Stochastic Processes as Concurrent Constraint Programs
(An Extended Abstract)

Vineet Gupta
vgupta@mail.arc.nasa.gov
Caelum Research Corporation
NASA Ames Research Center
Moffett Field CA 94035, USA

Radha Jagadeesan*
radha@cs.luc.edu
Dept. of Math. and Computer Sciences
Loyola University–Lake Shore Campus
Chicago IL 60626, USA

Prakash Panangaden†
prakash@cs.mcgill.ca
School of Computer Science
McGill University
Montreal, Quebec, Canada

Abstract

This paper describes a stochastic concurrent constraint language for the description and programming of concurrent probabilistic systems. The language can be viewed both as a calculus for describing and reasoning about stochastic processes and as an executable language for simulating stochastic processes. In this language programs encode probability distributions over (potentially infinite) sets of objects. We illustrate the subtleties that arise from the interaction of constraints, random choice and recursion. We describe operational semantics of these programs (programs are run by sampling random choices), denotational semantics of programs (based on labeled transition systems and weak probabilistic bisimulation), and prove soundness theorems. We show that Probabilistic cc is conservative over cc, thus justifying the design of Probabilistic cc. We use the semantic study to illustrate a novel use of probability to analyze a problem stated without reference to probability, namely the problem of indeterminacy in synchronous programs.

1 Introduction

This paper describes a stochastic concurrent constraint language for the description and programming of concurrent probabilistic systems. The language is both a calculus for reasoning about Markov processes and an executable language that can be used to program stochastic processes.

1.1 The Need for Probability

The traditional conceptual framework of programming languages is based on the assumption that enough information will be available to model a given system in as accurate detail as is needed so that appropriate causal, and maybe even determinate, models can be constructed. However, this assumption is violated in many situations — it becomes necessary to reason and program with approximate, incomplete or uncertain information. We consider three paradigmatic situations.

*Research supported in part by a CAREER grant from NSF.
†Research supported in part by NSERC.

Firstly, probability finds use as an abstraction mechanism to finesse inessential or unknown details of the system and/or the environment. For example, [3] analyzes a component of the Lucent Technologies' 5ESS® telephone switching system that is responsible for detecting malfunctions on the hardware connections between switches. This component responds to alarms being generated from another complicated system that is only available as a black-box. A natural model to consider for the black-box is a stochastic one, which represents the timing and duration of the alarm by random variables with a given probability distribution. [3] then shows that the desired properties hold with extremely high probability. For another instance of modeling a complex environment that is best done statistically, see [24].

Secondly, consider model-based diagnosis settings. Often information about *failure models* and their associated probabilities is obtained from field studies and studies of manufacturing practices. Failure models can be incorporated by assigning a variable, called the *mode* of the component, to represent the physical state of the component, and associating a failure model with each value of the mode variable. Probabilistic information can be incorporated by letting the mode vary according to the given probability distribution [21]. The diagnostic engine computes the most probable diagnostic hypothesis, given observations about the current state of the system.

Thirdly, probability finds use as one of the tools in the design of efficient algorithms for communication and computation in distributed unreliable networks. For example, [35] describes a class of scalable (probabilistic) protocols based on a probabilistic system model whose costs grow slowly with the size of the system. A similar class of examples arise in the area of amorphous computing [1, 10]. Here one assumes vast numbers of programmable elements amassed without precise interconnection and addressing; and solve control problems even though the individual entities are unreliable and interconnected in unknown and irregular patterns. Probabilistic methods are a basic tool used in the design of algorithms in this area, *e.g.* see [19] for probabilistic algorithms to compute maximal independent sets, [48] for probabilistic algorithms to build hierarchical abstractions, and [17] for programming an amorphous computer.

One can take the other extreme position and work with indeterminacy and avoid the subtleties of measure theory and probability theory. There are certainly situations where this is the recommended approach, in particular when one has no quantitative information at all: using a uniform probability distribution is not the same as expressing complete ignorance about the possible outcomes. However, one often does have the quantitative information needed for a probabilistic analysis and then one can get far more information from a probabilistic treatment than from a nondeterministic treatment — [22] shows that the logical characterization of a probabilistic system is almost the same as in the deterministic case and very different from the indeterminate case.

1.2 Design Criteria

The above examples motivate the following criteria on candidate modeling languages.

Concurrency. The need for concurrency arises from two sources. First, note that concurrency arises inevitably in the first and third examples. The second example motivates a second source of concurrency — concurrency as a structuring mechanism to aid in modular construction of models — as motivated by the analysis underlying synchronous programming languages [8, 30, 11].

Thus the intended stochastic language should permit the description of concurrency and interaction between concurrent processes. Furthermore — as earlier work on concurrent constraint programming [54, 50] has shown — concurrent composition is precisely modeled by conjunction. In the stochastic case we now have much richer possibilities for interaction, namely interaction between stochastic processes, but we do not want to model this by introducing a plethora of new combinators. Rather, we adopt as a design principle for our language that we have *a single composition primitive — traditional parallel composition — encompassing both interaction between stochastic processes and determinate processes.*

Executability. The motivation for executability in a programming context such as the third example is evident. Furthermore, even specification contexts, such as the first two examples, benefit from an executable modeling language. At a foundational level, executability helps to bridge the gap between a system and its model. Secondly, in the style of abstract-interpretation based program-analysis techniques, executability provides a systematic way for developing reasoning algorithms, *e.g.* in the stochastic context, [2] uses the simulation engine of the language to develop an algorithm for maximum likelihood estimation, and [42] uses the lazy evaluation mechanism of their stochastic functional programming language to compute conditional probabilities.

Constraints. One of the key motivations of the present work is to have a formalism that allows one to work with physical systems. In such systems the real numbers and associated continuous mathematics plays a significant role. Inevitably in working with such systems one has to work with approximate information, quite apart from the uncertainty in the dynamics. A natural way of expressing this uncertainty is as partial information, or constraints. In traditional numerical analysis the notion of partial information enters implicitly in every discussion of approximation and error estimation. However there one does not often directly manipulate intervals qua approximate real numbers; a notable exception is, of course, interval analysis [46] which influenced the early work by Scott on domain theory [57]. In this paper, we work directly with a cpo of intervals as computational approximations to real numbers.

