constraint system. Among the many techniques exploited to this aim, we mention dynamic programming [34,4] and branch and bound search [43].

$$\mathcal{M}(x,v) = \{x \mapsto v\} \qquad \mathcal{M}(v,v) = \emptyset \qquad \frac{\mathcal{M}(a_1,b_1) = \sigma_1 \quad \mathcal{M}(\bar{a}_2,\bar{b}_2) = \sigma_2}{\mathcal{M}((a_1,\bar{a}_2),(b_1,\bar{b}_2)) = \sigma_1 \uplus \sigma_2}$$

$$\frac{isCons(C \uplus \{c\})}{\mathcal{M}(\langle c, x\rangle, C) = \{x \mapsto C\}} \qquad \frac{C \vdash c}{\mathcal{M}(\langle c^{\vdash}, x\rangle, C) = \{x \mapsto C\}}$$

Table 2
Matching rules

if they are identical, and two tuples match if they have the same number of fields and corresponding fields do match. The new rules (shown in the lower part of the table) allow a two-field tuple to match a single value in two specific cases: a tuple $\langle c, x\rangle$ and a multiset of constraints C do match if $C \uplus \{c\}$ is consistent, while a tuple $\langle c^{\vdash}, x\rangle$ and a multiset of constraints C do match if c is entailed by C; in both cases, the substitution $\{x \mapsto C\}$ is returned. Notably, by applying the operator $\_^{\vdash}$ to a constraint one can require an entailment check instead of a consistency check.

The concurrent constraint computing model is based on a shared store of constraints that provides partial information about possible values that variables can assume. In COWS the store of constraints is represented by the following service:

$$store_C \triangleq [\hat{n}]\,(\,\hat{n}!\langle C\rangle \mid *\,[x]\,\hat{n}?\langle x\rangle.(\,p_s \bullet o_{get}!\langle x\rangle \mid [y]\,p_s \bullet o_{set}?\langle y\rangle.\hat{n}!\langle y\rangle\,)\,)$$

where p_s is a distinguished partner, o_{get} and o_{set} are distinguished operations. Other services can interact with the store service in mutual exclusion, by acquiring the lock (and, at the same time, the stored value) with a receive along $p_s \bullet o_{get}$ and by releasing the lock (providing the new stored value) with an invoke along $p_s \bullet o_{set}$. Notably, local stores of constraints can be simply modelled by restricting the scope of the partner name p_s.

The store is composed in parallel with the other services, which can act on it by performing operations for adding/removing constraints to/from the store (`tell` and `retract`, respectively), and for checking entailment/consistency of a constraint by/with the store (`ask` and `check`, respectively). These four operations can be rendered in COWS as follows:

$$\langle\!\langle \texttt{tell}\ c.s\rangle\!\rangle = [\hat{n}]\,(\,\hat{n}!\langle c\rangle \mid [y]\,\hat{n}?\langle y\rangle.[x]\,p_s \bullet o_{get}?\langle\langle y, x\rangle\rangle.(\{\!|\,p_s \bullet o_{set}!\langle x \uplus \{y\}\rangle\,|\!\}\mid \langle\!\langle s\rangle\!\rangle)\,)$$

$$\langle\!\langle \texttt{ask}\ c.s\rangle\!\rangle = [\hat{n}]\,(\,\hat{n}!\langle c^{\vdash}\rangle \mid [y]\,\hat{n}?\langle y\rangle.[x]\,p_s \bullet o_{get}?\langle\langle y, x\rangle\rangle.(\{\!|\,p_s \bullet o_{set}!\langle x\rangle\,|\!\}\mid \langle\!\langle s\rangle\!\rangle)\,)$$

$$\langle\!\langle \texttt{check}\ c.s\rangle\!\rangle = [\hat{n}]\,(\,\hat{n}!\langle c\rangle \mid [y]\,\hat{n}?\langle y\rangle.[x]\,p_s \bullet o_{get}?\langle\langle y, x\rangle\rangle.(\{\!|\,p_s \bullet o_{set}!\langle x\rangle\,|\!\}\mid \langle\!\langle s\rangle\!\rangle)\,)$$

$$\langle\!\langle \texttt{retract}\ c.s\rangle\!\rangle = [\hat{n}]\,(\,\hat{n}!\langle c\rangle \mid [y]\,\hat{n}?\langle y\rangle.[x]\,p_s \bullet o_{get}?\langle x\rangle.(\{\!|\,p_s \bullet o_{set}!\langle x - y\rangle\,|\!\}\mid \langle\!\langle s\rangle\!\rangle)\,)$$

where $\hat{n}$ is fresh. Essentially, each operation is a term that first takes the store of constraints (thus acquiring the lock so that other services cannot concurrently interact with the store) and then returns the (possibly) modified store (thus releasing the lock). Since the invoke activities $\hat{n}!\langle c\rangle$ and $\hat{n}!\langle c^{\vdash}\rangle$ can be performed only if $[\![c]\!]$ is defined, i.e. if c does not contain COWS variables, the store can only contain evaluated constraints. Availability of the store is guaranteed by the fact that, once the store and the lock have been acquired, the activities reintroducing the store and releasing the lock are protected from the effect of kill activities. This disciplined use of the store permits to preserve its consistency. Notably, the matching rules in the lower part of Table 2 are essential for faithfully modelling the semantics of the original operations.

While `tell` and `ask` are the classical concurrent constraint programming primitives, operations `check` and `retract` are borrowed from [9]. In particular, operation `retract` is debatable since its adoption prevents the store of constraints to be 'monotonically' refined. In fact, in concurrent constraint programming a computation step does not change the value of a variable, but may rule out certain values that were previously possible; therefore, the set of possible values for the variable is contained in the set of possible values at any prior step. This monotonic evolution of the store during computations permits to define the result of a computation as the least upper bound of all the stores occurring along the computation and provides concurrent constraint languages with a simple denotational semantics in which programs are identified to closure operators on the semi-lattice of constraints [39]. Therefore, if one wants to exploit some of the properties of concurrent constraint programming that require monotonicity, he must consider the fragment of COWS without `retract`. On the other hand, in the context of dynamic service discovery and negotiation, the use of operation `retract` enables modelling many frequent situations where it is necessary to remove a constraint from the store for, e.g., weakening a request.

