An Operational Semantics for Probabilistic Concurrent Constraint Programming

Alessandra Di Pierro Herbert Wiklicky

Department of Computer Science
City University London
London EC1V OHB, UK

Abstract

This paper investigates a probabilistic version of the concurrent constraint programming paradigm (CCP). The aim is to introduce the possibility to formulate so called "randomised algorithms" within the CCP framework. Differently from common approaches in (imperative) high-level programming languages, which rely on some kind of `random()` *function, we introduce randomness in the very definition of the language by means of a probabilistic choice construct. This allows a program to make stochastic moves during its execution. We call the resulting language Probabilistic Concurrent Constraint Programming (PCCP).*

We present an operational semantics for PCCP by means of a probabilistic transition system such that the execution of a PCCP program may be seen as a stochastic process, i.e. as a random walk on the transition graph. The transition probabilities are given explicitly. This semantics captures a notion of observables which combines results of computations and the probability of those results being computed.

This embedding of randomness within the semantics of a well structured programming paradigm, like CCP, also aims at providing a sound framework for formalising and reasoning about randomised algorithms and programs. Additionally, we give some examples of PCCP programs and we show how well-known randomised algorithms can be implemented very naturally in the new language.

1 Introduction

Algorithms incorporating randomness (e.g. a "coin flipping" device), such as those for calculating π or more generally the integration of complicated functions, have been known in mathematics for a long time (e.g. Monte Carlo algorithms).

However, it is only in the last decade that algorithms where the flow of information is determined by an element of random choice (so called *randomised algorithms*) have found widespread application in many different areas of computer science, for example as a tool in computational geometry and number theory. They are intensively studied especially from a complexity theoretic point of view. Various randomised algorithms and procedures have been investigated, of which we mention just a few examples: simulated annealing in combinatorial optimisation [1], genetic algorithms [7], probabilistic primality tests in particular for use in crypto-systems [21], and randomised proof procedures (e.g. for linear logic [17]).

The benefits of randomisation at the base of the tremendous growth of interest in such algorithms are simplicity and speed. For this reason the best known algorithms for many problems are nowaday randomised.

In this paper we introduce a method for expressing randomness in concurrent constraint programming (CCP), to the purpose of using this paradigm to formalise randomised algorithms. In our approach we incorporate this element of randomness in the form of a *probabilistic choice* directly within the semantics. This is different from other approaches where randomness is introduced either in an ad-hoc way by referring to some "external" function or procedure call (like `random()` in imperative languages such as C or FORTRAN), or by means of random variables [15, 10].

The resulting language, which we call Probabilistic CCP (PCCP), is then the concurrent constraint programming language where nondeterministic choice is replaced by a probabilistic one. This allows a program to make stochastic moves during its execution.

We define an operational semantics for PCCP by using the inductive style of Plotkin's SOS, i.e. by means of a transition system which describes the evolution of a program in a structural way [22]. Randomness is expressed by labels representing the probability that a transition takes place. Based on this we define

a notion of observables which captures the (finite) results of computation together with their probability of being computed.

Since our intent is to randomise programs rather than data, we do not need to base our language on any kind of fuzzy or belief system [14]. Therefore, it is sufficient for our purpose to maintain the same standard (cylindric) constraint system as in CCP.

The embedding of randomness within the semantics of a well structured programming paradigm, like CCP, also aims at providing a sound framework for reasoning about randomised algorithms and programs. In fact, as already mentioned before, up to now probabilistic algorithms have been written by using imperative languages extended with random facilities like pseudo-random number generators, and have been analysed largely only by ad-hoc methods. However, real problems are normally quite complicated and intuition is very likely to fail. Therefore, a formal deductive system would be much more handy in such cases. The task of defining such a system, which in imperative languages is a very complicated one, can be accomplished in CCP more easily, due to the "declarativeness" of this paradigm and its well-established denotational semantics simple and mathematically rigorous at the same time.

2 Probabilistic Nondeterminism

Before introducing the probabilistic concurrent constraint (PCCP) language we discuss the basic question of what a probabilistic choice is and how it is different from a (purely) nondeterministic one.

Nondeterminism in its pure form is not a computational concept: it just indicates all possible transitions in a machine, a deductive system, etc. but says nothing on how to make an actual choice among the various alternatives.

Of course, in any physical realisation of a nondeterministic machine one has to deal with the problem of actually making choices. As a consequence, a "realistic" interpretation of a nondeterministic system is usually that nondeterministic choices, though beyond the control of the system itself, are supposed to be resolved by a scheduler. The scheduler is traditionally thought of as a fictitious entity called "adversary", for it is supposed to make the worst nondeterministic decisions (as a worst case context is normally considered in the study and analysis of algorithms).