A major part of the philosophy of concurrent constraint programming was "putting partial information in the hands of the programmer". In our setting we have captured both a qualitative notion of uncertainty, embodied in the use of partial information, as well as the evident quantitative notion associated with the use of probabilities.

1.3 Our Results

We describe Probabilistic cc, an executable stochastic concurrent constraint language, which adds a form of probabilistic choice to the underlying concurrent constraint programming paradigm, cc. Probabilistic cc is probabilistically determinate [47], rather than nondeterministic. Probabilistic cc is expressive, *e.g.* Probabilistic cc is expressive enough to encode (a variant of) Probabilistic Petri nets and probabilistic dataflow networks.

- We describe an operational semantics given by SOS rules and a notion of sampling.
- We show that a denotational semantics based on input-output relations cannot be compositional; and provide examples to illustrate the impact of the non-monotonic character of the language on (semantic) interpretations of recursion.
- We describe a denotational semantics based on weak bisimulation on constraint-labeled transition systems. It is necessary to work with weak (rather than strong) bisimulation in a programming language context. We show that weak bisimulation is a congruence and is an adequate semantics for reasoning about the input-output relation of processes.
- The denotational semantics when specialized to cc is fully abstract for cc. We show that Probabilistic cc is conservative over cc— thus our integration of probability is coherent with the underlying concurrent constraint paradigm.

Our techniques suggest that probability has a role in the suite of techniques at the disposal of concurrency theorists. As evidence, we show that our semantic study permits a new characterization of a problem stated without *any* reference to probability — the indeterminacy issues of synchronous programming.

Organization of paper. The next section describes a collection of examples that illustrate the design of the language and some of its subtleties. We follow with a description of the operational semantics. The next section describes a sketch of the denotational semantics and contains soundness and conservativity theorems. We conclude the paper with a comparison with related work.

In this extended abstract, we content ourselves with an overview of the results, and some key examples, and defer the detailed technical development to the full paper.

2 The language

2.1 Concurrent constraint programming

The concurrent constraint (cc) programming paradigm [56] replaces the traditional notion of a store as a valuation of variables with the notion of a store as a constraint on the possible values of variables. Computation progresses by accumulating constraints in the store, and by checking whether the store entails constraints. Several concrete general-purpose programming languages have been implemented in this paradigm [39, 60], including timed [53] and hybrid extensions [27] of cc that have been used for modeling physical systems [28, 29].

A salient aspect of the cc computation model is that programs may be thought of as imposing constraints on the evolution of the system. cc provides four basic constructs: (tell) c (for c a primitive constraint), parallel composition (A, B), positive ask (**if** c **then** A) and hiding (**new** X **in** A). The program c imposes the constraint c. The program (A, B) imposes the constraints of both A and B — logically, this is the conjunction of A and B. **new** X **in** A imposes the constraints of A, but hides the variable X from the other programs — logically, this can be thought of as a form of existential quantification. The program **if** c **then** A imposes the constraints of A provided that the rest of the system imposes the constraint c — logically, this can be thought of as intuitionist implication. This declarative way of looking at programs is complemented by an operational view. The basic idea in the operational view is that of a network of programs interacting with a shared store of primitive constraints. The program c is viewed as adding c to the store instantaneously. The program (A, B) behaves like the simultaneous execution of both A and B. **new** X **in** A starts A but creates a new local variable X which is inaccessible outside A. The program **if** c **then** A behaves like A if the current store entails c.

2.2 Constraint systems

cc languages are described parametrically over a *constraint system*[54, 55]. For technical convenience, the presentation here is a slight variation on earlier published presentations.

The information added to the store consists of primitive constraints which are drawn from a *constraint system*. A constraint system $\mathcal{D}$ is a system of partial information, consisting of a set of primitive constraints (first-order formulas) or *tokens* D, closed under finite conjunction and existential quantification, and an inference relation (logical entailment) $\vdash$ that relates tokens to tokens. $\vdash$ naturally induces logical equivalence, written $\approx$. Formally,

Definition 2.1 *A* constraint system *is a structure* $\langle D, \vdash, \mathbf{Var}, \{\exists_X \mid X \in \mathbf{Var}\}\rangle$ *such that:*

- *D is closed under conjunction($\wedge$); if $a, b, c \in D$, then $\vdash \subseteq D \times D$ satisfies:*
  - $a \vdash a$*;* $a \vdash b$ *and* $b \wedge c \vdash d$ *implies that* $a \wedge c \vdash d$
  - $a \wedge b \vdash a$ *and* $a \wedge b \vdash b$*;* $a \vdash b$ *and* $a \vdash c$ *implies that* $a \vdash b \wedge c$
- $\mathbf{Var}$ *is an infinite set of* variables*, such that for each variable* $X \in \mathbf{Var}$*,* $\exists_X : D \to D$ *is an operation satisfying usual laws on existentials:*
  - $a \vdash \exists_X a$
  - $\exists_X(a \wedge \exists_X b) \approx \exists_X a \wedge \exists_X b$
  - $\exists_X \exists_Y a \approx \exists_Y \exists_X a$
  - $a \vdash b \Rightarrow \exists_X a \vdash \exists_X b$

A *constraint* is an entailment closed subset of $\mathcal{D}$. The set of all constraints, denoted $|D|$, is ordered by inclusion, and forms an algebraic lattice with least upper bounds induced by $\wedge$. We will use $\sqcup$ and $\sqcap$ to denote joins and meets of this lattice. From now on we will use $a, b, c, \ldots$ to denote constraints, noting that a token a can be represented as the constraint $\{b \in D \mid a \vdash b\}$: such constraints are the *finite* constraints, as they are the finite elements in the lattice. $\exists$ lifts to an operation on constraints. In any implementable language, $\vdash$ must be decidable, and we also assume the the set $\mathcal{D}$ is *countable*.

Examples of such systems are the system Herbrand (underlying logic programming) and FD [36](finite domains).

In this paper we will work with a constraint system that includes interval constraints [46]. Primitive constraints are built up (by conjunction and existential quantification) from tokens of the form $z \in [l, u]$ where l, u are rationals: $z \in [l, u]$ constrains the variable z to lie in the closed interval $[l, u]$. The entailment relation on interval constraints is induced by inclusion of intervals — $z \in I \vdash z \in J$ if $I \subseteq J$.