To avoid interference between communication and operations on the store, and to correctly implement the operation `retract`, we do not allow constraints in the store to contain variables, thus they cannot change due to application of substitutions generated by communication. Indeed, suppose constraints in the store may contain variables and consider the following example:

$$[x]\,(\,store_{\emptyset} \mid \texttt{tell}(x \leq 5).\,(\hat{n}!\langle 6\rangle \mid \hat{n}?\langle x\rangle)\,)$$

After action `tell` has added the constraint $x \leq 5$ to the store, communication along the endpoint $\hat{n}$ can modify the constraint in $6 \leq 5$. This way, the communication can make the store inconsistent. As another example, consider the following term:

$$[x]\,(\,store_{\{x\leq 5\}} \mid \texttt{tell}(x = 3).\,\texttt{retract}(x = 3).\,\texttt{tell}(x = 4)\,)$$

where actions `tell` and `retract` can modify the store while preserving its consistency. Now, let us change the term as follows:

$$[x]\,(\,store_{\{x\leq 5\}} \mid \texttt{tell}(x = 3).\,(\,\hat{n}!\langle 3\rangle \mid \hat{n}?\langle x\rangle \mid \texttt{retract}(x = 3).\,\texttt{tell}(x = 4)\,)\,)$$

After the first `tell` has taken place, due to the communication along $\hat{n}$ the substitution $\{x \mapsto 3\}$ will be applied to the store, thus obtaining the term:

$$store_{\{3\leq 5\}} \mid \mathtt{retract}(3 = 3).\mathtt{tell}(3 = 4)$$

Now, although the store is still consistent, action `retract` cannot modify it. This means that the write-once variables of COWS are not suitable for modelling constraint variables.

Therefore, as we stated before, we do not allow constraints in the store to contain variables. Instead, they can use specific names, that we call *constraint variables* and, for the sake of presentation, write as $\mathtt{x}, \mathtt{y}, \ldots$ (i.e. in the typewriter style). Indeed, names are not affected by expression evaluation (i.e. $[\![\mathtt{x}]\!] = \mathtt{x}$) and by substitution application (i.e. $\mathtt{x} \cdot \sigma = \mathtt{x}$). Moreover, names can be delimited, thus allowing us to model *local constraints*. In the sequel, we will use $cv(t)$ to denote the set of constraint variables occurring in a term t. Notice however that constraints occurring as arguments of operations may contain variables so that we can specify constraints that will be dynamically determined. For example, we can write $\mathtt{tell}\ (\mathtt{cost} \geqslant x_{min\_cost}).s$; of course, since $[\![\mathtt{cost} \geqslant x_{min\_cost}]\!]$ is undefined, this operation is blocked until variable x_{min_cost} is substituted by a value.

4 Communication protocols for constraints generation

Besides `ask`, `tell`, `retract` and `check`, inter-service communication can be used to implement many protocols allowing two parties to generate new constraints. For instance, in [9], service synchronization works like two global `ask` and `tell` constructs: as a result of the synchronization between the output $\bar{x}\langle y\rangle$ and the input $x\langle y'\rangle$ the new constraint $y = y'$ is added to the store. Therefore, synchronization allows local constraints (i.e. constraints with restricted names) to interact, thus establishing an SLA between the two parties, and (possibly) to become globally available. Differently, COWS does not allow communication to directly generate new constraints: e.g., an invoke $p \bullet o!\langle \mathtt{x}\rangle$ and a receive $p \bullet o?\langle \mathtt{y}\rangle$ cannot synchronize, because $\mathcal{M}(\mathtt{y}, \mathtt{x})$ does not hold. In the rest of this section, we present three example protocols that permit establishing new constraints. For the sake of readability, in the protocols we will use a sort of conditional choice, that can be thought of as a 'macro' encodable in COWS as follows

$$\langle\!\langle \textbf{if } (e) \textbf{ then } \{s_1\} \textbf{ else } \{s_2\}\rangle\!\rangle = [\hat{m}]\,(\hat{m}!\langle e\rangle \mid (\hat{m}?\langle \textbf{true}\rangle.\langle\!\langle s_1\rangle\!\rangle + \hat{m}?\langle \textbf{false}\rangle.\langle\!\langle s_2\rangle\!\rangle)\,)$$

where $\hat{m}$ is fresh, and **true** and **false** are the values that can result from evaluation of e.

A simple protocol.
To create constraints of the form $\mathtt{x} = \mathtt{y}$, where each of `x` and `y` is initially local to only one party, we can use the standard COWS communication mechanism together with operation `tell`. For example, the following term

$$store_C \mid p \bullet o!\langle \mathtt{x}\rangle \mid [z]\, p \bullet o?\langle z\rangle.\mathtt{tell}\ (z = \mathtt{y}).s \qquad (*)$$

for z fresh in s, adds to the store the constraint $\mathtt{x} = \mathtt{y}$, if it is consistent with C.

Indeed, the communication along endpoint $p \bullet o$ takes place before the consistency check (performed by operation `tell`), and the term evolves into

$$store_C \mid \mathtt{tell}\ (\mathtt{x} = \mathtt{y}).s$$

Now, if $\mathtt{x} = \mathtt{y}$ is not consistent with the store, the receive and invoke activities along $p \bullet o$ are definitively consumed and the execution of term s is blocked.

This protocol is simple and divergence-free, but it may introduce deadlocked states in the terms. This fact has a relevant impact on the specification of protocols for negotiation, particularly when there are more parties that provide (or require) the same service. For example, consider the following term

$$store_C \mid p \bullet o!\langle \mathtt{x} \rangle \mid [z]\, p \bullet o?\langle z \rangle.\mathtt{tell}\ (z = \mathtt{y}).s \mid [z']\, p \bullet o?\langle z' \rangle.\mathtt{tell}\ (z' = \mathtt{w}).s'$$

where another receive activity is put in parallel with term $(*)$. Now, if $\mathtt{x} = \mathtt{y}$ is not consistent with C, then the term can non-deterministically evolve in a stuck state by performing the receive $p \bullet o?\langle z \rangle$, although $\mathtt{x} = \mathtt{w}$ might be consistent with C.

A divergent protocol.