Alternatively, the problem of nondeterminism can be approached as the absence of information which needs to be specified for the system be able to continue its computation. In this case the solution is to "refine" the notion of nondeterminism so as that some additional requirement on how the choice has to be made is available together with the possibility itself to make a choice.

One example of such a "refinement" is the notion of a *fair choice* where it is required that all the possible choices are actually made sometime. Note that this only specifies something (additional) about the computation we might expect to observe, but does not change the (number of) possibilities of how to continue with that computation.

Another example of providing more information of how possible branches of computation will be realised is the notion of a *probabilistic choice*. Here we put some requirement on the frequency of choices. We specify how likely it is (repeating the same computation 'sufficiently' often) that a certain possible continuation of the computation is actually performed.

In this paper we present a method of introducing a probabilistic nondeterminism in CCP. We explain the idea behind this notion by discussing an example well-known in probability theory, namely the *random walk* [8, 9].

2.1 Random Walk

In order to illustrate the difference between pure nondeterminism and our approach to a probabilistic version of nondeterminism consider a (directed or non-directed) graph $\Gamma = (N, E)$. To each edge of Γ is assigned a probability $p(v)$ of choosing it. It is therefore reasonable (and necessary) to require that the sum of all probabilities assigned to the edges leaving one node is one. A so-called *random walk* on Γ can be then described as follows: We start at an initial node and (according to the assigned probabilities) choose one adjacent node to go to. From there we repeat this procedure. We obtain this way some (random) path through the graph. In this setting we might not only ask if two nodes are connected, i.e. if a path exists, but also how likely it is to find a path from one node to another.

A very simple example of such a process is obtained by considering a random walk on a graph where each node corresponds to an integer, and two nodes are connected if they differ only by ± 1. If we further assume that the probability to go from one integer to its successor or to its predecessor is each 50% we get a process usually referred to as a symmetric random walk in one dimension, which perhaps resembles a drunken sailor's way staggering along a narrow street.

We can calculate the probability p of how likely it is that after one step, or two steps, etc. we will be at position n in this graph. Let us express this by an ordered pair $< n, p >$. Suppose we start at the

$$\{<\pm 0, 1>\}$$
$$\{<-1, \tfrac{1}{2}>, <+1, \tfrac{1}{2}>\}$$
$$\{<-2, \tfrac{1}{4}>, <\pm 0, \tfrac{1}{2}>, <+2, \tfrac{1}{4}>\}$$
$$\{<-3, \tfrac{1}{8}>, <-1, \tfrac{3}{8}>, <+1, \tfrac{3}{8}>, <+3, \tfrac{1}{8}>\}$$
$$\{<-4, \tfrac{1}{16}>, <-2, \tfrac{4}{16}>, <\pm 0, \tfrac{6}{16}>, <+2, \tfrac{4}{16}>, <+4, \tfrac{1}{16}>\}$$
$$\{<-5, \tfrac{1}{32}>, <-3, \tfrac{5}{32}>, <-1, \tfrac{10}{32}>, <+1, \tfrac{10}{32}>, <+3, \tfrac{5}{32}>, <-5, \tfrac{1}{32}>\}$$
$$\{<-6, \tfrac{1}{64}>, <-4, \tfrac{6}{64}>, <-2, \tfrac{15}{64}>, <\pm 0, \tfrac{20}{64}>, <+2, \tfrac{15}{64}>, <+4, \tfrac{6}{64}>, <+4, \tfrac{1}{64}>\}$$
$$\ldots$$

Table 1: Symmetric random walk in one dimension.

origin, i.e. the initial situation is given by $<\pm 0, 1>$: with certainty we are at the origin. After one step, with probability $\frac{1}{2}$ we will be either at position $+1$ or position -1.

When we continue this walk we get the picture in Table 1 (where impossible positions, i.e. with $p = 0$ are omitted). In principle this is just Pascal's triangle where additionally going a step further means also that we have to half all the probabilities involved.

It is now possible to investigate further properties of this simple random walk [8, 9]. For example, we could ask how likely it is to return to given point, e.g. the origin. The probability of returning some time to the origin is the probability of returning after one step, plus the probability of returning after two steps, etc. It is a well-known result that the recurrence probability is one. That means that although there exist paths which will never bring us back their probability vanishes and it is practically certain to return to the starting point.

3 Probabilistic CCP

In order to allow the formulation of various randomised algorithms – some of which we mentioned in the introduction – within the declarative framework of constraint programming we introduce a new language we call Probabilistic Concurrent Constraint Programming (PCCP).

The syntax of PCCP is an extension of classical CCP as given in Table 2. The intuitive semantics is identical to the common one in CCP, except that we introduce a probabilistic choice (instead of a purely nondeterministic one) as

$$\prod_{i=1}^{n} c_i \mid p_i \rightarrow A_i.$$

The intended meaning of this construct is as follows: First, check whether constraints c_i are entailed by the store. All those agents A_i such that c_i are entailed are called *enabled*.