In addition, our constraint system will also include signal constraints like On and S — these are 0-ary predicates, and telling such a constraint is like emitting a signal. $S_1 \wedge \ldots \wedge S_n \vdash T$ iff $S_i = T$ for some $i \in 1..n$.

2.3 Syntax

In this paper, we describe the integration of discrete random variables in cc. This paper extends the results of [26] with recursion. We augment the usual cc syntax with a **choose** construct, **choose** X **from** Dom **in** P. Thus the Probabilistic cc syntax is as follows:

$$\begin{array}{rcl}
Decl & ::= & \epsilon \mid g(X_1, \ldots, X_n) :: P \mid Decl, Decl \\
P & ::= & c \mid g(t_1, \ldots, t_n) \mid \textbf{if } c \textbf{ then } P \\
 & & \mid P, P \mid \textbf{new } X \textbf{ in } P \\
 & & \mid \textbf{choose } X \textbf{ from } Dom \textbf{ in } P \\
Prog & ::= & Decl, P
\end{array}$$

In this description, a $Decl$ is a set of procedure declarations, with g a procedure name and $X_1, \ldots, X_n$ a list of parameters. $t_1, \ldots, t_n$ is a list of variables or values. X is a variable and Dom is a finite set of real numbers.

2.4 Constraints on random variables

Random variables (RV) must be declared using the **choose** combinator. **choose** X **from** Dom **in** A models a fair coin toss that takes values from the set Dom — **choose** X **from** Dom **in** A reduces to A after choosing a value for X from Dom. Furthermore, this combinator has the scoping connotations of **new** X **in** A combinator— the variable X is a new local variable and no information on it can be communicated outside this scope. Thus our RVs satisfy:

RV Property 2.2

- *RVs are hidden,* i.e. *not part of the observable results of the program.*
- *Each RV is associated with a unique probability distribution.*

Example 2.3 *Consider the following program: it has one RV, X, with 4 equiprobable possible values.*

$$\begin{array}{l}
\textbf{choose } X \textbf{ from } \{0, 1, 2, 3\} \textbf{ in} \\
\quad [\textbf{if } X = 0 \vee X = 1 \textbf{ then } a, \textbf{if } X = 2 \textbf{ then } b]
\end{array}$$

On input `true`[1]*, this program will produce output a with probability* 0.5*, b with probability* 0.25 *or* `true` *with probability* 0.25*. Note that the visible outputs do not include X.*

Random variables, like any other variables, may be constrained. In fact this is a fundamental aspect of the entire framework. Each RV acquires a value according to some distribution and the choices are all made independently but the overall effect of the interaction between these choices is achieved by imposing constraints on the RVs. In particular if a choice is made which conflicts with a constraint then the inconsistent choices are discarded by the implementation and the probabilities get renormalized according to the consistent choices. Thus constraints may cause portions of the domains of the RVs to be eliminated. In such cases the renormalization of the result yields the *conditional probability* of a given output given the valid set of outputs. Relatively complex joint distributions

[1] Input `true` to a program in the cc context means running the program in the unconstrained store, *i.e.* without external input.

can be emulated by this mechanism. The encoding of Probabilistic Petri nets in Example 2.8 requires constraints on RVs, as does the encoding of synchronous programs in Example 2.13.

In order to understand how to interpret the probabilities reported in the formal semantics we think of execution in the following "Monte Carlo" style. We consider a program as standing for an ensemble of initially identical processes. Each process makes choices according to the indicated distributions as it evolves. As processes become inconsistent they are discarded from the ensemble. At the "end" the probability reported is the conditional probability of observing a given store conditioned on seeing a consistent store. This is the sampling view of the process which is what the semantics captures.

Example 2.4

$$\textbf{choose } X \textbf{ from } \{0,1,2,3\} \textbf{ in } [X \leq 2, \\ \textbf{if } X = 0 \vee X = 1 \textbf{ then } a, \textbf{if } X = 2 \textbf{ then } b]$$

On input `true`, this program will output a or b; `true` is not a valid output because of the constraint $X \leq 2$. a is associated with 0.5 and b with 0.25; however to compute the probabilities, we must normalize these numbers with the sum of the numbers associated with all outputs. This yields the probability $2/3$ for a and $1/3$ for b.

We make the following assumption.

RV Property 2.5 *The choices of values for different RVs are made independently.*

Correlations between random variables are established by constraints so we cannot just say that the RVs are independent.

Example 2.6

$$\textbf{choose } X \textbf{ from } \{0,1\} \textbf{ in } [X = z], \\ \textbf{choose } Y \textbf{ from } \{0,1\} \textbf{ in } [\textbf{if } z = 1 \textbf{ then } Y = 1]$$

There are a total of four execution paths. One of these paths ($X = 1, Y = 0$) gets eliminated because of inconsistent constraints on the random variable Y, leaving three valid execution paths. Normalizing the probabilities, we get the following paths (and associated probabilities): $X = 0, Y = 0, z = 0(1/3)$, $X = 0, Y = 1, z = 0(1/3)$, $X = 1, Y = 1, z = 1(1/3)$. Since the random variables are hidden, the visible outputs (and associated probabilities) are:

$$z = 0(2/3), \;\; z = 1(1/3)$$

The above examples show how to get any finite distribution with rational probabilities by just using fair coin tosses. For the rest of this paper, we will use the derived combinator: **choose** X **from** Dom **with distribution** f **in** P to indicate that the random variable X is chosen from Dom with distribution function given by f. Consider the rational version of the probabilistic choice operator of [40].

Example 2.7 *Let r be a rational. The probabilistic choice operator of [40], $P +_r Q$, reduces to P with probability r and to Q with probability $(1-r)$. This combinator can be defined in* Probabilistic cc *as:*

$$\textbf{choose } X \textbf{ from } \{0,1\} \textbf{ with distribution } \{r, 1-r\} \textbf{ in} \\ [\textbf{if } X = 0 \textbf{ then } P, \textbf{if } X = 1 \textbf{ then } Q]$$

where $\{r, 1-r\}$ represents the function $f(0) = r, f(1) = 1-r$. X is not free in P, Q. Since the random variables are hidden, we get the expected laws: $P +_r P = P$(absorption), $P +_r Q = Q +_{(1-r)} P$ (commutativity), $(P +_r Q) +_s R = P +_{rs} (Q +_{\frac{s(1-r)}{1-rs}} R)$(associativity).