To overtake the previous problems, the following more refined protocol restores the communication activities if the constraints generated when communication takes place are not consistent with the current store. To simplify the encoding, we assume that a single communication cannot produce both substitutions and new constraints. The extended communication activities can be rendered as follows:

$$\langle\!\langle p \bullet o!\bar{e} \rangle\!\rangle = [\hat{n}]\,(\,\hat{n}!\langle\rangle \mid *\,\hat{n}?\langle\rangle.[\hat{m}]\,(\,p \bullet o!(\hat{m}, \bar{e}) \mid \hat{m}?\langle\rangle.\hat{n}!\langle\rangle\,)\,)$$

$$\langle\!\langle p \bullet o?\bar{w}.s \rangle\!\rangle = \begin{cases} [x]\, p \bullet o?(x, \bar{w}).\langle\!\langle s \rangle\!\rangle & \text{if } cv(\bar{w}) = \emptyset \\ s' & \text{if } \bar{w} = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle \\ undef & \text{otherwise} \end{cases}$$

$$\begin{aligned} s' \triangleq\ & [\hat{n}]\,(\,\hat{n}!\langle\rangle \mid *\,\hat{n}?\langle\rangle.[x, x_1, \ldots, x_n]\, p \bullet o?\langle x, x_1, \ldots, x_n \rangle.[y]\, p_s \bullet o_{get}?\langle y \rangle. \\ & [\hat{r}]\,(\,\{\!|\ \mathbf{if}\ (isCons(y \uplus \{\mathtt{x}_1 = x_1, \ldots, \mathtt{x}_n = x_n\})) \\ & \qquad \mathbf{then}\ \{\, p_s \bullet o_{set}!\langle y \uplus \{\mathtt{x}_1 = x_1, \ldots, \mathtt{x}_n = x_n\} \rangle \mid \hat{r}!\langle\rangle\,\} \\ & \qquad \mathbf{else}\ \{\, p_s \bullet o_{set}!\langle y \rangle \mid x!\langle\rangle \mid \hat{n}!\langle\rangle\,\}\ |\!\!\} \\ & \mid \hat{r}?\langle\rangle.\langle\!\langle s \rangle\!\rangle\,)\,) \end{aligned}$$

for $\hat{n}$, $\hat{m}$ and $\hat{r}$ fresh in $\bar{e}$, $\bar{w}$ and s. An invoke activity is encoded as a term that performs the same invoke with, as an additional argument, a private endpoint $\hat{m}$ where, if communication fails, it waits for an acknowledgement that triggers the restart of the term. A receive activity with a tuple, as an argument, without constraint variables is encoded as a term that performs the same receive with, as an additional argument, a dummy variable (i.e. x), that stores the endpoint for the acknowledgement (that, however, is not sent in this case). A receive activity with, as an argument, a tuple of constraint variables is encoded as a term that performs a receive with a tuple of COWS variables which store the endpoint for the acknowl-

edgement and the received data (i.e. constraint variables or values). After the receive, the term takes the current store of constraints and checks its consistency with the constraints that would be generated by taking place of the communication. In the positive case, it updates the store with the new constraints and triggers the (encoding of the) continuation term s by a signal along the endpoint $\hat{r}$. Otherwise, it leaves unchanged the store, sends an ack back to the corresponding invoking term, and restarts its execution. As in the encodings of concurrent constraint programming primitives, in order to guarantee the release of the lock on the shared store of constraints, the activities following the acquisition of the lock are protected.

The encoded receives can be terms like s', hence they cannot be used as guards of a choice. Therefore, to implement a choice between encoded receives, they must be put in parallel and synchronized by using a lock (as in [35]).

Communication can generate constraints expressing equalities between names (alike fusions of [9]) or equalities between names and values. For example, the invoke $p \bullet o!\langle \mathtt{x} \rangle$ and the receive $p \bullet o?\langle \mathtt{y} \rangle$ can synchronize and add the constraint $\mathtt{x} = \mathtt{y}$ to the store, if consistency is preserved, otherwise the synchronization is forbidden. Similarly, the receive above can synchronize with the invoke $p \bullet o!\langle v \rangle$ and generate the constraint $\mathtt{y} = v$. Let us now consider the following term

$$store_C \mid \langle\!\langle p \bullet o!\langle \mathtt{x} \rangle \rangle\!\rangle \mid \langle\!\langle p \bullet o?\langle \mathtt{y} \rangle.s \rangle\!\rangle \mid \langle\!\langle p \bullet o?\langle \mathtt{w} \rangle.s' \rangle\!\rangle$$

and assume that $\mathtt{x} = \mathtt{w}$ is consistent with the store while $\mathtt{x} = \mathtt{y}$ is not. In this case, by performing the receive $p \bullet o?\langle \mathtt{y} \rangle$ the term will come back to the initial state, while by performing the receive $p \bullet o?\langle \mathtt{w} \rangle$ it becomes $store_{C \uplus \{\mathtt{x}=\mathtt{w}\}} \mid \langle\!\langle p \bullet o?\langle \mathtt{y} \rangle.s \rangle\!\rangle \mid \langle\!\langle s' \rangle\!\rangle$ where, for simplicity sake, we omit the stuck terms produced by the encoding.

Of course, since the protocol can diverge (i.e. an invoke can synchronize infinitely often with the same receive without modifying the store), a *fairness assumption* is essential to guarantee progress properties: if an invoke can synchronize with many receives and at least one synchronization produces consistent constraints, then eventually this synchronization will succeed.

A divergence-free protocol.
To get rid of divergence, we could add the following pattern-matching rule

$$\frac{|\bar{\mathtt{x}}| = |\bar{x}| = |\bar{v}| \qquad isCons(C \uplus \{\bar{\mathtt{x}} = \bar{v}\})}{\mathcal{M}((\bar{\mathtt{x}} : \bar{x}, y), (\bar{v}, C)) = \{\bar{x} \mapsto \bar{v}, y \mapsto C\}}$$