After that we have to *normalise* the probability distribution by considering only the enabled agents. This can be done by replacing p_i for enabled agents by a normalised transition probability:

$$\tilde{p_i} = \frac{p_i}{\sum_{\vdash c_j} p_j}$$

where the sum is over all enabled agents. This guarantees that $\sum \tilde{p_i} = 1$.

Finally, one of the enabled agents is chosen according to this new probability distribution $\tilde{p_i}$.

Note that *deterministic* agents can be obtained by imposing $n = 1$. This means that only one agent is enabled and therefore the normalised probability has to be 1.

We also replace pure non-determism associated with the paralell construct:

$$A \parallel A$$

by a probabilistic version. Basically, we assume that the scheduler treats all interleavings equally and choses any of them with the same probability.

Like all CCP languages, PCCP is defined parametrically w.r.t. to a given *constraint system*. For PCCP we do not require any substantial modification of the standard cylindric constraint systems as defined in [24, 25]. In other words, we do not intend to introduce randomness on the data, which would require to restructure the constraint system as a fuzzy or belief system. Since the only randomisation is on the programs (i.e. on the choice construct) we can safely assume a standard constraint system as underlying our language.

4 Operational Semantics for PCCP

We define the operational semantics in the style of SOS [22], i.e. by means of a transition system which

$$
\begin{array}{lcl}
P & ::= & D.A \\
D & ::= & \epsilon \mid D.D \mid p(x) :- A \\
A & ::= & \mathbf{stop} \mid \mathbf{tell}(c) \mid \Box_{i=1}^{n}\ c_i | p_i \rightarrow A_i \mid A \parallel A \mid \exists_x A \mid p(x)
\end{array}
$$

Table 2: The syntax for PCCP.

describes the evolution of the system of concurrent agents in a structural way.

A configuration represents the state of the system at a certain moment, namely the agent A which has still to be executed, and the common store d. We denote a configuration by $\langle A, d\rangle$.

In order to express the probability of each transition step we introduce a *labelled* transition system, which consists of a pair $(Conf, \longrightarrow_p)$, where $Conf$ is a set of *configurations* and $\longrightarrow_p \subseteq Conf \times \mathbb{R} \times Conf$ is the transition relation defined by the rules in Table 3.

The label p represents the transition probability between two configurations. We denote the transitive closure of $\longrightarrow_p$ by $\longrightarrow_p^*$. As we deal with a labelled transition system, we also have to define the label p associated to $\longrightarrow_p^*$. It is obvious that the probability of getting from one configuration to another one in several steps is determined by the product of the probabilities associated to the single steps.

Suppose, for example, we have

$$A_1 \longrightarrow_{p_{12}} A_2 \text{ and } A_2 \longrightarrow_{p_{23}} A_3$$

then

$$A_1 \longrightarrow_{p_{13}}^* A_3$$

with

$$p_{13} = p_{12}\ p_{23}.$$

The agent **stop** represents successful termination. The basic actions are given by 'telling' a constraint to the common store and by 'asking' if a constraint is entailed by the store. The first action is expressed by the **tell**(c) construct. Given a store d, as shown by rule **R1**, the execution of **tell**(c) updates the store to $c \sqcup d$ and then stops. This transition, as it is the only possible one, is done with probability 1.

The ask action is incorporated in the choice construct in the form $c \mid p \rightarrow A$. In case the number of alternatives is just one, the choice construct corresponds to the **ask**$(c) \rightarrow A$ of CCP. The effect of the probabilistic choice has been described informally in the previous section. Rule **R2** gives a formal description. One of the agents A_j is enabled if the store entails c_j. After normalising the probabilities to

$$\tilde{p_j} = \frac{p_j}{\sum_{\vdash c_i} p_i}$$

as described before, a probabilistic choice (according to $\tilde{p_j}$) is made among the enabled agents. This means that if this choice is repeated (under the same condition and sufficiently often) the relative frequency of executions of an agent A_j is exactly $\tilde{p_j}$.

In rule **R3** the semantics of the parallel execution of agents is described. It is basically modelled as interleaving. Rule **R4** describes the usual semantics of the hiding operation. We use the notation $\exists_x^d A$ for the agent A with local store d containing information on x which is hidden in the external store (see [5, 24] for further details). Obviously, the transition probability p is not changed by hiding.