Example 2.8 *We model the Probabilistic Petri nets described in [45, 62]. In these nets, places are responsible for the probabilistic behavior, while transitions impose constraints to ensure correct behavior. Nets are 1-safe, so a place may contain at most one token. Temporal evolution is discrete (modeled here by a recursive call). At each time tick, a place with a token chooses randomly a transition which it would like to send its token to, or choose not to send it anywhere. Similarly an empty place chooses which transition to accept a token from, or accept no token at all. A transition constraint ensures that either all preconditions and postconditions of a transition choose it, or none of them choose it. Then a new marking is computed, and the net starts over again. The program is shown in Figure 1. Note the use of constraints on RV's to ensure that only correct runs are generated.*

```
P(S, t) ::        /* S ⊆ [1..n] is the marking, t = time*/
  new T_1, ... T_n in {
    /* T_i will be the transition chosen by place i to send a
    token to or receive from. */
    /* Select a transition for each place */
    if (i ∈ S) then
      choose X from Pre_i with distribution f_i in T_i = X,
      /* Pre_i is the set of transitions of which i is a
      precondition. Also, 0 ∈ Pre_i, signifying no choice. */
    if (i ∉ S) then
      choose X from Post_i with distribution g_i in T_i = X,
      /* As above, Post_i is the set of transitions of which i
      is a postcondition, and 0 ∈ Post_i. */
    ... ,

    /* Transition constraints — one for each transition.
    Contraint for transition i. i_1, ... , i_k are its pre and
    postconditions. If one of its pre or postconditions
    chose transition i, all must choose it. */
    if (T_{i_1} = i ∨ ... ∨ T_{i_k} = i) then
      {T_{i_1} = i, ... , T_{i_k} = i}
    ... ,

    /* Compute the next marking */
    new newS in {
      if ((1 ∈ S ∧ T_1 = 0) ∨ (1 ∉ S ∧ T_1 ≠ 0)) then
        1 ∈ newS,
      if ((1 ∉ S ∧ T_1 = 0) ∨ (1 ∈ S ∧ T_1 ≠ 0)) then
        1 ∉ newS,
      ... ,

      P(newS, t + 1)          /* next instant */
    }
  }
```

Figure 1: Probabilistic Petri nets as Probabilistic cc programs.

2.5 Recursion and limit computations

Recursion increases the expressiveness of our language by allowing the programming of continuous distributions.

Example 2.9 *Consider the program*

$$U(l, u, z) :: \;\; z \in [l, u], \\ \textbf{choose } X \textbf{ from } \{0,1\} \textbf{ in } [\\ \textbf{if } X = 0 \textbf{ then } U(l, (u+l)/2, z), \\ \textbf{if } X = 1 \textbf{ then } U((u+l)/2, u, z)]$$

This program can be visualized as a full binary branching tree, where the branches correspond to the two possible values for the random variable of each recursive call. Each internal node of the tree can be associated with the interval that z has been constrained to be in. Thus, in the limit, at the leaves of the tree, z gets constrained to be a real number. Furthermore, the induced probability distribution is on infinite binary sequences satisfying the condition that each finite sequence has the same probability of being extended by a 0 or by 1. By classical results, this is the uniform random distribution on z over $[0, 1]$(see [4] Page 195, Prob. 8). In the rest of this paper, we will use U to stand for the program defined in Example 2.9.

The following example conveys the flavor of use of probability in the programming of large arrays of simple devices. This example satisfies all the properties required of an amorphous computer [19].

Example 2.10 *Suppose we have a large array of tiny light emitting diodes (LEDs). Assume that each LED can be switched on or off. We would like to produce light of a certain intensity. One method is to switch on a fraction of the LEDs by telling each LED to switch on or off. This method requires each LED to have a unique identity. A more efficient way is to let each LED light up with a certain probability. The central limit theorem [4] ensures that the intensity will be proportional to the probability. The probability can be communicated to the LEDs via an electric potential, thus replacing individual messages by a broadcast communication.*

$$\textbf{new } X \textbf{ in } [U(0,1,X),\ \textbf{if } (X < Potential) \textbf{ then } On]$$

A clever implementation would need to unwind the recursion in U only finitely (almost always). Furthermore, this method allows one to compensate for a broken fraction of LED's by increasing the potential suitably.

How do we compute probabilities in the presence of recursion? We use a computationally motivated analogy to the limiting process [59] that computes conditional probabilities in measure theory. This subtle limiting process plays a key role in our theory. The important point here is that we can have situations where the probability of an inconsistent store is 1, i.e. the conditional probability is undefined according to classical measure theory. In our theory we can sometimes define these probabilities *by taking into account the way in which the computational approximation proceeds.*

We begin with a seductively simple warm-up example that illustrates our techniques.

Example 2.11 *Consider* $P = [U(0, 1, z), z = 0]$

Intuitively we would expect its output to be $z = 0$ with probability 1, since after all the *only* possible output of this program is $z = 0$. Note however, that the probability of $z = 0$ in the above program is 0, so a naive calculation at normalization time leads us to calculate $0/0$.

We handle this problem by computing the required probabilities for a recursive program as a limit of the probabilities of its finite unwindings. Consider the program $P_n = [U_n, z = 0]$ got by replacing the recursion in U by a finite unwinding U_n. U_n can be viewed as a finite subtree of the tree associated with U, see example 2.9. P_n is a finite program operating on a finite discrete probability distribution. Each of these finite programs yield output $z = 0$ with probability 1. Thus, the result is $z = 0$ with probability 1.

Apropos, this example also illustrates the *need* for these techniques — the above problem arises anytime a random variable is constrained. The limiting process yields expected answers in normal situations.

