Stochastic Concurrent Constraint Programming

Luca Bortolussi

Department of Mathematics and Computer Science
University of Udine, Italia

Abstract

We present a stochastic version of Concurrent Constraint Programming (CCP), where we associate a rate to each basic instruction that interacts with the constraint store. We give an operational semantic that can be provided either with a discrete or a continuous model of time. The notion of observables is discussed, both for the discrete and the continuous version, and a connection between the two is given.

Key words: Concurrent Constraint Programming, Stochastic Languages, Probabilistic Semantics, Continuous Time.

1 Introduction

Concurrent Constraint Programming [20] (CCP) is a language based on the idea of "store as constraint". It has the syntax of a simple process algebra, but with the power of working over a constraint system, a form of memory where constraints on the variables into play are stored and information is refined monotonically. In its fifteen years of life, it has been extended in different directions: with a timed semantic [21], in the distributed setting [13], and with a (discrete time) probabilistic semantic [9,8], just to cite a few. Its fame can be related both to its logical simplicity, allowing to reason about concurrency, and to the power of the constraint systems, enabling to model complex programs with a reasonable effort.

In this work we extend [9], by giving a different probabilistic syntax and semantics that can model both discrete and continuous time, allowing more refined forms of quantitative analysis of CCP programs.

The basic idea of this extension is to assign a rate to each instruction that interacts with the constraint store, i.e. ask, tell and the procedure call. The stochastic behaviour is obtained by a race condition between the different agents of this form that are enabled in a particular instant. The fastest process is executed and a new

[1] luca.bortolussi@dimi.uniud.it

race is started ex novo. In this way we give a continuous time semantics, in the same way as stochastic π-calculus [17], PEPA [15], EMPA [1] or pKLAIM [7,6]. On the other hand, we can think at rates as weights or priorities given to the instructions, and so we can normalize the rates of the active agents, and associate a probability of execution to each of them, obtaining a discrete probabilistic computation. Therefore, the same syntax can be used to model both continuous and discrete time. To capture properly those two different behaviours, we define a concept of abstract trace, that is then concretized with the chosen model of time.

There are some technical problems with our approach, in particular related to the hiding operator: we do not associate any rate to it, hence it has to be "removed" before starting any race condition. This is tackled by keeping track in the configuration of the set of local variables, and by replacing hidden variables with fresh free ones.

Comparing the discrete-time semantic presented here with the one developed in [9], we remark that the main difference resides in the way probabilities are assigned. In [9], probability distributions are given to each summation and parallel composition, forcing to define these operators and their semantic parametrically w.r.t. the number of processes involved, thus losing some of their algebraic properties (i.e. associativity). Assigning priorities to instructions, instead, set us free from this limitation (see Section 4). Moreover, we can have both a discrete and a continuous model of time in the same framework.

The introduction of this new semantic has several motivations. First of all one, can apply the broad range of techniques developed in performance analysis, in order to extract information from a program. Secondly, the modeling capabilities are extended, though keeping untouched the simplicity of the syntax. In particular, this stochastic version of CCP, with small extensions, can be used to model biological systems, like metabolical pathways, signaling cascades, gene regulatory networks and so on, almost in the same way as stochastic π-calculus [17] is used for this task [18]. The basic idea is to identify biological entities with processes, and model their interaction as a form of communication. Stochastic π-calculus has only agents communicating synchronously, hence all biological actors and their interactions must be described explicitly. On the other hand, CCP offers a different form of communication, proceeding asynchronously via the store, which can be seen as a form of memory endowed with computational capabilities. This feature allows a simplification in the modeling process, as part of the interactions can be described at the store level. Note that we can think at the constraint store as a description of the environment, populated by biological actors, i.e. processes interacting with it by means of ask and tell instructions. [2]

The paper is organized as follows. In Section 2 some preliminary concepts are recalled. In Section 3 the syntax of the language is discussed, together with the machinery for dealing with hiding removal. Section 4 presents the operational semantics, while the models of time, and the concept of observables are put forward

[2] Note that synchronous communication can be easily integrated into CCP following, for instance, [2]. This addition would extend even more the modeling capabilities.

in Section 5. Finally, in Section 6 we give an example, in Section 7 we discuss some related work and in Section 8 we draw some conclusions.

2 Preliminaries

Constraint systems are one of the basic ingredients of CCP, and they are defined (cf. [5] or [22]) as complete algebraic lattices, where the ordering $\sqsubseteq$ is given by the inverse of the entailment relation $\vdash$. Usually, such a constraint system is derived from a first-order language together with an interpretation, where constraints are formulas of the language, and a constraint c entails a constraint d, $c \vdash d$, if every valuation satisfying c satisfies also d. In this case we write $d \sqsubseteq c$. Clearly, in every real implementation the predicate $\vdash$ must be decidable. In addition, to model hiding and parameter passing, the previous lattice is enriched with a cylindric algebraic structure (cf. [14]), i.e. with cylindrification operators and diagonal elements.

Formally, a constraint system $\mathcal{C} = (Con, Con_0, \mathcal{V}, \sqsubseteq, \sqcup, \top, \bot, \exists_x, d_{xy})$ is a complete algebraic lattice where Con is the set of constraints, ordered by $\sqsubseteq$, Con_0 is the set of finite elements, $\sqcup$ is the least upper bound (lub) operation, $\mathcal{V}$ is the set of variables, $\top$ and $\bot$ are respectively the bottom and the top element, $\{\exists_x \mid x \in \mathcal{V}\}$ are the cylindrification operators and $\{d_{xy} \mid x, y \in \mathcal{V}\}$ are the diagonal elements.