(notice that, since the tuples $(\bar{\mathtt{x}} : \bar{x}, y)$ and $(\bar{v}, C)$ have different length, the rule does not interfere with the other ones) and encode communication activities as follows:

$$\begin{aligned} \langle\!\langle p \bullet o!\bar{e} \rangle\!\rangle = {} & [\hat{n}]\,(\,\hat{n}!\langle\rangle \mid *\,\hat{n}?\langle\rangle.[\hat{m}]\,(\,[x]\,p_s \bullet o_{get}?\langle x \rangle. \\ & (\,p \bullet o!\langle(\bar{e}, x), \hat{m}\rangle \mid \{\!|\, p_s \bullet o_{set}!\langle x \rangle \,|\!\}\mid \hat{m}?\langle\rangle.\hat{n}!\langle\rangle\,)\,) \end{aligned}$$

$$\langle\!\langle p \bullet o?\bar{w}.s \rangle\!\rangle = \begin{cases} [z, x]\,p \bullet o?\langle(\bar{w}, z), x\rangle.\langle\!\langle s \rangle\!\rangle & \text{if } cv(\bar{w}) = \emptyset \\ s' & \text{if } \bar{w} = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle \\ undef & \text{otherwise} \end{cases}$$

$$
\begin{aligned}
s' \triangleq\ & [\hat{n}]\,(\,\hat{n}!\langle\rangle \mid *\ \hat{n}?\langle\rangle.[x_1,\ldots,x_n,z,x]\,p_{\bullet}o?\langle\langle \mathtt{x}_1,\ldots,\mathtt{x}_n,x_1,\ldots,x_n,z\rangle,x\rangle.\\
& [y]\,p_s{\bullet}o_{get}?\langle y\rangle.\\
& [\hat{r}]\,(\,\{\!|\,\mathbf{if}\ (y == z)\\
& \qquad \mathbf{then}\ \{\,p_s{\bullet}o_{set}!\langle y \uplus \{\mathtt{x}_1 = x_1,\ldots,\mathtt{x}_n = x_n\}\rangle \mid \hat{r}!\langle\rangle\,\}\\
& \qquad \mathbf{else}\ \{\,p_s{\bullet}o_{set}!\langle y\rangle \mid x!\langle\rangle \mid \hat{n}!\langle\rangle\,\}\,|\!\}\\
& \mid \hat{r}?\langle\rangle.\langle\!\langle s\rangle\!\rangle\,)\,)
\end{aligned}
$$

for $\hat{n}$, $\hat{m}$ and $\hat{r}$ fresh in $\bar{e}$, $\bar{w}$ and s. Essentially, the encoding of the invoke reads the store and releases it, invoke and receive activities synchronize (i.e. the new constraints are consistent with the store), the encoding of the receive reads the store and, if the value is unchanged, adds the new constraints, otherwise it restarts the terms. The encoding is divergence-free in the sense that whenever the terms are restarted the value of the store of constraints differs from that in the previous execution, namely the terms *cannot stutter* on the same store. Of course, more robust protocols (that, e.g., avoid also starvation) could be defined. However, they should rely on synchronization among more than two entities at the same time (as it is permitted, e.g., by the join input of the Join-calculus [17]), that goes against our choice to reconcile expressiveness and implementability.

5 A Web hosting scenario

We now present a scenario, inspired by [9], that shows how our framework can be used to model both automatic service discovery mechanisms and negotiation mechanisms with the aim of achieving service level agreements. Consider a client C that needs a web hosting service, and some service providers P_1, P_2, ..., that offer different Web hosting solutions, varying in cost and bandwidth. In order to be invoked by clients, provider services need to be discovered. The dynamic service discovery mechanism relies on a (single) registry R that, similarly to an UDDI registry, allows providers to publish their service description, as a WSDL document, and clients to discover published services by performing search queries. Suppose that providers obtain their bandwidth resources from third parties T_1, T_2, ..., and that each provider can cover only a delimited geographical area. The whole system results from the parallel composition of C, R, $store_{\emptyset}$ and all provider and third party services. In the following COWS specification of this scenario, we implicitly rely on one communication protocol of those presented in Section 4.

The client service C is defined as

$$
\begin{aligned}
\mathsf{C} \triangleq\ & p_{\mathsf{R}}{\bullet}o_{search}!\langle \text{“}web\ hosting\text{”}, p_{\mathsf{C}}, (\mathtt{zip} = 10012)\rangle\\
& \mid\ [x]\,p_{\mathsf{C}}{\bullet}o_{corr}?\langle x\rangle.\,[k]\ *\ [x_{\mathsf{P}}]\,p_{\mathsf{C}}{\bullet}o_{resp}?\langle x_{\mathsf{P}}\rangle.\\
& \qquad\qquad (\,x_{\mathsf{P}}{\bullet}o_{req}!\langle p_{\mathsf{C}}\rangle \mid [z]\,p_{\mathsf{C}}{\bullet}o_{startNeg}?\langle z\rangle.\,\mathsf{C}^{\,neg}\,)
\end{aligned}
$$

The endpoint $p_{\mathsf{R}}{\bullet}o_{search}$ is used to perform a query on the registry and transmit the client's partner name p_{C}. Besides the string identifying the kind of required service,

a query contains a constraint identifying the location of the client [3]. The registry will reply by sending along $p_{\mathsf{C}} \bullet o_{corr}$ a private name used to send a stop signal to the registry, and along $p_{\mathsf{C}} \bullet o_{resp}$ all partner names corresponding to providers that satisfy the query. For each of them, an instance of the client is created that starts a negotiation phase, implemented by term C^{neg}. We assume that once a negotiation succeeds, C^{neg} forces termination of all the other parallel instances (by performing $\mathbf{kill}(k)$) and sends a signal to the registry to stop the database querying (by performing $\{\!|p_{\mathsf{R}} \bullet o_{stop}!\langle x\rangle|\!\}$). Client C issues one request at a time; in case of concurrent queries, correlation can be exploited to relate a query and its answers. The more general case of multiple clients can still be dealt with by using correlation or by relying on the (very reasonable) assumption that clients' partner names are pairwise distinct.

A service provider is defined alike the following term P

$$\begin{aligned}\mathsf{P} \triangleq\ & p_{\mathsf{R}} \bullet o_{pub}!\langle \text{“}web\ hosting\text{”}, p_{\mathsf{P}}, ((\mathtt{zip} \geqslant 10000) \wedge (\mathtt{zip} \leqslant 14905))\rangle \\ & \mid *\,[x_{\mathsf{C}}]\, p_{\mathsf{P}} \bullet o_{req}?\langle x_{\mathsf{C}}\rangle.\, [p]\,(\, x_{\mathsf{C}} \bullet o_{startNeg}!\langle p\rangle \mid p_{\mathsf{T}} \bullet o_{startNeg}!\langle p\rangle \\ & \qquad\qquad \mid [x_{\mathsf{T}}]\, p \bullet o_{third}?\langle x_{\mathsf{T}}\rangle.\, \mathsf{P}^{neg}\,)\end{aligned}$$

The endpoint $p_{\mathsf{R}} \bullet o_{pub}$ is used for invoking the registry service and transmitting the description of the provided service. This description consists of a string identifying the kind of provided service, the provider's partner name p_{P}, and a constraint that defines the area covered by the provider [4] which, of course, may differ from provider to provider. Notably, `zip` is a global constraint variable. Each request sent by a client, say C, triggers a new negotiation phase. Specifically, when a request arrives along the endpoint $p_{\mathsf{P}} \bullet o_{req}$, the provider creates a new instance that generates a new partner name p, that defines a private endpoint used to receive from C and from the considered third party, say T. Indeed, the provider instance sends p to C and T. After that, T replies with another private partner name (stored in x_{T}), that allows the instance of P to interact with the correct instance of T, and the provider instance continues as P^{neg}.