Example 2.12 *Consider* $P = U(0, 1, z), \textbf{if } (1/4 < z < 3/4) \textbf{ then } S$ *where* S *is a signal.*

Let us compute the probability of S being present — we expect the answer to be $1/2$. Consider a finite unwinding U_n of U; the required answer for this unwinding is the sum of the sizes of the finite intervals contained completely in the interval $(1/4, 3/4)$. Consider the set of real numbers that are answers deduced at the finite unwindings. The only limit point of this set of reals numbers is $1/2$. This can be visualized as follows. Take *any* directed set of finite subtrees of the tree associated with U (we are thinking of trees under the prefix ordering on trees), whose limit is U. Then, the limit of the probabilities associated with this directed set is $1/2$.

The above example tells us when probability numbers are well defined — any way of unwinding the various recursions in a program should yield the same answer. Do limits exits always? Unfortunately not! These subtleties are illustrated by the next set of examples that deal with program combinators reminiscent of indeterminacy issues in synchronous programming languages [30, 8, 12, 31, 25, 33, 53].

Example 2.13 *Consider*

$$\textbf{new } X \textbf{ in } [U(0,1,X),\ \textbf{if } a \textbf{ then } X = 0, \textbf{if } X > 0 \textbf{ then } b]$$

This program can be thought of as **if** a **else** b, *i.e.* if a is not present, produce b with probability 1. If a is present, b is *not* produced. In essence, we use the RV to be certain that b will be produced; however, this expectation can be denied on the production of a.

The analysis of this program proceeds as follows. On input `true`, the program produces constraint b with probability 1 and `true` with probability 0. On input a the only output of this program is a. Indeed in this latter case, computation of the probability proceeds similar to Example 2.11 — the probability prior to normalization is 0, which seems to lead to a $0/0$ case — however, our analysis tells us that the probability is 1.

The next example shows how indeterminacy issues in synchronous programming languages show up as problems with the limit computation of probabilities. For example, the program **if** a **else** b, **if** b **else** a has two potential outputs a and b on input `true`[2].

Example 2.14 *Consider* $P = \textbf{if } a \textbf{ else } b, \textbf{if } b \textbf{ else } a.$

A simple calculation shows that on input `true` the outputs are a, b or `true`, with all non-normalized probabilities being 0. However, in this case the limiting process does not produce unique answers. The formal calculation proceeds as follows.

Consider approximations **if** a **else**$_n b$ to **if** a **else** b. **if** a **else**$_n b$ stands for the program got by unwinding the recursion in U n times. A simple calculation shows that **if** a **else**$_n b$ behaves as follows: if a is not present, produce b with probability $(1 - 2^{-n})$. If a is present, b is *not* produced.

Now consider approximations $P_{m,n} = \textbf{if } a \textbf{ else}_m b, \textbf{if } b \textbf{ else}_n a$ to P. On input `true` the non-normalized probabilities of a and b are $2^{-m}(1 - 2^{-n}), 2^{-n}(1 - 2^{-m})$. Now the limits of the normalized probabilities (resp. $\frac{2^n-1}{2^m+2^n-1}, \frac{2^m-1}{2^m+2^n-1}$) depend on the relative rates of m, n approaching ∞, *i.e.* the limits depend on the "speed of unwinding" of the two recursions involved. In our theory, this program would thus not have a defined result.

[2]Hence is rejected by the compilers for synchronous programming languages.

3 Operational semantics

We first describe a SOS style transition system for finite (recursion-free) Probabilistic cc programs (as in [26]), and follow it with a formalization of the limit constructions (alluded to earlier) for handling recursion.

3.1 Transition relation of recursion-free programs.

We follow the treatment of cc. We assume that the program is operating in isolation — interaction with the environment can be coded as an observation and run in parallel with the program. A configuration is a multiset of programs Γ. $\sigma(\Gamma)$ will denote the store in the multiset — it is recoverable as the conjunction of the (tell) primitive constraints in Γ.

$$\frac{\sigma(\Gamma) \vdash a}{\Gamma, \textbf{if } a \textbf{ then } B \longrightarrow \Gamma, B}$$

$$\Gamma, (A, B) \longrightarrow \Gamma, A, B$$

$$\Gamma, \textbf{new } X \textbf{ in } A \longrightarrow \Gamma, A[Y/X] \quad (Y \text{ not free in } \Gamma)$$

$$\Gamma, \textbf{choose } X \textbf{ from } Dom \textbf{ in } A \longrightarrow \Gamma, Y = r, A[Y/X]$$
$$r \in Dom, Y \text{ not free in } \Gamma$$

Consider the finite set of consistent quiescent configurations of A on any input a, *i.e.* $\{\Gamma_i \mid A, a \longrightarrow^* \Gamma_i \not\longrightarrow, \sigma(\Gamma_i) \not\approx \texttt{false}\}$. The unnormalized probability of Γ_i is determined by the RVs in $\sigma(\Gamma_i)$ — the transition sequence(s) to Γ_i do not play a role in the calculation. The unnormalized probability is $\Pi_{\vec{Y}} 1/|Dom_Y|$ where $\vec{Y}$ is the set of RVs, $Y = r_Y \in \Gamma_i$, and Dom_Y is the domain of Y.

The (finite set of) outputs of a process P on an input a, denoted $\texttt{OpsemIO}(P, a)$ is given by hiding the random variables and new variables in the set of $\sigma(\Gamma_i)$. The probability of an output o, written $Pr(P, a, o)$ is computed as follows:

- For each output, compute the unnormalized probability by adding the probabilities of all configurations that yield the same output.
- Normalize probabilities of the set of outputs.

Define

- $\overline{Pr}(P, a, o) = \sum\{Pr(P, a, o') \mid o' \in \texttt{OpsemIO}(P, a), o' \subseteq o\}$. $\overline{Pr}(P, a, o)$ is a cumulative probability interpreted as the probability that the output of P on a is at most o. Pr can be recovered from $\overline{Pr}$ by an inclusion–exclusion principle.
- $\underline{Pr}(P, a, o) = \sum\{Pr(P, a, o') \mid o' \in \texttt{OpsemIO}(P, a), o \subseteq o'\}$. $\underline{Pr}(P, a, o)$ is a cumulative probability interpreted as the probability that the output of P on a is at least o. Pr can be recovered from $\underline{Pr}$ by an inclusion–exclusion principle.