An important instrument while working with constraint systems are the substitutions of variables in a constraint, allowing to model hiding and recursive call in a simple way. We can think of a substitution as a mapping $f : \mathcal{V} \to \mathcal{V}$, with $|\{x \in \mathcal{V} \mid x \neq f(x)\}| < \infty$. Formally, given a constraint c, and two vectors of variables $\mathbf{x} \subset fv(c)$ and $\mathbf{y}$, we define the substitution of $\mathbf{x}$ with $\mathbf{y}$ in c as the constraint $c[\mathbf{x}/\mathbf{y}] = \exists_\alpha(\mathbf{y} = \alpha \sqcup \exists_{\mathbf{x}}(\alpha = \mathbf{x} \sqcup c))$, with $\alpha \cap (fv(c) \cup \mathbf{y} \cup \mathbf{x}) = \emptyset$ and $|\alpha| = |\mathbf{x}| = |\mathbf{y}|$. Substitutions satisfy some basic properties, cf. [11] for a detailed review. In particular, if f is a renaming (bijective mapping), then $\exists_{f(\mathbf{x})} c[f] = (\exists_{\mathbf{x}} c)[f]$, i.e. bounded variables can always be renamed.

3 Syntax

The syntax of the calculus is given in Table 1. The instructions are basically the ones of CCP, with the introduction of the *rates* λ. In particular, rates are added to three instructions, i.e. ask, tell and the recursive call. A rate is simply a real positive number and it can be interpreted in two different ways. We can consider it either as a priority, inducing discrete probability distributions (through normalization), or as a rate representing the "speed" associated to the corresponding instruction, where the higher the rate, the higher the speed. Of course, in this second case we have in mind mainly a continuous model of time, so those rates will be assigned to an exponential distribution, and therefore they determine the time employed by an instruction to be executed. In particular, we can think of a rate as the expected number of times an instruction will be executed in an unit of time.
The rationale behind gluing rates to ask and tell operations is, in the continuous time case, that those instructions operate on the constraint store of the system,

$$Program = Decl.A$$

$$D = \varepsilon \mid Decl.Decl \mid [p(\mathbf{x})]_\lambda :- A$$

$$A = \mathbf{0} \mid \mathrm{tell}_\lambda(c).A \mid \mathrm{ask}_\lambda(c).A \mid [p(\mathbf{x})]_\lambda \mid \exists_x A \mid (A_1 + A_2) \mid (A_1 \parallel A_2)$$

Table 1
Syntax of stochastic CCP

hence they effectively require a certain amount of time to be executed. This time may depend on the complexity of the constraint to be told, or on the complexity of the effective computation of the entailment relation. Moreover we can think that there is a time needed by the system to compute the least upper bound operation in our constraint system, as this corresponds to a form of constraint propagation. Note that there is a rate associated also to procedure calls. This may be questionable, but in fact there is a cost associated also to such a operation: the process must access to the repository where declarations are stored, and it must replace the free variables with the specified ones. Moreover, this assumption is needed to avoid infinite branching in the transition system, cf. below. This two-fold interpretation of rates finds its motivation in the definition of a common framework where studying the relationship between discrete and continuous models of time.

The only canonical instruction that does not have a rate is the hiding operator. We really think that the declaration of a local variable is really a too simple operation to assign a rate to it. However, we want to have a semantics where each transition has some probabilistic information attached to it, so not giving rates to hiding implies that we need a mechanism to hide variables outside the transition system. Essentially, we obtain this by replacing an hidden variable with a fresh one, and guaranteeing that no other agent can access to it. This mechanism is presented in next subsection.

Finally, we do not require that the choice operator is always guarded by ask instructions. This is because we always have as factors agents competing through a race condition (determined by their rate), and moreover we can think that non-guarded agents are preceded by an $\mathrm{ask}_\lambda(\top)$ instruction, which is always enabled.

In Table 2 we define a congruence relation between agents. The first three rules simply state that the parallel operator is a commutative monoid in the space of agents, while the next three rules imply the same property for the choice operator. The last three rules deal with some basic properties of the hiding operator. We stress that the request of having all executable processes encapsulated by a rate is

$(CR1)$	$A_1 \parallel (A_2 \parallel A_3) \equiv (A_1 \parallel A_2) \parallel A_3$	
$(CR2)$	$A_1 \parallel A_2 \equiv A_2 \parallel A_1$	
$(CR3)$	$A_1 \parallel \mathbf{0} \equiv A_1$	
$(CR4)$	$A_1 + (A_2 + A_3) \equiv (A_1 + A_2) + A_3$	
$(CR5)$	$A_1 + A_2 \equiv A_2 + A_1$	
$(CR6)$	$A_1 + \mathbf{0} \equiv A_1$	
$(CR7)$	$\exists_x \exists_y A \equiv \exists_y \exists_x A$	
$(CR8)$	$\exists_x A \equiv \exists_y A[y/x]$	if y is not free in A
$(CR9)$	$\exists_x A \equiv A$	if x is not free in A

Table 2
Congruence Relation for stochastic CCP

crucial for the semantics, cf. Section 4 for more details.

3.1 Configurations and Removal of Hiding

Transitions of the system are determined by the agent to be executed and the current configuration of the store, thus the configuration space should be $\mathcal{P} \times \mathcal{C}$, where $\mathcal{P}$ is the space of processes (modulo the congruence relation of Table 2), and $\mathcal{C}$ is the constraint system. However, the mechanism to remove hiding requires the introduction of another term in the product above.

First, we split the set of variables $\mathcal{V}$ into two infinite disjoint subsets $\mathcal{V}_1$ and $\mathcal{V}_2$, $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$ and $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$. Then we ask that all the variables used in the definition of agents are taken from $\mathcal{V}_1$, so that all variables in $\mathcal{V}_2$ are fresh before starting a computation. Intuitively, we will use variables taken from $\mathcal{V}_2$ to replace hidden variables in an agent. In this way we can avoid name clashes with free variables or with other hidden variables. However, we need also a way to remember which variables of $\mathcal{V}_2$ we used. The simplest way is to carry this information directly in the configuration of the system. Therefore, if we denote with $\wp_f(\mathcal{V}_2) = \overline{\mathcal{V}}$ the collections of finite subsets of $\mathcal{V}_2$, then a configuration of the system will be a point in the space $\mathcal{P} \times \mathcal{C} \times \overline{\mathcal{V}}$, indicated by $\langle A, d, V \rangle$. This idea of adding the set of local variables to system configurations is borrowed from [11].