The execution of a procedure call, $p(y)$, is modelled by the recursion rule **R5** which replaces $p(y)$ by the body of its definition in the program P, after the link between the actual parameter y and the formal parameter x has been established. Following the method introduced in [25], we express this link between formal and actual parameters by the context $\exists_\alpha(\delta_{y\alpha} \wedge \exists_x(\delta_{\alpha x} \wedge \ldots))$, where α is a variable which does not occur free in any program P or goal G (i.e. $\exists_\alpha c = c$ for all constraints c occurring in P or G), and α does not occur as a parameter (either as a formal parameter or as an actual parameter). Note that throughout the whole computation only one variable α is needed. As in rule **R1** the associated transition probability is 1.

5 Interleaving

The transition rule **R3** for the paralell construct has two conclusions. That means that its execution will lead to two different transition trees. We treat both of these interleavings as independent. In each interleaving the transition probabilities remain unchanged.

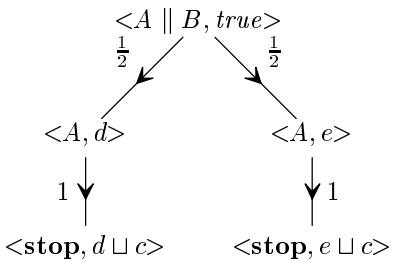
R1	$<\mathbf{tell}(c), d> \rightarrow_1 <\mathbf{stop}, c \sqcup d>$
R2	$<\Box_{i=1}^{n} \ c_i \mid p_i \rightarrow A_i, d> \rightarrow_{p_j} <A_j, d> \qquad j \in [1,n] \ and \ d \vdash c_j$
R3	$\dfrac{<A, c> \rightarrow_p <A', c'>}{\begin{array}{c}<A \parallel B, c> \rightarrow_p <A' \parallel B, c'> \\ <B \parallel A, c> \rightarrow_p <B \parallel A', c'>\end{array}}$
R4	$\dfrac{<A, d \sqcup \exists_x c> \rightarrow_p <B, d'>}{<\exists_x^d A, c> \rightarrow_p <\exists_x^{d'} B, c \sqcup \exists_x d'>}$
R5	$<p(y), c> \rightarrow_1 <\exists_\alpha (\delta_{y\alpha} \wedge \exists_x (\delta_{\alpha x} \wedge A)), c> \qquad p(x) :- A \in P$

Table 3: The transition system for PCCP.

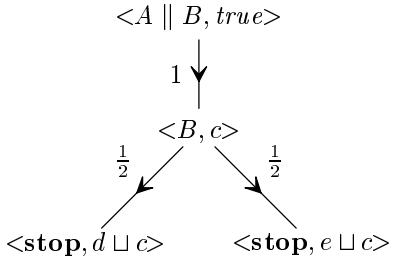
For example, let us introduce the following shorthand notation for two agents A and B:

$$\begin{array}{lcl} A & \equiv & \mathbf{tell}(c) \\ B & \equiv & true \mid \frac{1}{2} \ \rightarrow \ \mathbf{tell}(d) \\ & & \Box \ true \mid \frac{1}{2} \ \rightarrow \ \mathbf{tell}(e). \end{array}$$

We obtain two possible interleavings described by the following transition trees:



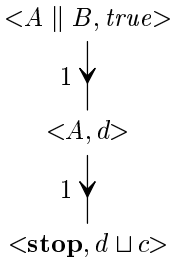
and



With the two agents A and B defined as:

$$\begin{array}{lcl} A & \equiv & \mathbf{tell}(c) \\ B & \equiv & true \mid \frac{1}{2} \ \rightarrow \ \mathbf{tell}(d) \\ & & \Box \ c \mid \frac{1}{2} \ \rightarrow \ \mathbf{tell}(e) \end{array}$$

the first derivation tree degenerates to



and the second one remains as in the first example.

Only when we define observables (see Section 6) we will consider the combination of the results of computations we obtain in all possible interleavings. This will lead to a "normalisation" of probabilities in order to obtain the correct notion of observables.

One could also try to replace **R3** in order to incorporate such a normalistaion directly in the transition rules. For example, one could replace **R3** by a slight variation **R3'**, which would attempt to model a scheduler which executes one of two agents with the same probability of $\frac{1}{2}$.

$$\dfrac{<A, c> \rightarrow_p <A', c'>}{\begin{array}{c}<A \parallel B, c> \rightarrow_{\frac{p}{2}} <A' \parallel B, c'> \\ <B \parallel A, c> \rightarrow_{\frac{p}{2}} <B \parallel A', c'>\end{array}}$$

Unfortunately, such a direct approach would lead to a non-associative paralell construct. For example, in a program like $(A \parallel B) \parallel C$ the agents A and B will be scheduled first with a 25% chance, while C has a 50% chance, whereas in $A \parallel (B \parallel C)$ the situation is that A has a 50% chance, and B and C will be scheduled first only in 25% of cases.

Alternatively, one could consider the paralell execution of n agents (instead of a binary paralell operator), but this leads to a very tedious expresssion as we have to consider $n!$ interleavings.