3.2 Handling recursion.

We first generate all syntactic finite approximations to the given program. Each of these finite programs is executed using the above transition relation. We then describe the limit calculation for the probabilities.

The partial order, $(\text{App}(P), \preceq_P)$ describes the syntactic approximations of a Probabilistic cc program P. $\preceq_P$, the partial ordering is intended to capture refinement of approximation. App(P) is defined by structural induction on P. For a recursive program, App(P) is constructed by considering the set of approximations of all finite unwindings with ordering induced by the (Ω) match ordering on recursive approximations [44], i.e. an expression is refined by replacing the "least program" (in our case $\texttt{true}$) by an expression. For a sample of other cases:

$$\text{App}(c) = \{c\}$$
$$\text{App}(A_1, A_2) = \{(A'_1, A'_2) \mid A'_i \in \text{App}(A_i)\},$$
ordering induced by the cartesian product of $\preceq_{A_i}$

App(P) has the expected properties.

Lemma 3.1 *App(P) is a directed set.*

Let c be an input. Then, c induces a function on App(P) that maps $A_i \in \text{App}(P)$ to $\texttt{OpsemIO}(A_i)(c)$. If $A_i \preceq_{\text{App}(P)} A'_i$, then $(\forall d'_i \in \texttt{OpsemIO}(A'_i)(c))\ (\exists d_i \in \texttt{OpsemIO}(A_i)(c)) d_i \subseteq d'_i$. The outputs of P on c are determined by a limit computation — d is an output of P on c if, there exists a monotone function F_d mapping App(P) to constraints such that $F_d(A_i) \in \texttt{OpsemIO}(A_i)(c)$, and $d = \bigcup_i F_d(A_i)$. [3]

We now turn to computing the probability numbers. Let d be an output of P on c. Then, d maps each $A_i \in \text{App}(P)$ to the cumulative probability $\overline{Pr}(A_i, c, d)$. Since App(P) is a directed set, we use the standard definition of convergence (e.g. see [4], page 371). The cumulative probability of the output d equals r if:

$$(\forall \epsilon)\ (\exists A \in \text{App}(P))\ (\forall A')\ [A \preceq A' \Rightarrow |\overline{Pr}(A', c, d) - r| < \epsilon]$$

V may not converge in general. We say that the output is defined for the program P on a given input c, if the cumulative probability is defined for all outputs of P on c.

We first explore the relationship of the above definition to the two notions of cumulative probabilities discussed in the preceding subsection.

Relationship to $\overline{Pr}$ and $\underline{Pr}$

Example 3.2 *Let P be a finite process. Then the set of outputs given by the above definition on input c is $\texttt{OpsemIO}(P, c)$; the output is defined on c and the definition yields $\overline{Pr}(P, c, d)$ for all $d \in \texttt{OpsemIO}(P, c)$.*

The inclusion–exclusion style argument underlying the recovery of probability (Pr) from cumulative probability ($\overline{Pr}$) can be used for programs that satisfy the following countability condition — for any input c, any output d satisfies the condition that the cardinality of the subset of outputs $e \subseteq d$ is countable. This condition is satisfied by *all* our earlier examples.

Example 3.3 *On Example 2.12 for input* $\texttt{true}$*, the definition yields cumulative probability* 1 *for output S,* 0.5 *for output* $\texttt{true}$*; thus yielding* 0.5 *for both probabilities.*

Let e be a constraint. Then, e induces a function that maps $A_i \in \text{App}(P)$ to $\underline{Pr}(A_i, c, e)$. Since App($P$) is a directed set, we can once again use the standard definition of convergence and define $\underline{Pr}(P, c, e)$ equals r if:

$$(\forall \epsilon)\ (\exists A \in \text{App}(P))\ (\forall A')\ [A \preceq A' \Rightarrow |\underline{Pr}(A', c, e) - r| < \epsilon]$$

Again, convergence is not assured. We have the following relationships between $\underline{Pr}(P, c, e)$ and the cumulative probability of outputs of P on c.

[3] We can prove the following fact: There is a monotone function F_d satisfying the properties above, such that $d = \bigcup_i F_d(A_i)$ iff there is a derivation starting from A whose output is d, where the output of an infinite derivation is the lub of the outputs of its prefixes.

- Let P be a finite process. Then, the above definition of $\underline{Pr}(P, c, e)$ agrees with the definition of $\underline{Pr}$ from the preceding subsection.

- Let P be a program such that its output is defined on input c. Furthermore, let $\underline{Pr}(P, c, e)$ be defined for all finite e. Then, the cumulative probability $\overline{Pr}(P, c, d)$ of an output d can be recovered from the $\underline{Pr}$ information by an inclusion–exclusion principle on the (finite) constraints e such that $d \not\supseteq e$.

Interpreting the probability numbers. How are these probability numbers to be interpreted? Recall that we say that there is an *ensemble* of processes each of which executes a copy of the Probabilistic cc program. The probability numbers are interpreted *statistically* with respect to this ensemble, *conditional* on the process being consistent. The next few examples illustrate what we have in mind.

Example 3.4 *Let P be a* Probabilistic cc *program in which random variables are "read only",* i.e. *no constraints are imposed on random variables; they are only queried in asks. (This class of programs includes all programs from [42].) In this case, the function $\overline{Pr}(,c, d)$ as defined above is a monotone, decreasing, bounded (by 0) function on App(P). Thus, $\overline{Pr}(,c, d)$ converges, and the output of P is defined for all inputs.*

Recall example 2.14 for examples of non–convergence. Let A be a Default cc ("synchronous", [53]) program and A' be a Probabilistic cc program obtained from A via the definition for **else** from example 2.13. Then,

Theorem 3.5 *If A has multiple outputs on an input c, the output of A' is not defined for c. If A is determinate, the output of A' on any input c is the the output of A on c with probability 1.*

The next couple of examples illustrate the fact that our theory agrees with the answers produced by standard probability theory, when the standard theory produces defined answers.