It's clear that it is not really important how a local variable is called, as long as it is used just by the process that has defined it. This justifies the introduction of the following definition:

Definition 1 *Let* $\langle A_1, d_1, V_1 \rangle, \langle A_2, d_2, V_2 \rangle \in \mathcal{P} \times \mathcal{C} \times \overline{\mathcal{V}}$. $\langle A_1, d_1, V_1 \rangle \equiv_r \langle A_2, d_2, V_2 \rangle$ *if and only if there exists a renaming* $f : \mathcal{V}_2 \to \mathcal{V}_2$ *such that* $f(V_2) = V_1$, $d_1[f] = d_2$ *and* $A_2[f] \equiv A_1$.

It is simple to see that $\equiv_r$ is a congruence relation (r stays here for renaming).

The mechanism of hiding removal will be achieved through a function $\mu : \mathcal{P} \times \mathcal{C} \times \overline{\mathcal{V}} \rightarrow \mathcal{P} \times \mathcal{C} \times \overline{\mathcal{V}} / \equiv_r$, mapping configurations with into ones where no *active* agent is preceded by an hiding operator. Definition of μ is given by structural induction on the agents. To simplify the notation, in the following we denote $\mu(\langle A, d, V\rangle)$ by $\langle A, d, V\rangle_\mu$ In addition, if $\langle A, d, V\rangle_\mu = \langle A', d, V'\rangle$, we set $\langle A, d, V\rangle_{\mu_1} = A'$ and $\langle A, d, V\rangle_{\mu_2} = V'$.

Definition 2 *The function $\mu : \mathcal{P} \times \mathcal{C} \times \overline{\mathcal{V}} \rightarrow \mathcal{P} \times \mathcal{C} \times \overline{\mathcal{V}}$ is defined inductively by:*

(i) $\langle \mathbf{0}, d, V\rangle_\mu = \langle \mathbf{0}, d, V\rangle$;

(ii) $\langle \mathrm{tell}_\lambda(c).A, d, V\rangle_\mu = \langle \mathrm{tell}_\lambda(c).A, d, V\rangle$;

(iii) $\langle \mathrm{ask}_\lambda(c).A, d, V\rangle_\mu = \langle \mathrm{ask}_\lambda(c).A, d, V\rangle$;

(iv) $\langle [p(\mathbf{y})]_\lambda, d, V\rangle_\mu = \langle [p(\mathbf{y})]_\lambda(c).A, d, V\rangle$;

(v) $\langle \exists_x A, d, V\rangle_\mu = \langle A[x/y], d, V \cup \{y\}\rangle_\mu$, *with $y \in \mathcal{V}_2 \setminus V$, if x is free in A;*

(vi) $\langle \exists_x A, d, V\rangle_\mu = \langle A, d, V\rangle_\mu$, *if x is not free in A;*

(vii) $\langle A_1 \parallel A_2, d, V\rangle_\mu =$
$$\left\langle \langle A_1, d, V\rangle_{\mu_1} \parallel \left\langle A_2, d, \langle A_1, d, V\rangle_{\mu_2}\right\rangle_{\mu_1}, d, \left\langle A_2, d, \langle A_1, d, V\rangle_{\mu_2}\right\rangle_{\mu_2}\right\rangle;$$

(viii) $\langle A_1 + A_2, d, V\rangle_\mu =$
$$\left\langle \langle A_1, d, V\rangle_{\mu_1} + \left\langle A_2, d, \langle A_1, d, V\rangle_{\mu_2}\right\rangle_{\mu_1}, d, \left\langle A_2, d, \langle A_1, d, V\rangle_{\mu_2}\right\rangle_{\mu_2}\right\rangle.$$

The above definition is a little bit hard to parse. Its idea, however, is simple: an agent starting with tell, ask, **0** or a recursive call is left untouched, while whenever there is an hiding operator, we substitute a fresh variable from $\mathcal{V}_2$ to the hidden one, and then we recursively operate the remaining agent. Rule *(vi)* deals with the case in which the hidden operator is in excess, and it can be removed by rule (CR9). This rule is not implied by *(v)*, as in this case we do not want to add new variables in V. Finally, the two rules for dealing with parallel and summation constructs simply state that μ is applied recursively first on the left agent and then on the right agent. This is necessary because we must not use the same name for local variables in the left and the right processes, so we need to have the information about local variables of the left process while removing hidings in the right (or viceversa).

As we are not interested in the actual name given to local variables, the correct space of configurations to use is $\mathfrak{C} = \mathcal{P} \times \mathcal{C} \times \overline{V} / \equiv_r$. μ con be safely lifted to this new space.

Proposition 3.1 *The function $\mu : \left(\mathcal{P} \times \mathcal{C} \times \overline{\mathcal{V}}\right) / \equiv_r \rightarrow \left(\mathcal{P} \times \mathcal{C} \times \overline{\mathcal{V}}\right) / \equiv_r$, where $\mu([\langle A, d, V\rangle]) = [\mu(\langle A, d, V\rangle)]$, is well defined.* ■

As a final remark, we observe that we cannot use a similar mechanism to get rid of recursion, as an automatic application of procedure call may generate an infinite degree of parallelism. For instance, consider $p : -\mathrm{tell}_\lambda(c) \parallel p$; if we solve automatically the recursion, we would get an infinite number of copies of tell.

This mechanism for the elimination of hiding operator can be seen as an instantaneous transition, which in the language of stochastic modeling means an instruction with an infinite rate associated to it. If we had modeled hiding in this way, we could have simplified the configurations of the system, by dropping the information about local variables. On the other hand, the introduction of infinite rates generates a non-conventional Continuous Time Markov Chain, where the classical analysis techniques cannot be used safely. This fact requires to deal differently with normal and infinite rates, complicating the definition of the operational semantics. This is, for instance, what happens with EMPA [1].