6 The Observables

If there are several paths leading from configuration A to B, then, in order to get the right probability associated to the last configuration, we have to consider the sum of all the probabilities associated to these different paths. Additionally, we have to collect and normalise the probabilities in different interleavings. This is not done via the transition system but will be handled by the notion of observables we introduce next.

From the operational model described above we can abstract various semantics depending on what we want to *observe* of a process. The classical notion of observables in CCP are the results computed by an agent for a given initial store, where we can look at finite as well as infinite computations. Fair computations are to be considered when infinite computations are taken into account. The result of a fair computation is defined as the least upper bound of all the stores occurring in the computation. This definition is possible because, for the monotonic properties of CCP, the stores of a computation form a chain.

As for CCP we will consider also for PCCP a notion of observables corresponding to the input/output behaviour of an agent. However, in our case we consider not only the possible outcomes of a computation but also the probailities associated.

Given a program P, we define the result $\mathcal{R}_P$ of an agent A and an initial store d as the set of all pairs $<c, p>$, where c is the least upper bound of the partial constraints accumulated during a computation starting from d; and p is the probability of reaching that result.

$$\begin{aligned}\mathcal{R}(A,d) &= \{<c,p> \mid <A,d> \longrightarrow_p^* <B,c> \not\longrightarrow\} \cup \\ &\quad \{<\bigsqcup_i d_i, \prod_i p_i> \mid <A,d_0> \longrightarrow_{p_0} \ldots\}.\end{aligned}$$

The first term describes the results of finite computations where the least upper bound of the partial store corresponds to the final store. The second term covers the infinite results.

The probability of obtaining a certain result depends on the probabilities p associated to the possible paths which lead to it. To capture the true behaviour of an agent we have to identify different computational paths leading to the same result as well as to collect the accumulated probabilities associated with different interleavings. In order to do this in a precise way we have to define two operations. The first collects all the probabilities along different paths and in different interleavings associated to the same resulting constraint (store):

$$\mathcal{K}(\{<c_i, p_i>\}_i) = \{<c_i, P_{c_i}> \mid P_{c_i} = \textstyle\sum_i p_i\}.$$

The second operation normalises the probabilities. This is necessary as the probabilities in each interleaving add up to one such that the overall sum of probabilities is exactly the number of possible interleavings. This process of re-normalisation effectively implies that all interleavings are equally likely.

$$\mathcal{N}(\{<c_i, p_i>\}_i) = \{<c_i, \tfrac{p_i}{P}> \mid P = \textstyle\sum_i p_i\}.$$

With these two operations we can define the observables associated to an agent A and an initial store simply as:

$$\mathcal{O}_P(A,d) = \mathcal{N}(\mathcal{K}(\mathcal{R}(A,d))).$$

Note that this notion of observables differs from the classical notion of input/output behaviour in CCP. In the classical case a constraint c belongs to the input/output observables of a given agent A if *at least* one path leads from the initial store d to the final result c. In the probabilistic case we have to consider *all* possible paths leading to the same result c and combine the associated probabilities.

In the two examples in Section 5 the observables are in the first case:

$$\begin{aligned}\mathcal{O}(A \parallel B, true) &= \mathcal{N}(\mathcal{K}(\{<d \sqcup c, \tfrac{1}{2}>, <e \sqcup c, \tfrac{1}{2}>, \\ &\qquad <d \sqcup c, \tfrac{1}{2}>, <e \sqcup c, \tfrac{1}{2}>\})) \\ &= \mathcal{N}(\{<d \sqcup c, 1>, <e \sqcup c, 1>\}) \\ &= \{<d \sqcup c, \tfrac{1}{2}>, <e \sqcup c, \tfrac{1}{2}>\}\end{aligned}$$

and in the second case:

$$\begin{aligned}\mathcal{O}(A \parallel B, true) &= \mathcal{N}(\mathcal{K}(\{<d \sqcup c, 1>, <d \sqcup c, \tfrac{1}{2}>, \\ &\qquad <e \sqcup c, \tfrac{1}{2}>\})) \\ &= \mathcal{N}(\{<d \sqcup c, 1\tfrac{1}{2}>, <e \sqcup c, \tfrac{1}{2}>\}) \\ &= \{<d \sqcup c, \tfrac{3}{4}>, <e \sqcup c, \tfrac{1}{4}>\}\end{aligned}$$

It is interesting to observe that it might happen that, although the computation is proceeding randomly, there is nevertheless only one final result c which (by combing the probabilities along several possible paths leading to it) will be reached with probability $p = 1$. Such an agent, therefore, exhibits a completely deterministic input/output behaviour despite the apparent probabilistic nature of the computation. This corresponds to the behaviour of what is called a Las Vegas algorithm. This kind of randomised algorithms – like for example the quicksort algorithm with

a randomly chosen pivot element – always produce a (unique) correct output. The alternative kind of randomised algorithm are Monte Carlo algorithms. They compute (different possible) results, which are correct only with a certain probability [21].