The registry service R is defined as

$$\begin{aligned}\mathsf{R} \triangleq\ & [\hat{n}]\,(\, *\,[x_{type}, x_{\mathsf{P}}, x_c]\, p_{\mathsf{R}} \bullet o_{pub}?\langle x_{type}, x_{\mathsf{P}}, x_c\rangle.\, \hat{n}!\langle x_{type}, x_{\mathsf{P}}, x_c\rangle \\ & \quad \mid *\,[x_{type}, x_{\mathsf{C}}, x'_c]\, p_{\mathsf{R}} \bullet o_{search}?\langle x_{type}, x_{\mathsf{C}}, x'_c\rangle. \\ & \qquad\qquad [id]\,(\, x_{\mathsf{C}} \bullet o_{corr}!\langle id\rangle \mid [p_s]\,(\, store_{\emptyset} \mid \mathtt{tell}\, x'_c.\, \mathsf{R}'\,)\,)\,)\end{aligned}$$

$$\begin{aligned}\mathsf{R}' \triangleq\ & [k]\,(\, *\,[x_{\mathsf{P}}, x_c]\, \hat{n}?\langle x_{type}, x_{\mathsf{P}}, x_c\rangle. \\ & \qquad\qquad (p_{\mathsf{R}} \bullet o_{pub}!\langle x_{type}, x_{\mathsf{P}}, x_c\rangle \mid \mathtt{check}\, x_c.\, x_{\mathsf{C}} \bullet o_{resp}!\langle x_{\mathsf{P}}\rangle) \\ & \quad \mid p_{\mathsf{R}} \bullet o_{stop}?\langle id\rangle.\, \mathbf{kill}(k)\,)\end{aligned}$$

For each publication request received along the endpoint $p_{\mathsf{R}} \bullet o_{pub}$ from a provider service, the registry service emits a tuple containing the service description along

[3] A client position is expressed by a zip code. In the example, the client C is located at the Computer Science Department of the New York University.

[4] A geographical area is defined by a set of United States Postal Service zip codes. In the example, the provider P covers the whole State of New York.

the private endpoint $\hat{n}$. The parallel composition of these outputs represents the database of the registry service. When a client request is received along $p_{\mathsf{R}} \bullet o_{search}$, R replies by sending a new correlation identifier id, that will be used to correlate stop signals sent from the client along $p_{\mathsf{R}} \bullet o_{stop}$, and initializes a new local store by adding the constraint within the query message. Then, it cyclically reads a tuple (whose first field is the string specified by the client) from the internal database, checks if the provider constraints are consistent with the store and, in case of success, sends the provider's partner name to the client. Notably, reading a tuple in the database, in this case, consists of an input along $\hat{n}$ followed by an output along $p_{\mathsf{R}} \bullet o_{pub}$; this way we are guaranteed that, after being consumed, the tuple is correctly added to the database. The termination of the loop is triggered by the receiving of a signal along $p_{\mathsf{R}} \bullet o_{stop}$. It is worth noticing that database tuples are non-deterministically chosen, thus the same provider name can be sent many times. This could be avoided by refining the specification, e.g. by tagging each tuple with an identifier (stored in an additional field), that permits reading the tuples in an orderly way.

Finally, the third party T which P relies on is defined as

$$\mathsf{T} \triangleq *\,[x_{\mathsf{P}}]\, p_{\mathsf{T}} \bullet o_{startNeg}?\langle x_{\mathsf{P}} \rangle .\, [p']\, (\, x_{\mathsf{P}} \bullet o_{third}!\langle p' \rangle \mid \mathsf{T}^{\,neg}\,)$$

Once the discovery phase terminates, the client, the selected provider and the corresponding third party initiate the negotiation phase, in order to sign an SLA contract before the execution of the service. Notably, the success of the negotiation also depends on the resources provided by the third party.

Each party specifies its SLA requirements or guarantees: the client C imposes that 600 Euro is the maximum cost it is willing to pay for the service; the provider P indicates the minimum annual cost of 350 Euro for the service and the cost per unit of bandwidth $\mathtt{cost} = \mathtt{bw} \cdot 0.05$; and the third party T fixes the maximum bandwidth that it can supply at a rate of $10^{\cdot}000$ Mbit/s. Thus, we have

$$\begin{aligned}
\mathsf{C}^{\,neg} \triangleq\ & [\mathtt{bw}', \mathtt{cost}']\, \mathtt{tell}\ (\mathtt{cost}' \leqslant 600). \\
& (\, z \bullet o_{sync}!\langle \mathtt{bw}', \mathtt{cost}' \rangle \mid p_{\mathsf{C}} \bullet o_{sign}?\langle\rangle .\, (\, z \bullet o_{ackC}!\langle\rangle \mid \mathsf{C}'\,)\,)
\end{aligned}$$

$$\begin{aligned}
\mathsf{C}' \triangleq\ & [x', y]\, p_{\mathsf{C}} \bullet o_{fix}?\langle x', y \rangle . \\
& \mathtt{check}\ ((x' = \mathtt{bw}') \wedge (y = \mathtt{cost}')).(\mathbf{kill}(k) \mid \{\!|\, p_{\mathsf{R}} \bullet o_{stop}!\langle x \rangle \,|\!\})
\end{aligned}$$

$$\begin{aligned}
\mathsf{P}^{\,neg} \triangleq\ & [\mathtt{bw}, \mathtt{cost}]\, \mathtt{tell}\ (\,(\mathtt{cost} \geqslant 350) \wedge (\mathtt{cost} = \mathtt{bw} \cdot 0.05)\,) . \\
& p \bullet o_{sync}?\langle \mathtt{bw}, \mathtt{cost} \rangle . \\
& (x_{\mathsf{T}} \bullet o_{sync}!\langle \mathtt{bw} \rangle \mid p \bullet o_{ackT}?\langle\rangle .\, (x_{\mathsf{C}} \bullet o_{sign}!\langle\rangle \mid p \bullet o_{ackC}?\langle\rangle .\, \mathsf{P}')\,)
\end{aligned}$$