4 Operational Semantic

The operational semantics of the language is given in the Structural Operational Semantics style by a labeled transition system, corresponding to a labeled relation $\longrightarrow$. Formally, if the space of configurations is $\mathfrak{C} = \left(\mathcal{P} \times \mathcal{C} \times \overline{\mathcal{V}}\right) / \equiv_r$, then $\longrightarrow \subseteq \mathfrak{C} \times [0,1] \times \mathbb{R}^+ \times \mathfrak{C}$. Thus the relation is labeled by two real numbers, one between 0 and 1 and the other one positive. Intuitively, the first number corresponds to the probability associated to the particular transition, while the second term is the rate at which the transition happens. Concretely, these two numbers are what we need to define the probabilistic model of the underlying graph, i.e. the Markov Chain associated to the program execution, be it discrete or continuous (cf. Section 5).

The transition is defined in Table 3. First, we give a formal definition of the function $\mathrm{rate} : \mathcal{P} \times \mathcal{C} \to \mathbb{R}$, assigning to each agent its global rate. This global rate is nothing but the sum of the rates of all the active subagents of the considered agent.

Definition 3 *The function* $\mathrm{rate} : \mathcal{P} \times \mathcal{C} \to \mathbb{R}$ *is defined by*

(i) $\mathrm{rate}\left(\mathrm{tell}_\lambda(c).A, d\right) = \lambda$;

(ii) $\mathrm{rate}\left(\mathbf{0}, d\right) = 0$;

(iii) $\mathrm{rate}\left(\mathrm{ask}_\lambda(c).A, d\right) = \lambda$ *if* $d \vdash c$;

(iv) $\mathrm{rate}\left(\mathrm{ask}_\lambda(c).A, d\right) = 0$ *if* $d \nvdash c$;

(v) $\mathrm{rate}\left([p(\vec{y})]_\lambda, d\right) = \lambda$;

(vi) $\mathrm{rate}\left(A_1 \parallel A_2, d\right) = \mathrm{rate}\left(A_1, d\right) + \mathrm{rate}\left(A_2, d\right)$;

(vii) $\mathrm{rate}\left(A_1 + A_2, d\right) = \mathrm{rate}\left(A_1, d\right) + \mathrm{rate}\left(A_2, d\right)$.

Now we give some comments on the rules of Table 3. The first observation is that after each transition, the function μ defined in Section 3.1 is applied to the obtained configuration. This guarantees that the agent ready to be executed in the next step has all active hidings resolved.
Rule (SR1) states that the tell instruction add its argument to the constraint store, and this happens with probability one and at rate λ, i.e. the one specified in the tell instruction itself. Rule (SR2) works similarly for the ask instruction, relatively to rate and probability, provided that the current store entails the asked constraint. Rule (SR3) implements the recursive call by using a simple substitution; the def-

$$(SR1) \qquad \langle \mathrm{tell}_\lambda(c).A, d, V\rangle \longrightarrow_{(1,\lambda)} \langle A, d \sqcup c, V\rangle_\mu$$

$$(SR2) \qquad \langle \mathrm{ask}_\lambda(c), d, V\rangle \longrightarrow_{(1,\lambda)} \langle A, d, V\rangle_\mu \qquad \text{if } d \vdash c$$

$$(SR3) \qquad \langle [p(\mathbf{y})]_\lambda, d, V\rangle \longrightarrow_{1,\lambda} \langle A[\mathbf{x}/\mathbf{y}], d, V\rangle_\mu \qquad \text{if } [p(\mathbf{y})]_\lambda :- A$$

$$(SR4) \qquad \frac{\langle A_1, d, V\rangle \longrightarrow_{(p,\lambda)} \langle A_1', d', V\rangle_\mu}{\langle A_1 + A_2, d, V\rangle \longrightarrow_{(p',\lambda')} \langle A_1', d', V\rangle_\mu}$$
$$\text{with } p' = \tfrac{p\lambda}{\lambda + \mathrm{rate}(A_2,d)} \text{ and } \lambda' = \lambda + \mathrm{rate}(A_2, d)$$

$$(SR5) \qquad \frac{\langle A_1, d, V\rangle \longrightarrow_{(p,\lambda)} \langle A_1', d', V\rangle_\mu}{\langle A_1 \parallel A_2, d, V\rangle \longrightarrow_{(p',\lambda')} \langle A_1' \parallel A_2, d', V\rangle_\mu}$$
$$\text{with } p' = \tfrac{p\lambda}{\lambda + \mathrm{rate}(A_2,d)} \text{ and } \lambda' = \lambda + \mathrm{rate}(A_2, d)$$

Table 3
Operational Semantics of stochastic CCP

inition of Section 2 guarantees that variables are connected correctly (in the same way done by the Δ operator, cf. [5]). Rules (SR4) and (SR5) are similar, and they implement the race condition mechanism, or the probabilistic choice. They state that, if a single term of the sum or of the parallel composition can evolve with a certain probability and a certain rate, the whole construct can evolve with a new probability and a new rate given by the expressions of Table 3. Note that we need just the rules for the left agents, as the corresponding ones for right agents can be derived from the commutativity of $+$ and $\parallel$. Intuitively, what happens is a competition between the tell, ask, and procedure call instructions that are executable by the current configuration. We can show that the rate of execution is the sum over all rates of executable agents, and that the probability of execution is the rate of the executed agent divided by the global rate.
Formally, we can define the executable agents for a given configuration as

Definition 4 *Let $\langle A, d, V\rangle \in \mathfrak{C}$. The set of executable instructions of $\langle A, d, V\rangle$, denoted by $\mathrm{exec}(\langle A, d, V\rangle)$, is defined inductively by:*

(i) $\mathrm{exec}(\langle \mathrm{tell}_\lambda(c).A, d, V\rangle) = \{\mathrm{tell}_\lambda(c)\}$;

(ii) $\mathrm{exec}(\langle \mathrm{ask}_\lambda(c).A, d, V\rangle) = \{\mathrm{ask}_\lambda(c)\}$ *if* $d \vdash c$;