7 Examples

We now present some easy examples of PCCP programs demonstrating the expressiveness of the language and its characteristic features which distinguish it from classical CCP.

Example 7.1 Let A and B be any two agents, and consider the following simple PCCP program.

$$\begin{array}{ll} & c \mid \frac{1}{2} \to A \\ \Box & c \mid \frac{1}{2} \to B \end{array}$$

If the constraint c is not entailed by the store this process will get stuck as none of the agents is enabled. On the other hand, in the case where c is entailed both agents are enabled and are taken with the same probability of 50%. That is, when this program is executed repeatedly the relative frequency of each of the two agents to be executed is the same.

Another very simple example is given by the following program:

$$\begin{array}{ll} & c \mid 1 \to A \\ \Box & c \mid 1 \to B \end{array}$$

Because of the normalisation of probabilities we introduced, this is semantically equivalent to the agent in the previous example, both agents are executed with the same probability of 50%.

Under the hypothesis that the constraint d is not entailed, another equivalent program is given by:

$$\begin{array}{ll} & c \mid 1 \to A \\ \Box & c \mid 1 \to B \\ \Box & d \mid 1 \to C \end{array}$$

Example 7.2 Consider now the following, somehow non-typical PCCP program:

$$\begin{array}{ll} & c \mid 1 \to A \\ \Box & c \mid 0 \to B \end{array}$$

Again, if c is not entailed, then the whole process is stuck. But if c is entailed then the behaviour is as follows: With probability 1 agent A is executed and with probability 0 agent B. Note, that this does not mean that agent B is never executed: When we repeat "test-runs" of this program, then only the relative frequency of observing B being executed among all observed "test-runs" vanishes (in order to make this a deterministic execution we had to eliminate the second branch completely).

Example 7.3 The following program for probabilistic counting illustrates the difference between pure nondeterminism, angelic nondeterminism and probabilistic nondeterminism

$$\begin{array}{lll} nat(x) & :- & true \mid \frac{1}{2} \to \mathbf{tell}(x = 0) \\ & \Box & true \mid \frac{1}{2} \to \exists_y(\mathbf{tell}(x = s(y)) \\ & & \parallel nat(y)) \end{array}$$

With a purely nondeterministic choice (and ignoring the guards) we could not predict what the results of running $nat(x)$ were. It might or might not terminate. Even if it terminates we could not guarantee that it produces as results all natural numbers; it might be that the first branch is always taken and therefore only '0' would be obtained as output.

If we look at the formulation of $nat(x)$ in CCP (which is the same as the PCCP one but for the probability components of the guards) we obtain a more useful semantics. Since the nondeterminism occurring in the CCP formulation is angelic [16] we are guaranteed that $nat(x)$ generates all natural numbers. Note that in the CCP case "fairness" of the choice construct, i.e. all possible branches are sometime executed, is guaranteed by backtracking.

In our PCCP approach, we require that the chances of selecting either of the two branches are equal (namely $\frac{1}{2}$). In this case we can deduce even more information about the expected behaviour of $nat(x)$ than in the CCP case: we see that not only all natural numbers are produced, but also that the likelihood for a larger number to be output decreases (exponentially).

Example 7.4 This example gives a possible implementation of a well known stochastic process, namely the symmetric random walk (in one dimension).

$$\begin{array}{lll} walk(x,y) & :- & x \neq y \mid \frac{1}{2} \to \exists_{x'}(\mathbf{tell}(x' = x + 1) \\ & & \parallel walk(x', y)) \\ & \Box & x \neq y \mid \frac{1}{2} \to \exists_{y'}(\mathbf{tell}(y' = y + 1) \\ & & \parallel walk(x, y')) \\ & \Box & x = y \mid 1 \to A \end{array}$$

This procedure will call itself until both its parameters are equal and then execute the agent A. Each time it is called with different parameters it will increase either variable x or variable y by one, with which one being determined by a probabilistic choice which gives both variables a 50% chance of being increased.

The relation to the classical formulation of the random walk becomes perhaps more obvious if one thinks of the difference between the variables x and y as the

current position. The execution of $walk(1,2)$, for example, corresponds to a random walk starting at position $p = y - x = 1$.

From the theory of random walks or Markov chains we know that the probability of returning to the origin, i.e. to the position $p = 0$ or in our case that both variables once have the same value $x = y$, is one [8]. Therefore, we easily see that this program is semantically equivalent to the PCCP fragment we gave in the example 7.2 where we take as B a never terminating agent; the probability of eventually enabling the agent A (which then will be executed) is one and the probability of an infinite loop is zero.