$$\begin{aligned}
\mathsf{P}' \triangleq\ & p_{\mathsf{P}} \bullet o_{reqMetrics}!\langle p \rangle \ \mid\ [x_{bw}]\, p \bullet o_{metrics}?\langle x_{bw} \rangle . \\
& (\, x_{\mathsf{C}} \bullet o_{fix}!\langle x_{bw}, x_{bw} \cdot 0.05 \rangle \mid \mathtt{check}\ ((x_{bw} \cdot 0.05 = \mathtt{cost}) \wedge (x_{bw} = \mathtt{bw}))\,)
\end{aligned}$$

$$\mathsf{T}^{\,neg} \triangleq [\mathtt{bw}'']\, \mathtt{tell}\ (\mathtt{bw}'' \leqslant 10^{\cdot}000).\, p' \bullet o_{sync}?\langle \mathtt{bw}'' \rangle .\, x_{\mathsf{P}} \bullet o_{ackT}!\langle\rangle$$

Each party starts by adding its local constraints (i.e. constraints with restricted constraint variables) to the shared global store by performing an operation `tell`. Then, for sharing the local constraints, all parties synchronize each other by invoking operation o_{sync} (that each party provides). Finally, since all constraints are consistent, by communicating along $p_{\mathsf{C}} \bullet o_{sign}$ and $p \bullet o_{ackC}$, C and P sign the following contract:

$$(\mathtt{cost} \geqslant 350) \wedge (\mathtt{cost} = \mathtt{bw} \cdot 0.05) \wedge (\mathtt{cost}' \leqslant 600) \wedge (\mathtt{bw}'' \leqslant 10{\cdot}000)$$
$$\wedge (\mathtt{bw} = \mathtt{bw}') \wedge (\mathtt{bw} = \mathtt{bw}'') \wedge (\mathtt{cost} = \mathtt{cost}')$$

Once the contract is signed, P invokes an internal service (along the endpoint $p_{\mathsf{P}} \bullet o_{reqMetrics}$) to obtain a run-time measurement of the bandwidth effectively supplied to the client. For simplicity sake, P's subservice performing the measurement is not explicitly represented. Then, P fixes the bandwidth, communicates it to client C (along the endpoint $p_{\mathsf{C}} \bullet o_{fix}$), and, by performing some operations `check`, the two parties validate the signed contract with respect to the value fixed by the provider. Afterwards, during the execution, the client service could use again operation `check` in a similar way to verify compliance with the SLA defined at negotiation time, by exploiting run-time data provided by some trustworthy measurement service. Finally, in case contract validation succeeds, C stops all its other instances that are concurrently performing negotiation phases, by means of a kill activity, and notifies the registry that it does not need further query results, by communicating along $p_{\mathsf{R}} \bullet o_{stop}$. Notably, our prioritized semantics guarantees that only one instance of C signs a contract with the provider.

6 Other concurrent constraint programming constructs

In the previous sections, for the sake of presentation, only four operations have been defined to interact with the store. We want now to show that variants of these operations or other concurrent constraint programming constructs can be easily implemented in COWS, to model some peculiar aspects of discovery and negotiation processes.

Non-blocking operations.
The operations `tell` c, `check` c and `ask` c are blocking operations, i.e. if the constraint c is not consistent with/entailed by the current store, the operations, and their continuation, are suspended until the constraint is consistent/entailed. Nevertheless, non-blocking variants of these operations can be defined. For example, by adding the following pattern-matching rule

$$\frac{\neg\, isCons(C \uplus \{c\})}{\mathcal{M}(\langle c^{\neg}, x\rangle, C) = \{x \mapsto C\}}$$

the non-blocking operation `tell` $c\,\{s_1\}\{s_2\}$ – that adds c to the store and continues as s_1, if c is consistent with the store, or otherwise continues as s_2 – can be rendered as follows:

$$\begin{aligned}\langle\!\langle \texttt{tell}\ c\,\{s_1\}\{s_2\}\rangle\!\rangle \;=\; [\hat{n}]\,(\;&\hat{n}!\langle c, c^{\neg}\rangle \\ &\mid [y_1, y_2]\,\hat{n}?\langle y_1, y_2\rangle. \\ &\quad [x]\,(\;p_s \bullet o_{get}?\langle\langle y_1, x\rangle\rangle.(\{\!|\, p_s \bullet o_{set}!\langle x \uplus \{y_1\}\rangle \,|\!\}\mid \langle\!\langle s_1\rangle\!\rangle) \\ &\qquad + \; p_s \bullet o_{get}?\langle\langle y_2, x\rangle\rangle.(\{\!|\, p_s \bullet o_{set}!\langle x\rangle \,|\!\}\mid \langle\!\langle s_2\rangle\!\rangle)\;)\end{aligned}$$

This operation can be used, for example, to model a party of a negotiation that, in case its first-rate constraint is too strong to reach an agreement, weakens the requirements and retries with another constraint. For example, the term

$$\texttt{tell}\ c_{strong}\,\{s_{strongSuccess}\}\{\,\texttt{tell}\ c_{weak}\,\{s_{weakSuccess}\}\{s_{quit}\}\,\}$$

continues as $s_{strongSuccess}$ (resp. $s_{weakSuccess}$) if constraint c_{strong} (resp. c_{weak}) is consistent with the current store; if both attempts fail, it gives up the negotiation and continues as s_{quit}.

Getting the (best) solutions.
During the negotiation phase, one is usually interested in satisfaction or violation of constraints. However, when the involved parties reach an agreement, one could be interested to obtain (one of) the best solution of the resulting multiset of constraints.

To achieve this aim, we introduce a function $getSol(C)$ that takes a constraint multiset C and, if C is consistent, returns a solution. Formally, in case of crisp constraints, $getSol(\{c_1, \ldots, c_n\})$ returns an assignment η such that $c_1\eta \wedge \ldots \wedge c_n\eta \neq$ **false**. Instead, in case of soft constraints, it returns one of the optimal solutions, i.e. an assignment η such that $c_1\eta \times \ldots \times c_n\eta \neq 0$ and $c_1\eta' \times \ldots \times c_n\eta' \leq c_1\eta \times \ldots \times c_n\eta$ for any η'. Like for consistency and entailment predicates, we do not consider here the problem of solving a constraint multiset and refer the interested reader to the literature (see e.g. [34,4,43]).