(iii) $\mathrm{exec}(\langle \mathrm{ask}_\lambda(c).A, d, V\rangle) = \emptyset$ *if* $d \not\vdash c$*;*

(iv) $\mathrm{exec}(\langle [p(\vec{y})]_\lambda, d, V\rangle) = \{[p(\vec{y})]_\lambda\}$*;*

(v) $\mathrm{exec}(\langle \mathbf{0}, d, V\rangle) = \emptyset$*;*

(vi) $\mathrm{exec}(\langle A_1 \parallel A_2, d, V\rangle) = \mathrm{exec}(\langle A_1, d, V\rangle) \cup \mathrm{exec}(\langle A_2, d, V\rangle)$*;*

(vii) $\mathrm{exec}(\langle A_1 + A_2, d, V\rangle) = \mathrm{exec}(\langle A_1, d, V\rangle) \cup \mathrm{exec}(\langle A_2, d, V\rangle)$*;*

Equipped with this definition, the following proposition can be easily shown:

Proposition 4.1 *Let* $\langle A, d, V\rangle \in \mathfrak{C}$ *be the current configuration. Then the next transition executes one of the agents belonging to the set* $\mathrm{exec}(\langle A, d, V\rangle)$*, call it* $\overline{A}$*. If* $\mathrm{rate}\left(\mathrm{exec}(\langle A, d, V\rangle)\right) = \sum_{A \in \mathrm{exec}(\langle A,d,V\rangle)} \mathrm{rate}(A)$*, then the probability of the transition is* $\mathrm{rate}(\overline{A})/\mathrm{rate}\left(\mathrm{exec}(\langle A, d, V\rangle)\right)$*, and the rate associated to the transition is* $\mathrm{rate}\left(\mathrm{exec}(\langle A, d, V\rangle)\right)$*.* ■

Comparing this semantic with the one of the probabilistic version of CCP presented in [9,10], we can observe two things. First, their definitions of the parallel and the choice operators are parametric w.r.t. the number of processes involved. Secondly, they assign probability distributions to those operators, while we assign rates to basic agents, and derive the probability distributions by combining together those rates compositionally. This different approach allows to maintain the associativity of both sum and parallel composition. On the other hand, we do not require that the processes appearing in the choice are all guarded, implying that we can have a nested alternation of $\parallel$ and $+$. However, this is not a problem, thanks to the compositional nature of the semantics and the finiteness of the set $\mathrm{exec}(\langle A, d, V\rangle)$, for each reachable configuration (this last fact implies that the rate and the probability of a transition are always well defined).

5 Traces and Observables

The operational semantic defined in Table 3 is abstract w.r.t. the notion of time involved. In fact, in the labels we carry sufficient information to construct both a discrete time and a continuous time model of the language, by "concretizing" in two different ways the traces. This corresponds to the two different interpretation of rates, either as priorities or as frequencies of execution per unit of time. An example of this double interpretation is demanded to Section 6. To formalize the above sentence, we first need to introduce the notion of computational trace.

Definition 5 *A* trace *of length* n *is a sequence of* n *consecutive transitions, of the form* $\tau = \langle A_1, d_1, V_1\rangle \longrightarrow_{(p_1,\lambda_1)} \ldots \longrightarrow_{(p_n,\lambda_n)} \langle A_{n+1}, d_{n+1}, V_{n+1}\rangle$.

With $\mathcal{T}_n$ we denote the set of all traces of length n, while $\mathcal{T} = \bigcup_{n=0}^{\infty} \mathcal{T}_n$ is the set of all possible traces. Note that $\mathcal{T}_0 = \mathfrak{C}$. Finally, with $\mathcal{T}\left(\langle A_1, d_1, V_1\rangle, \langle A_2, d_2, V_2\rangle\right)$ we indicate the set of all traces, of any length, leading from $\langle A_1, d_1, V_1\rangle$ to $\langle A_2, d_2, V_2\rangle$, while the set of all traces from the starting state $\langle A_1, d_1, \emptyset\rangle$ to a state $\langle A_2, d_2\rangle$ is $\mathcal{T}\left(\langle A_1, d_1\rangle, \langle A_2, d_2\rangle\right) = \bigcup_{V \subseteq \wp_f(\mathcal{V}_2)} \mathcal{T}\left(\langle A_1, d_1, \emptyset\rangle_\mu, \langle A_2, d_2, V\rangle_\mu\right)$.

5.1 Discrete Time

First we deal with the discrete time concretization of the language. Basically, we discard all the information relative to the rates, and keep only the probability associated to transitions. Note that Proposition 4.1 guarantees that the probabilities of all the transitions exiting from a node of the transition graph sum up to one. So this discrete time realization induces a Discrete Time Markov Chain (DTMC) on the transition graph. We want to stress that a single agent, a tell construct say, does not have an absolute probability assigned. It only have a rate representing its propensity or priority of being executed, while the actual probability of being chosen depends on the context in which it is inserted. The broader this context, the smaller this probability.

In the following we define discrete time traces and observables, that correspond here to the input output behaviour (cf. [9]). To simplify the notation, we indicate with σ an element $\langle A, d, V\rangle \in \mathfrak{C}$.

Definition 6

- *A* discrete time trace *is a sequence* $\sigma_1 p_1 \sigma_2 p_2 \sigma_3 \dots p_n \sigma_{n+1}$*, and the set of all discrete time traces of variable length is* $\mathcal{DT}$.
- *The* discrete time concretization δ *is a function* $\delta : \mathcal{T} \to \mathcal{DT}$*. If* $\tau \in \mathcal{T}$ *is* $\tau = \sigma_1 \longrightarrow_{(p_1,\lambda_1)} \dots \longrightarrow_{(p_n,\lambda_n)} \sigma_{n+1}$*, then* $\delta(\tau) = \delta_\tau = \sigma_1 p_1 \dots p_n \sigma_{n+1}$.
- *If* $\delta_\tau = \sigma_1 p_1 \dots p_n \sigma_{n+1}$*, then* $\mathrm{length}\,(\delta_\tau) = n$ *and* $\mathrm{Prob}(\delta_\tau) = \prod_{i=1}^{\mathrm{length}(\delta_\tau)} p_i$.