Example 7.5 One of the most well known randomised algorithms is the so called Metropolis Algorithm or Simulated Annealing [19, 1, 11]. We will present this algorithm in the PCCP language and show how natural it fits into the framework of concurrent constraint programming.

The strategy of this algorithm is inspired by the analogy with the physical process of slowly cooling down a fluid until it crystallises, often referred to as an annealing process.

This algorithm is intended to find the optimal (minimal) solution to a combinatorial optimisation problem. A typical application of this type of problems is the travelling salesman problem (TSP). In this case the set of possible solutions is given by all possible paths connecting a certain number of "cities". The problem is then to find the shortest path connecting all cities.

The basic idea is to start with one possible solution to the problem (*instance* in our program) and then try to improve it by generating new possible solutions, which are compared to the old one and eventually accepted or rejected (in the hope to find a global optimum). This process sometimes also accepts a locally "bad" solution in order to overcome so called local minima. This behaviour is guided by a control parameter *control*, which often is also referred to as "temperature". There are various ways to "cool down", i.e. to change this parameter, during the execution; for details see [1].

The following is a PCCP program which implements a generic version of the simulated annealing algorithm:

$$
\begin{array}{l}
simulated_anneal \ :- \\
\quad initial(control) \\
\quad \| \ first(instance) \\
\quad \| \ simulate(control, instance)
\end{array}
$$

$$
\begin{array}{l}
simulate(control, instance) \ :- \\
\quad \exists_{control'} \ (cool(control, control') \\
\quad \| \ \exists_{instance'} \ (generate(instance, instance') \\
\quad \| \ \neg \ stopcriterion \mid 1 \\
\qquad \rightarrow anneal(control', instance, instance')))
\end{array}
$$

$$
\begin{array}{l}
anneal(control, instance, instance') \ :- \\
\quad cost(instance') \leq cost(instance) \mid 1 \\
\qquad \rightarrow simulate(control, instance') \\
\quad \square \ \ cost(instance') > cost(instance) \mid \\
\qquad \exp(\frac{cost(instance) - cost(instance')}{control}) \\
\qquad \rightarrow simulate(control, instance') \\
\quad \square \ \ cost(instance') > cost(instance) \mid \\
\qquad 1 - \exp(\frac{cost(instance) - cost(instance')}{control}) \\
\qquad \rightarrow simulate(control, instance)
\end{array}
$$

A concrete instance of this algorithm is given by defining two sets of procedures: to specify a concrete "cooling scheme" we had to define the procedures *initial* and *cool* as well as the condition *stopcriterion*; to take account of the problem specific aspects we had to make precise the effect of the procedure $first$ and *generate* as well as to provide a concrete function *cost*.

The main procedure *simulated_annealing* initialises the control parameter *control*, generates the first *instance*, which is an element of the set of possible solutions, and calls the procedure $simulate(control, instance)$.

The actual simulated annealing process is implemented by two mutually recursive defined procedures *simulate* and *anneal*. The procedure *simulate* calculates a new control parameter (by calling a sub-routine *cool*) and generates a new candidate *instance'* by calling *generate*. This depends on the concrete space of solutions; in the TSP example it generates a new path by slightly modifying the old one. Then – provided some kind of halting conditions (represented by a constraint, *stopcriterion*) are not yet satisfied – it is the task of the procedure *anneal* to decide whether the new solution is acceptable or not.

In case the cost (e.g. length of a path) of the newly generated solution is lower than of the old one, the annealing process is continued on the new solution. Otherwise, it is still possible that *anneal* accepts a solution which has a higher cost than the old one. The probability for such an "irrational" decision is $\exp(\frac{cost(instance) - cost(instance')}{control})$. In the beginning the likelihood of accepting a more expensive instance is supposed to be very high but in the course of the annealing process, when *control* is going to zero, this probability vanishes.

8 Conclusions and Further Work

We have introduced a probabilistic version of CCP and an operational semantics for it, which is based on a probabilistic transition system. According to this semantics a PCCP program may be seen as a stochastic process and its execution as a random walk on the transition graph. The transition probabilities are given explicitly.

Furthermore, we have presented a series of examples illustrating the distinctive features of PCCP, and how to implement randomised algorithms in PCCP (see Example 7.5), and to simulate stochastic processes (see Example 7.4). These examples have been successfully implemented by using an experimental interpreter for PCCP, currently being developed by our group, in particular by Nicos Angelopoulos [2].

A substantially different attempt to incorporate probabilistic aspects into CCP languages has been suggested in [10]. There are at least two main differences between that approach and ours. The first one is the different *motivation*: while we are interested in giving a semantics for probabilistic programming languages, the work in [10] is aimed at modelling stochastic systems. The second difference is about the way how to incorporate randomness: we introduce randomness via the *algorithms*, namely by using a random choice construct, while the approach of [10] is based on some kind of fuzzy *data*, i.e. on the use of random variables.