We also add the following rule to those defining the pattern-matching function:

$$\frac{getSol(C) = \eta \qquad \eta\,|_{\bar{\mathtt{x}}} = \bar{v}}{\mathcal{M}(\langle \bar{\mathtt{x}}, \bar{x}, y\rangle, C) = \{\bar{x} \mapsto \bar{v}, y \mapsto C\}}$$

Here, $\_\,|_{\bar{\mathtt{x}}}$ is a projection function that, given an assignment η, returns the tuple of values associated by η to the constraint variables $\bar{\mathtt{x}}$. Therefore, the construct $\texttt{getSol}(\bar{\mathtt{x}}, \bar{x}).s$ – that gets (one of) the best solution of the current store of constraints, assigns to $\bar{x}$ the values associated to $\bar{\mathtt{x}}$ and continues as s – can be rendered in COWS as follows:

$$\langle\!\langle \texttt{getSol}(\bar{\mathtt{x}}, \bar{x}).s\rangle\!\rangle \;=\; [y]\,p_s \bullet o_{get}?\langle \bar{\mathtt{x}}, \bar{x}, y\rangle.\,(\{\!|\, p_s \bullet o_{set}!\langle y\rangle \,|\!\}\mid \langle\!\langle s\rangle\!\rangle)$$

Notably, if a variable within $\bar{x}$ is replaced before the execution of $\texttt{getSol}(\bar{\mathtt{x}}, \bar{x})$, the pattern-matching rule above cannot be applied and, thus, the operation is stuck forever. However, this unwanted behaviour can be easily prevented by properly delimiting the variables, as in, e.g., the term $[\bar{x}]\,\texttt{getSol}(\bar{\mathtt{x}}, \bar{x}).s$.

We now illustrate the semantics of the operation `getSol` by means of some examples. Suppose that the following crisp constraints are the result of a negotiation

between a client and a provider:

$$c_{client} \;=\; \texttt{cost} \leqslant 600 \qquad\qquad c_{provider} \;=\; \texttt{cost} \geqslant 150$$

Then, any assignment that maps `cost` to a value between 150 and 600 is an effective solution. Thus, in this case, execution of $\texttt{getSol}(\texttt{cost}, x)$ has the effect of substituting x with a value between 150 and 600. As another example, consider the soft constraints

$$c'_{client} \;=\; \lfloor 600/\texttt{cost} \rfloor \qquad\qquad c'_{provider} \;=\; \lfloor \texttt{cost}/150 \rfloor$$

defined over the domain of interpretation [100..800] for the variable `cost` and returning values within the c-semiring $\langle [0..6], max, min, 0, 6\rangle$. Constraints c'_{client} and $c'_{provider}$ associate to each assignment for the variable `cost` an element of the c-semiring, which represents a grade of preference. For example, from the client point of view, the assignment $\{\texttt{cost} \mapsto 500\}$ has grade of preference 1, while the assignment $\{\texttt{cost} \mapsto 300\}$ has grade of preference 2, because (of course) the client prefers to save money. Instead, from the provider point of view, the greater the values of `cost` are the higher the grade of preference is. Moreover, c'_{client} states that values greater than 600 are not acceptable for the client, because the corresponding grade is 0; similarly, $c'_{provider}$ states that values lesser than 150 are not acceptable for the provider. In this case, the operation $\texttt{getSol}(\texttt{cost}, x)$ has the effect of substituting x with one of the best solutions of the constraint system, i.e. an assignment that produces the maximal grade of preference. For instance, the assignment $\{\texttt{cost} \mapsto 300\}$ is one of the best solutions, indeed $min(c'_{client} \cdot \{\texttt{cost} \mapsto 300\}, c'_{provider} \cdot \{\texttt{cost} \mapsto 300\}) = min(2, 2) = 2$ and one can prove that 2 is the highest grade of preference for the combination of the two constraints.

Of course, more complex variants of the operation `getSol` could be implemented, in order to get all the (best) solutions or all the solutions with a grade better than a certain threshold.

7 Concluding remarks and related work

By focussing on QoS requirement specifications and SLA achievements, we have demonstrated that COWS is a suitable formalism for modelling publication, discovery, negotiation, deployment and execution of service-oriented applications. Specifically, we have shown that constraints and operations on them can be smoothly incorporated in COWS, and proposed a disciplined way to model multisets of constraints and manipulate them through appropriate interaction protocols. The novelty of our proposal is that all the above different key aspects of SOAs are dealt with in an homogeneous and direct way by using a single linguistic formalism that already provides a number of analytical tools and techniques (see e.g. [29,37,16]). Proof techniques, such as type systems and observational semantics, for analysing properties of service-oriented applications modelled in COWS are currently under

investigation.

We end by touching upon more strictly related work. Most of the proposals in the literature result from the extension of some well-known process calculus with constructs to describe QoS requirements. This is, for example, the case of cc-pi [9], a calculus that generalises the explicit name 'fusions' of the pi-F calculus [44] to 'named constraints', namely constraints defined on enriched c-semiring structures. Rather than on fusions of names, COWS relies on substitutions of variables with values and can thus express also soft constraints by exploiting the simpler notion of c-semiring. Moreover, COWS permits defining local stores of constraints while cc-pi processes necessarily share one global store. A similar approach to SLAs negotiation is proposed in [2], although it is based on fuzzy sets instead of constraints and relies on three different languages, one for client requests, one for provider descriptions and one for contracts creation and revocation. SLA compliance has been also the focus of $\mathcal{K}$oS [13] and Kaos [14], two calculi designed for modelling network aware applications with located services and mobility. In both cases, QoS parameters are associated to connections and nodes of nets, and operations have a QoS value; the operational semantics ensures that systems evolve according to SLAs. All the mentioned proposals aim at specifying and concluding SLAs, while COWS permits also modelling other service-oriented aspects, such as e.g. service publication, discovery and orchestration, fault and compensation handling, service instances and interactions.

Integrations of the concurrent constraint paradigm with process calculi have also been used to define foundational formalisms for computer music languages. This is the case of the π^+-calculus [15], an extension of the (polyadic) π-calculus with constraint agents that can interact with a store of constraints by performing 'tell' and 'ask' actions. Differently from COWS, the store of constraints is not a term of the calculus, indeed the operational semantics of π^+-calculus is defined over configurations consisting of pairs of an agent and a store, and local stores are not supported.