Example 3.6 *Let $f : [a, b] \longrightarrow [c, d]$ be a Riemann integrable function. Consider the program*

$$\begin{aligned} P ::\ & U(a, b, X), U(c, d, Y), \\ & \textbf{if } (Y < f(X)) \textbf{ then } A, \\ & \textbf{if } (Y > f(X)) \textbf{ then } B \end{aligned}$$

What is the probability of the output A? Standard probability theory tells us that if $R = (b - a) * (d - c)$, it should be $(1/R) \times \int_a^b (f - c)$. Now consider the set of approximations to P. Each approximation defines a partition on $[a, b]$ and a partition on $[c, d]$, and successive approximations refine these partitions. Thus the probability number for signal A corresponds to a lower Riemann sum, and similarly the probability number for B corresponds to an upper Riemann sum. It is now easy to show that $P(A) \longrightarrow \frac{1}{R}\underline{\int}_a^b (f - c)$ and $P(B) \longrightarrow 1 - \frac{1}{R}\overline{\int}_a^b (f - c)$, thus our computations agree with standard probability theory. These results extend to functions of many variables; and U can be replaced by any program that produces the uniform distribution, such as V in Example 3.9 below.

Example 3.7 *The following program emits the signal $notC$ if z is not an element of the Cantor set.*

$$\begin{aligned} C(l, u, z) ::\ & C(l, (u + 2l)/3, z), \\ & C((2u + l)/3, u, z), \\ & \textbf{if } ((u + 2l)/3 < z < (2u + l)/3) \textbf{ then } notC \end{aligned}$$

Now, consider the program:

$$P ::\ U(0, 1, X), C(0, 1, X)$$

The probability of the output $notC$ as per our theory is 1, in conformance with standard probability theory.

In the case that a constraint forces the choices on a random variable to be inconsistent with probability 1 we get a situation in which conventional probability theory has no answer. This is the situation with examples 2.13 and 2.14. Standard probability theory would say that if we condition with respect to a set of measure zero the resulting conditional probability is undefined — this is true both for discrete (countable) systems, and for continuous state spaces, where the conditional probabilities are computed based on some notion of derivative of measures, for example by use of the Radon-Nikodym theorem [4, 59]. We can however ascribe a limiting probability *based on the structure of the computational process*. This additional information allows us to associate sensible probabilities in situation where conventional theory leaves the numbers undefined. Perhaps a better way of saying it is that probability theory leaves the possibility of augmenting the information to come up with a sensible conditional probability. The limit formula above is defined to capture exactly the *computational information* that goes into the definition.

Example 3.8 *Consider:*

$$\textbf{new } X \textbf{ in new } Y \textbf{ in } [U(0, 1, X), U(0, 1, Y), X = Y]$$

This should and does yield a uniform distribution on the "diagonal" of the unit $X - Y$ square. This is intuitively plausible but completely at variance with probability theory. That is at it should be. If one were to say that there are two independent uniform distributions on X and Y then the question "what is the distribution given that X and Y are equal?" is not answerable. In our case *the operational model shows how the uniform distribution is obtained by successive approximation.* The calculation based on the limit formula is exactly the embodiment of this idea.

It is important to not mistake the intent of the above discussion. *The numbers that we compute do not depend on the details of the execution.* Thus in the example just above we are claiming that even if the recursions in the two calls to U are unwound at very different rates we get the same answer. An explicit calculation verifies this easily. However if we took two *different* processes both generating the uniform distribution we get a very different answer.

Example 3.9 *Consider a variant of the program U:*

$$\begin{aligned} & V(l, u, z) :: \\ & \quad z \in [l, u], \\ & \quad \textbf{choose } X \textbf{ from } \{0, 1\} \textbf{ with distribution } \{1/3, 2/3\} \textbf{ in } [\\ & \qquad \textbf{if } X = 0 \textbf{ then } V(l, (2l + u)/3, z), \\ & \qquad \textbf{if } X = 1 \textbf{ then } V((2l + u)/3, u, z)] \end{aligned}$$

This program chooses 0 with probability $1/3$ and 1 with probability $2/3$. It then subdivides the interval into two unequal portions — one twice the size of the other — and recursively calls itself. This also produces the uniform distribution in the limit. Now consider

$$U(X, 0, 1), V(Y, 0, 1), X = Y$$

We do *not* get the uniform distribution along the diagonal. It is easy to verify that the distribution assigns to the subinterval $(0, 0.1)$ a smaller probability than it does to the interval $(0.9, 1)$. The actual distribution is quite fascinating (see [14, pp. 408]) but does not concern us further here. What is important is the fact that we got two very different answers to the question "if X and Y are uniformly distributed, what is the distribution given that X and Y are equal?" when the programs generating the uniform distributions are different. Here we see why the conventional probability theory answer is sensible; without further information one can get an almost arbitrary answer. However our semantics provides exactly this additional information.

4 Denotational semantics

4.1 CC semantics

We begin with a brief review of the model for cc programs, referring the reader to [54] for details. In our treatment of cc, we will consider *finite programs* first. We describe recursion (in parenthetical remarks) by working with the sets of programs obtained by unwinding the recursion. An observation of a cc program A is a store u in which it is quiescent, i.e. running A in the store u adds no further information to the store. Formally we define the relation $A \downarrow^u$, read as A *is quiescent on* u, with the evident axioms:

$$\frac{A_1 \downarrow^u \quad A_2 \downarrow^u}{(A_1, A_2) \downarrow^u} \qquad \frac{A \downarrow^v \quad \exists_X u = \exists_X v}{(\textbf{new } X \textbf{ in } A) \downarrow^u} \qquad \frac{c \in u}{c \downarrow^u}$$

$$\frac{c \notin u}{(\textbf{if } c \textbf{ then } A) \downarrow^u} \qquad \frac{A \downarrow^u}{(\textbf{if } c \textbf{ then } A) \downarrow^u}$$

The denotation of a program A can be taken to be the set of all u such that $A \downarrow^u$. The semantics is compositional since the axioms above are compositional. The output of A on any given input a is now the least u containing a on which A quiesces. (If A does not become quiescent on a, which might happen with a recursive program, then we have to take the lub of the stores that are produced by the finite unwindings of A on a.)

4.2 The problems

We turn now to Probabilistic cc. The input–output relation is not compositional. The following example is inspired by Russell's simplified version [52] of the Brock-Ackermann anomaly [16].