To define the I/O observables, we have first to define what is the probability of going from the state σ_I to the state σ_O, i.e. the probability of the transitive closure of the transition relation, indicated by $\mathrm{Prob}(\sigma_I \longrightarrow \sigma_O)$:

$$\mathrm{Prob}(\sigma_I \longrightarrow \sigma_O) = \sum \{p \mid p = \mathrm{Prob}(\delta_\tau),\ \delta_\tau = \sigma_I p_1 \sigma_2 \dots \sigma_n p_n \sigma_O\}\,.$$

Now we have to get rid of the information about the sets of local variables:

$$\mathrm{Prob}\left(\langle A, d\rangle \longrightarrow \langle A', d'\rangle\right) = \sum_{|V|=0}^{\infty} \mathrm{Prob}(\langle A, d, \emptyset\rangle_\mu \longrightarrow \langle A', d', V\rangle_\mu).$$

Note that we do not distinguish between V sets of the same cardinality, as they are all equivalent modulo renaming.

Definition 7 *The discrete time I/O observables is a (sub)distribution of probability on the constraint store* $\mathcal{C}$*, given by:*

$$\mathcal{O}_d\left(\langle A, d\rangle\right) = \{(d', p) \mid p = \mathrm{Prob}\left(\langle A, d\rangle \longrightarrow \langle \mathbf{0}', d'\rangle\right)\}\,.$$

The previous distribution is in general a sub-probability, because there can be infinite non-terminating computations and we do not collect probability from them. However, there are general probabilistic constructions that can avoid those problems, cf. [23], but we do not follow that direction here. In fact, we are satisfied by

the interpretation of the loss of probability mass as the probability that the program never terminates.

5.2 Continuous Time

In this section we define the continuous time concretization of the operational semantic. The idea is to define an underlying Continuous Time Markov Chain (CTMC) [16] not by assigning a rate to each transition, but by giving the jumping probability and the holding times in each node of the transition graph. In other words, we take the DTMC defined above, and we attach to each node a rate, corresponding to the rate of an exponential distribution returning the time spent in that node before doing the transition. To define the observables, we make use of the Q-matrix associated to the Markov chain, that can be easily reconstructed from the jumping probabilities and the global rates, cf. [16]. From the Q-matrix we can reconstruct the distribution of probability of going to a state j from a state i in time t, by solving the following system of differential equations (that can be infinite, if so is the state space):

$$P'(t) = QP(t),\ P(0) = Id. \tag{1}$$

The introduction of a notion of time allows a more refined modeling than in the case of a discrete succession of states. The idea behind is that the active processes at a certain node of the transition graph compete through a race condition, in such a way that the fastest one is executed, and then the race is restarted from scratch, thanks to the memoryless property of Markov Chains. Continuity of time assures that the probability of two processes terminating the race condition exactly at the same time is zero.
First we give the definition of continuous time traces (they are essentially the abstract traces, but written in a different form to be distinguished), and then we give the notion of continuous time observables, as a function of the elapsed time t.

Definition 8

- *A* continuous time trace *is a sequence* $\sigma_1 (p_1, \lambda_1) \sigma_2 (p_2, \lambda_2) \sigma_3 \ldots (p_n, \lambda_n) \sigma_{n+1}$, *and the set of all continuous time traces of variable length is* $\mathcal{CT}$.
- *The* continuous time concretization χ *is a function* $\chi : \mathcal{T} \to \mathcal{CT}$. *If* $\tau \in \mathcal{T}$ *is* $\tau = \sigma_1 \longrightarrow_{(p_1,\lambda_1)} \ldots \longrightarrow_{(p_n,\lambda_n)}$, *then* $\chi(\tau) = \chi_\tau = \sigma_1 (p_1, \lambda_1) \ldots (p_n, \lambda_n) \sigma_{n+1}$.
- *The probability of going from* σ_I *to* σ_O *in time* t *is* $\mathrm{Prob}(\sigma_I \longrightarrow \sigma_O)(t) = P(t)_{\sigma_I,\sigma_O}$, *where* $P(t)$ *is the matrix defined in equation (1).*

We have now to abstract from the set of local variables V, as done for the discrete time case. The probability of going from a a state $\langle A, d\rangle$ to a state $\langle A', d'\rangle$ in time t is

$$\mathrm{Prob}(\langle A, d\rangle \longrightarrow \langle A', d'\rangle)(t) = \sum_{|V|=1}^{\infty} \mathrm{Prob}\left(\langle A, d, \emptyset\rangle_\mu \longrightarrow \langle A', d', V\rangle_\mu\right)(t).$$

Definition 9 *The continuous time I/O observables at time* t *are a sub-probability*

distribution over the constraint store, defined as

$$\mathcal{O}_c(\langle A, d\rangle)(t) = \{(d', p) \mid p = \mathrm{Prob}\left(\langle A, d\rangle \longrightarrow \langle \mathbf{0}, d'\rangle\right)(t)\}\,.$$

Intuitively, this gives the probability that a process terminates with final store d' in t units of time or less. This is a sub-probability because in general not all the computations have stopped after time t. We pinpoint that the notion of observable defined here is a function of the elapsed time. Actually, it corresponds to a sub-vector of the row $P(t)_{\langle A,d,\emptyset\rangle_\mu}$ of the matrix $P(t)$.[3] Continuity of $P(t)$ implies that $\mathcal{O}_c(t)$ is continuous. An interesting question is the long term behaviour of $\mathcal{O}_c(t)$, i.e. what happens in the limit $t \to \infty$. Essentially, we are asking what is the probability of eventually reaching a terminal state. But those states are absorbing classes of the CTMC, and a general result guarantees that those long term probabilities depend only on the underlying discrete chain (cf. [16] for further details). This fact allows us to prove the following

Theorem 1

$$\lim_{t\to\infty} \mathcal{O}_c(\langle A, d\rangle)(t) = \mathcal{O}_d(\langle A, d\rangle).$$

■

Therefore, the t-dependent observable distribution in the continuous case converges to the observable distribution defined for the discrete model of time.