A different approach to QoS is adopted in [37], where a stochastic extension of COWS is presented to enable quantitative reasoning about service behaviours. Specifically, COWS syntax and semantics are enriched along the lines of Markovian extensions of process calculi [20], and then probabilistic verification is carried on by using the PRISM probabilistic model checker.

There are also some other works that, differently from COWS, exploit static service discovery mechanisms. For example, [3] introduces λ^{req}, an extension of the λ-calculus with primitive constructs for call-by-contract invocation. In particular, an automatic machinery, based on a type system and a model-checking technique, constructs a viable plan for the execution of services belonging to a given orchestration. Non-functional aspects are also included and enforced by means of a runtime security monitor. In [30], user's requests and compositions of web services are statically modelled via constraints. Finally, the calculi of contracts of [8] represent a more abstract approach for statically checking compliance between the client requirements and the service functionalities. A contract defines the possible flows of

interactions of a service, but does not takes into account non-functional properties and, thus, cannot be used for specifying and negotiating SLAs.

Acknowledgement

We thank the anonymous referees for their useful comments.

References

[1] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Grid Resource Allocation Agreement Protocol (GRAAP) WG, 2007. Available at http://www.ogf.org.

[2] D. Bacciu, A. Botta, and H. Melgratti. A fuzzy approach for negotiating quality of services. In *TGC*, volume 4661 of *LNCS*, pages 200–217. Springer, 2006.

[3] M. Bartoletti, P. Degano, and G. Ferrari. Security Issues in Service Composition. In *Proceedings of FMOODS 2006, 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *Lecture Notes in Computer Science*, pages 1–16, 2006.

[4] S. Bistarelli. *Semirings for Soft Constraint Solving and Programming.* LNCS 2962. Springer, Berlin, 2004.

[5] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[6] S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Log.*, 7(3):563–589, 2006.

[7] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. SCC: a Service Centered Calculus. In *WS-FM*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

[8] M. Bravetti and G. Zavattaro. Contract based multi-party service composition. In *FSEN*, volume 4767 of *LNCS*, pages 207–222. Springer, 2007.

[9] M. Buscemi and U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *ESOP*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.

[10] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *COORDINATION*, volume 4038 of *LNCS*, pages 63–81. Springer, 2006.

[11] M.J. Butler, C.A.R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, volume 3525 of *LNCS*, pages 133–150. Springer, 2005.

[12] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[13] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A Formal Basis for Reasoning on Programmable QoS. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 436–479. Springer, 2003.

[14] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A Process Calculus for QoS-Aware Applications. In *COORDINATION*, volume 3454 of *LNCS*, pages 33–48. Springer, 2005.

[15] J.F. Díaz, C. Rueda, and F.D. Valencia. π^+-calculus: A calculus for concurrent processes with constraints. *CLEI Electron. J.*, 1(2), 1998.

[16] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. Technical report, 2007. Available at http://rap.dsi.unifi.it/cows/.

[17] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *POPL '96*, pages 372–385. ACM Press, 1996.

[18] E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artif. Intell.*, 58(1-3):21–70, 1992.

[19] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[20] S.T. Gilmore and M. Tribastone. Evaluating the scalability of a web service-based distributed e-learning and course management system. In *WS-FM*, volume 4184 of *LNCS*, pages 214–226. Springer, 2006.

[21] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *ICSOC*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.

[22] P.C.K. Hung, H. Li, and J. Jeng. WS-Negotiation: an overview of research issues. In *HICSS*, volume 01. IEEE Computer Society, 2004.

[23] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management, Special Issue on E-Business Management*, 11(1), 2003.

[24] C. Laneve and L. Padovani. Smooth orchestrators. In *FoSSaCS*, volume 3921 of *LNCS*, pages 32–46. Springer, 2006.

[25] C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.

[26] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2006. Available at http://rap.dsi.unifi.it/cows.

[27] A. Lapadula, R. Pugliese, and F. Tiezzi. A WSDL-based type system for WS-BPEL. In *COORDINATION*, volume 4038 of *LNCS*, pages 145–163. Springer, 2006.

[28] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[29] A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN*, volume 4767 of *LNCS*, pages 223–239. Springer, 2007.

[30] A. Lazovik, M. Aiello, and R. Gennari. Encoding requests to web service compositions as constraints. In *CP*, volume 3709 of *LNCS*, pages 782–786. Springer, 2005.

[31] H. Ludwig, A. Keller, A. Dan, R.P. King, and R. Franck. WSLA language specification version 1.0. Technical report, IBM Corporation, 2003. Available at http://www.research.ibm.com/wsla.

[32] L.G. Meredith and S. Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.

[33] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.

[34] U. Montanari and F. Rossi. Constraint relaxation may be perfect. *Artif. Intell.*, 48(2):143–170, 1991.

[35] U. Nestmann and B.C. Pierce. Decoding choice encodings. In *CONCUR*, volume 1119, pages 179–194. Springer, 1996.

[36] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007.
Available at http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[37] D. Prandi and P. Quaglia. Stochastic COWS. In *ICSOC*, volume 4749 of *LNCS*, pages 245–256. Springer, 2007.

[38] V.A. Saraswat and M.C. Rinard. Concurrent constraint programming. In *POPL*, pages 232–245. ACM Press, 1990.

[39] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *POPL*, pages 333–352, 1991.

[40] N. Srinivasan, M. Paolucci, and K. Sycara. Semantic web service discovery in the OWL-S IDE. In *HICSS*, volume 6. IEEE Computer Society, 2006.

[41] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic web services. *J. Web Sem.*, 1(1):27–46, 2003.

[42] F. van Breugel and M. Koshkina. Models and verification of BPEL. Technical report, DRAFT, 2006. Available at http://www.cse.yorku.ca/∼franck/research/drafts/tutorial.pdf.

[43] M. Wirsing, G. Denker, C. Talcott, A. Poggio, and L. Briesemeister. A rewriting logic framework for soft constraints. In *WRLA 2006, 6th International Workshop on Rewriting Logic and its Applications*, April 2006. To appear in ENTCS, 2006.

[44] L. Wischik and P. Gardner. Explicit fusions. *Theor. Comput. Sci.*, 340(3):606–630, 2005.