Further work will lead to the formulation of a denotational semantics for PCCP. It is natural to base this semantics on measure theoretic notions, e.g. on spaces of measures, cf. [23, 15, 13]. We are already able to define a Banach space based semantics for a restricted version of PCCP in which all constraints in the probabilistic choice construct are *true* [6]. We called this language Probabilistic Constraint Logic Programming (PLCP). As a next step we plan to define a denotational semantics of a similar type for full PCCP and to compare it with the operational semantics given in this paper.

Additional investigations will also relate PCCP to other semantic approaches towards probabilistic computation, e.g. probabilistic predicate transformers [20, 18], probabilistic logic and model checking [3, 12], and probabilistic process algebras [4].

A possible extension of the PCCP language can be obtained by considering a "fuzzy" constraint system where strict entailment is replaced by a weighted justification of constraints. This would be particularly interesting in the case of a distributed system where one had to weight the quality of certain entailments according to their plausibility. In this case the transition probability could be made dependent on the quality of the entailment of a certain constraint.

Acknowledgments

This work was supporeted by TMR/Marie Curie grants No.FMBICT960884 and No.FMBICT960907.

References

[1] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines.* John Wiley & Sons, Chicester, 1989.

[2] Nicos Angelopoulos, Alessandra Di Pierro, and Herbert Wiklicky. Implementing randomised algorithms in constraint logic programming. In *JICSLP'98 – Joint International Conference and Symposium on Logic Programming*, page (to appear), Cambridge, Massachusetts – London, England, June 1998. MIT Press.

[3] Christel Baier and Marta Z. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. Technical Report CSR-96-12, School of Computer Science, University of Birmingham, June 1996.

[4] Marco Bernardo and Roberto Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. Technical Report UBLCS-96-17, Department of Computer Science, University of Bologna, January 1997.

[5] Frank S. de Boer, Alessandra Di Pierro, and Catuscia Palamidessi. Nondeterminism and Infinite Computations in Constraint Programming. *Theoretical Computer Science*, 151(1), 1995. Selected Papers of the Workshop on Topology and Completion in Semantics, Chartres, France.

[6] Alessandra Di Pierro and Herbert Wiklicky. A Banach space based semantics for Probabilistic Concurrent Constraint Programming. In *CATS'98 – Computing: the 4th Australian Theory Symposium*, volume 20(3) of *Australian Computer Science Communications*, pages 245–259, Singapore – Berlin – New York, February 1998. Springer Verlag.

[7] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison–Wesley, Reading, Massachusetts, 1989.

[8] Geoffrey R. Grimmett and D.R. Stirzaker. *Probability and Random Processes.* Clarendon Press, Oxford., second edition, 1992.

[9] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. American Mathematical Society, Providence, Rhode Island, second revised edition, 1997.

[10] Vineet Gupta, Radha Jagadeesan, and Vijay A. Saraswat. Probabilistic concurrent constraint programming. In *Proceedings of CONCUR 97*. Springer Verlag, 1997.

[11] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1997.

[12] Michael Huth and Marta Kwiatkowska. On probabilistic model checking. Technical Report CSR-96-15, School of Computer Science, University of Birmingham, August 1996.

[13] Claire Jones. *Probabilistic Non-Determinism*. PhD thesis, University of Edinburgh, Edingburgh, 1993.

[14] George J. Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic*. Prentice Hall, New Jersey, 1995.

[15] Dexter Kozen. Semantics for probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.

[16] Marta Z. Kwiatkowska. Infinite Behaviour and Fairness in Concurrent Constraint Programming. In J. W. de Bakker, W. P.de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 348–383, Beekbergen The Nederland, June 1992. REX Workshop, Springer Verlag.

[17] Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Stochastic interaction and Linear Logic. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Note Series*, pages 147–166. Cambridge University Press, Cambridge, 1995.

[18] Annabelle McIver and Carroll Morgan. Probabilistic predicate transformers: Part 2. Technical Report PRG-TR-5-96, Programming Research Group, Oxford University Computing Laboratory, 1996.

[19] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations for fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[20] Carroll Morgan, Annabelle McIver, Karen Seidel, and J.W. Sanders. Probabilistic predicate transformers. Technical Report PRG-TR-4-95, Programming Research Group, Oxford University Computing Laboratory, 1995.

[21] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, 1995.

[22] Gordon Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[23] N. Saheb-Djahromi. CPO's of measures for nondeterminism. *Theoretical Computer Science*, 12:19–37, 1980.

[24] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of POPL*, pages 232–245. ACM, 1990.

[25] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantics foundations of concurrent constraint programming. In *Proceedings of POPL*, pages 333–353. ACM, 1991.