This theorem connects together the long term behaviour of the discrete and the continuous time concretizations. Essentially, it states that in the asymptotic limit the additional information introduced by the continuous time model evaporates, leading to the same probability distribution induced by the discrete time model. This result is not straightforward, though its proof in the framework presented here is simple.

6 An Example

In this section we present a simple example, in order to show the main features of the language, with a particular focus on the observables. We consider a very simple program, composed by just one agent, i.e. $A = \mathrm{tell}_{\lambda_1}(c) \parallel \mathrm{ask}_{\lambda_2}(c).\mathrm{tell}_{\lambda_3}(d) + \mathrm{tell}_{\lambda_4}(e)$. The constraint system we have in mind here is very simple: c, d, e are the basic tokens, and all the different combination via $\sqcup$ of them are different. As in A there isn't any hiding operator, we can safely forget the set of local variables in the configurations. When the computation starts from an empty constraint store $\top$, the execution tree is depicted in Figure 1. Note that on the edges of the tree we have two numbers as labels: the first one is the probability associated to that transition, and the second one is the rate of the transition itself. At the beginning we can observe a race condition between two tell processes (the ask is not enabled),

[3] Actually, it corresponds to a linear function of that vector, due to the aggregation on the set of local variables.

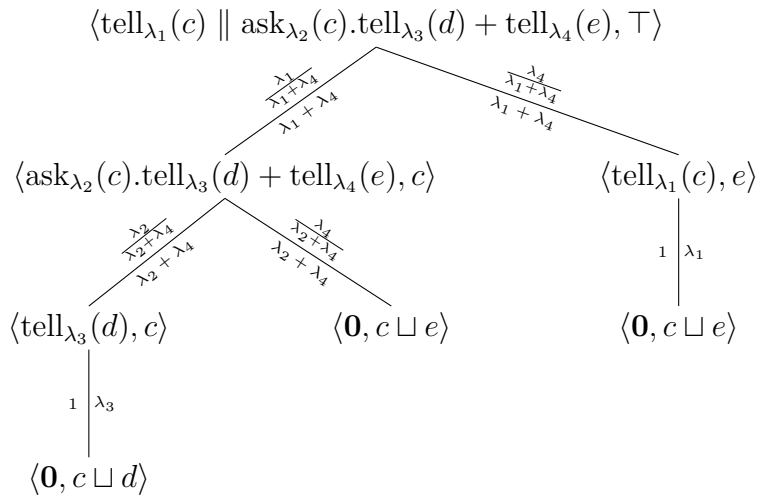

Fig. 1. The execution tree of the program $\langle \mathrm{tell}_{\lambda_1}(c) \parallel \mathrm{ask}_{\lambda_2}(c).\mathrm{tell}_{\lambda_3}(d) + \mathrm{tell}_{\lambda_4}(e), \top \rangle$

one with rate λ_1 and one with rate λ_4. So the probability of adding c to the store is $\frac{\lambda_1}{\lambda_1+\lambda_4}$, while e will be added with probability $\frac{\lambda_4}{\lambda_1+\lambda_4}$. In either case, the rate of transition is $\lambda_1 + \lambda_4$. Other transitions work similarly.

It is clear from the tree in Figure 1 that there are just two terminal states, i.e. $\langle \mathbf{0}, c \sqcup d \rangle$ and $\langle \mathbf{0}, c \sqcup e \rangle$. To compute the discrete time observables, we have to collect together all the probabilities of the traces leading to them, giving:

$$\mathcal{O}_d(\langle A, \top \rangle) = \left\{ (c \sqcup d, \frac{\lambda_1 \lambda_2}{\overline{\lambda}}), (c \sqcup e, \frac{\overline{\lambda} - \lambda_1 \lambda_2}{\overline{\lambda}}) \right\},$$

with $\overline{\lambda} = (\lambda_1 + \lambda_4)(\lambda_2 + \lambda_4)$. Note that this is a probability, as all the computations terminate.

To compute the observables for the continuous time case, we have to calculate the Q matrix for the underlying chain. First we have to fix an ordering for the states of the program. If we set $A_1 = \langle \mathrm{tell}_{\lambda_1}(c) \parallel \mathrm{ask}_{\lambda_2}(c).\mathrm{tell}_{\lambda_3}(d) + \mathrm{tell}_{\lambda_4}(e), \top \rangle$, $A_2 = \langle \mathrm{ask}_{\lambda_2}(c).\mathrm{tell}_{\lambda_3}(d) + \mathrm{tell}_{\lambda_4}(e), c \rangle$, $A_3 = \langle \mathrm{tell}_{\lambda_1}(c), e \rangle$, $A_4 = \langle \mathrm{tell}_{\lambda_3}(d), c \rangle$, $A_5 = \langle \mathbf{0}, c \sqcup d \rangle$, $A_6 = \langle \mathbf{0}, c \sqcup e \rangle$, with the ordering induced by the indexes, we have:

$$Q = \begin{pmatrix} -\lambda_1 - \lambda_4 & \lambda_1 & \lambda_4 & 0 & 0 & 0 \\ 0 & -\lambda_2 - \lambda_4 & 0 & \lambda_2 & 0 & \lambda_4 \\ 0 & 0 & -\lambda_1 & 0 & 0 & \lambda_1 \\ 0 & 0 & 0 & -\lambda_3 & \lambda_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Given the matrix Q, we can compute very easily the probability of going to one state to another in a certain time t, and consequently the observables. In fact,

for a finite matrix, the solution of Equation (1) is, in terms of matrix exponentials, $P(t) = e^{tQ}$.

For instance, if $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 1$, we have

$$\mathcal{O}_c(\langle A, \top \rangle)(1) = \{(c \sqcup d, 0.0513), (c \sqcup e, 0.3483)\},$$

$$\mathcal{O}_c(\langle A, \top \rangle)(2) = \{(c \sqcup d, 0.1467), (c \sqcup e, 0.6009)\},$$

$$\mathcal{O}_c(\langle A, \top \rangle)(100) = \{(c \sqcup d, 0.2500), (c \sqcup e, 0.7500)\}.$$

As we can see, the observables converge fast to the stationary distribution given by

$$\mathcal{O}_d(\langle A, \top \rangle) = \{(c \sqcup d, 0.2500), (c \sqcup e, 0.7500)\}.$$

7 Related Work

This version of CCP belongs to the class of process algebras, like PEPA [15], stochastic π-calculus [17] or EMPA [1], just to cite a few. There is, however, an important difference between CCP and these languages, namely the fact that communication in CCP is asynchronous, while the other languages implement a synchronous interaction between agents. Asynchronous communication is, in fact, simpler to deal from the point of view of performance analysis, as the definition of synchronization's rate is troublesome (cf. [3]). In this light, there is a stronger similarity with asynchronous stochastic languages like pKLAIM [7] and especially stochastic KLAIM [6], where rates are assigned to communication instructions.

CCP has been extended in the probabilistic setting in two different ways. In [12] the probabilistic ingredient is introduced by the addition of random variables in the constraint store, while the instructions of the language remain unchanged. More similar to our approach is the probabilistic version presented in [9], where probability distributions are attached both to the summation operator and to the parallel composition. This gives rise to a probabilistic language with a discrete model of time, cf. Section 4 for further comments. To our knowledge, this is the first proposal of a continuous time stochastic version of CCP.

8 Conclusions and Future Work

We presented a stochastic version of CCP, where each instruction communicating with the store has a real number attached to it, representing either the priority of the instruction or its "speed" rate. The operational semantic is given in an abstract form, that can be concretized with a specific model of time, either discrete or continuous. In addition, a concept of observables, corresponding to input/output behaviour is defined both for the discrete time and the continuous time version, and moreover the usual algebraic properties of summation and parallel composition are preserved. The bridge between discrete and continuous time is given by Theorem 1, stating their equivalence in the long term behaviour.

In the future we plan to work on a denotational model for the language, following the ideas of [8,10]. The challenge is to extend their methods to the continuous time case. Furthermore, as stated in the introduction, we would like to use this language to model biological processes. Towards this goal we need a simulation engine, i.e. the equivalent of SPiM [4] for π-calculus. Moreover, the development of probabilistic model checking procedures [19] for the probabilistic versions of CCP seems also another fundamental step in that direction.

Acknowledgments

Many thanks to Alessandra Di Pierro and Herbert Wiklicky for the useful suggestions and discussions, and to Agostino Dovier for his unrivalled support.

References

[1] M. Bernardo and R. Gorrieri. A tutorial on empa: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. Technical Report 96–17, Department of Computer Science, University of Bologna, 1997.

[2] F.S. de Boer, R.M. van Eijk, W. van der Hoek, and J-J.Ch. Meyer. Failure semantics for the exchange of information in multi-agent systems. In *Proceedings of CONCUR 2000*, 1998.

[3] J. Bradley and N. Davies. Reliable performance modeling with approximate synchronisations. In J. Hillston and M. Silva, editors, *Proceedings of the 7th workshop on process algebras and performance modeling*, pages 99–118. Prensas Universitarias de Zaragoza, September 1999.

[4] L. Cardelli and A. Phillips. A correct abstract machine for the stochastic pi-calculus. In *Proceeding of Bioconcur 2004*, 2004.

[5] F.S. de Boer, A. Di Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151(1), 1995.

[6] R. De Nicola, D. Latella, and M. Massink. Formal modeling and quantitative analysis of klaim-based mobile systems. In *Proceedings of SAC05*, 2005.

[7] A. Di Pierro, C. Hankin, and H. Wiklicky. Continuous time probabilistic klaim. In *Proceedings of SECCO 2004*, 2004.

[8] A. Di Pierro and H. Wiklicky. A banach space based semantics for probabilistic concurrent constraint programming. In *Proceedings of CATS'98*, 1998.

[9] A. Di Pierro and H. Wiklicky. An operational semantics for probabilistic concurrent constraint programming. In *Proceedings of IEEE Computer Society International Conference on Computer Languages*, 1998.

[10] A. Di Pierro and H. Wiklicky. Operator algebras and the operational semantics of probabilistic languages. In *Proceedings of MFCSIT 2004*, 2004.

[11] A. Garcia Villanueva. *Model Checking for the Concurrent Constraint Programming Paradigm*. PhD thesis, University of Udine, 2003.

[12] V. Gupta, R. Jagadeesan, , and V. A. Saraswat. Probabilistic concurrent constraint programming. In *Proceedings of CONCUR'97*, 1997.

[13] Rety. J. H. Distributed concurrent constraint programming. *Fundamentae Informaticae*, 34(3):323–346, 1998.

[14] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras, Part I*. North-Holland, Amsterdam, 1971.

[15] J. Hillston. Pepa: Performance enhanced process algebra. Technical report, University of Edimburgh, 1993.

[16] J. R. Norris. *Markov Chains*. Cambridge University Press, 1997.

[17] C. Priami. Stochastic π-calculus. *The Computer Journal*, 38(6):578–589, 1995.

[18] C. Priami and P. Quaglia. Stochastic π-calculus. *Briefings in Bioinformatics*, 5(3):259–269, 2004.

[19] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems*, P. Panangaden and F. van Breugel (eds.), volume 23 of *CRM Monograph Series*. American Mathematical Society, 2004.

[20] V. A. Saraswat. *Concurrent Constraint Programming*. MIT press, 1993.

[21] V. A. Saraswat, R. Jagardeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proceeding of 9th Annual IEEE Symposium on Logic and Computer Science*, pages 71–80, 1994.

[22] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proceedings of POPL*, 1991.

[23] P. C. Shields. *The Ergodic Theory of Discrete Sample Paths*, volume 13 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, Rhode Island, 1996.