Example 4.1 *Consider the programs*

$$\begin{aligned}
A_1 &= S_1, \textbf{if } R_1 \textbf{ then } S_2, \textbf{if } R_1 \wedge R_2 \textbf{ then } S_3\\
A_2 &= S_1, \textbf{if } R_1 \textbf{ then } S_2, \textbf{if } R_1 \wedge R_2 \textbf{ then } S_4\\
A_3 &= \textbf{if } R_1 \textbf{ then } S_1, \textbf{if } R_1 \wedge R_2 \textbf{ then } S_2 \wedge S_4\\
A_4 &= \textbf{if } R_1 \textbf{ then } S_1 \wedge S_2, \textbf{if } R_1 \wedge R_2 \textbf{ then } S_4\\
A_5 &= \textbf{if } R_1 \textbf{ then } S_1, \textbf{if } R_1 \wedge R_2 \textbf{ then } S_2 \wedge S_3
\end{aligned}$$

S_i, R_i are signals (see example 2.10). Define:

$$\begin{aligned}
B_1 &= \textbf{if } X = 1 \textbf{ then } A_1, \textbf{if } X = 2 \textbf{ then } A_2,\\
&\quad \textbf{if } X = 3 \textbf{ then } A_3, \textbf{if } X = 4 \textbf{ then } A_4\\
B_2 &= B_1, \textbf{if } X = 5 \textbf{ then } A_5
\end{aligned}$$

$$\begin{aligned}
P_1 &= \textbf{choose } X \textbf{ from } \{1,2,3,4\}\\
&\quad \textbf{with distribution } \{0.2, 0.1, 0.5, 0.2\} \textbf{ in } B_1\\
P_2 &= \textbf{choose } X \textbf{ from } \{1,2,3,4,5\}\\
&\quad \textbf{with distribution } \{0.1, 0.2, 0.4, 0.2, 0.1\} \textbf{ in } B_2
\end{aligned}$$

P_1 and P_2 have identical input/output behavior. Key cases are:

Input	*Output*
`true`	$\texttt{true}(0.7), S_1(0.3)$
R_1	$S_1(0.5), S_1 \wedge S_2(0.5)$
$R_1 \wedge R_2$	$S_1 \wedge S_2 \wedge S_3(0.2), S_1 \wedge S_2 \wedge S_4(0.8)$

*Let $Q =$ **if** S_1 **then** R_1, **if** S_2 **then** R_2. P_1, Q on input* `true` *outputs $S_1 \wedge S_2 \wedge S_3(0.2), S_1 \wedge S_2 \wedge S_4(0.1), \texttt{true}(0.7)$; P_2, Q input* `true` *outputs $S_1 \wedge S_2 \wedge S_3(0.1), S_1 \wedge S_2 \wedge S_4(0.2), \texttt{true}(0.7)$. Thus P_1 and P_2 can be distinguished by the context Q.*

Our solution to this problem is to make Probabilistic cc and cc amenable directly to some of the standard techniques of concurrency theory via a key technical innovation, CLTS (constraint labeled transition systems) with weak bisimulation, described in greater detail below.

Unfortunately, more problems loom. The mixture of probabilities and constraints violates basic monotonicity properties needed in standard treatments of recursion — these problems were indicated by the synchronous programming examples.

Example 4.2 *Consider*

$$\begin{aligned}
&P :: \textbf{choose } X \textbf{ from } \{0,1\} \textbf{ with distribution}\\
&\qquad\qquad \{0.2, 0.8\} \textbf{ in } [X = z, Trim(1)]\\
&Trim(Y) ::\\
&\quad Trim(1 - Y),\\
&\quad \textbf{if } z = Y \textbf{ then}\\
&\qquad \textbf{choose } X \textbf{ from } \{0,1\} \textbf{ with distribution}\\
&\qquad\qquad \{0.9375, 0.0625\} \textbf{ in } X = 1
\end{aligned}$$

Unwinding $Trim$ 0 times yields the approximation $P_0 = (z = 0) +_{0.2} (z = 1)$ to P. Unwinding $Trim$ once yields the approximation $P_1 = (z = 0) +_{0.8} (z = 1)$ to P. Unwinding $Trim$ twice yields the approximation P_0 again. In a standard monotonic least fixed point treatment of recursion, the denotation of the n'th unwinding is less than the denotation of the $(n+1)$'st unwinding for the purported order $\sqsubseteq$ on programs. Thus, $P_0 \sqsubseteq P_1 \sqsubseteq P_0$, forcing $P_0 = P_1$, an equation that is unsound.

The subtle interaction of recursion and normalization underlying this example is not handled by our study. We however show how to handle the issue of normalization in the case of recursion free programs.

4.3 CLTS

We begin with the intuitions underlying our definitions. Let P be a process. A transition system with transitions given by the operational semantics of cc is informally given in the following way. Each state of the system intuitively corresponds to a constraint store. There is an initial state, s_0, intuitively this corresponds to the store (`true`). The transitions are labeled in one of three ways: τ, $c!$ and $c?$ where c is a finite constraint. The $c!$ and $c?$ labels represent interactions with the environment and are deterministic, *i.e.* any state has at most one transition labeled with a $c!$ or $c?$. A transition $s \xrightarrow{c!} s'$ means that the system outputs the constraint c to the environment and adds it to the store. If c was already in the store, s' would be s, as the system can always output any constraint already in the store. A transition $s \xrightarrow{c?} s'$ means that the system reads the finite constraint c from the environment and adds it to the store. If the system contains a top-level **choose**, the actual choice will not be visible to the environment and the transition is labeled with τ and the probability.

CLTS closure conditions

make them a probabilistically determinate system in the context of cc computation.

- **Probabilistic transitions:** Only τ-transitions can be associated with probabilities, and the associated probability must be strictly positive.

- **Acyclicity:** The only cycles in the transition system are 1-cycles of non-probabilistic transitions. All states are reachable from the start state.
- **Determinacy**: for every state s and every label $c?$ or $d!$ there is at most one transition from s with that label.
- **Receptivity:** In every state s and for every finite constraint c there is a transition labeled $c?$.
- **Commutativity:** If there is a state s_1 and transitions $s_1 \xrightarrow{c} s_2$, $s_1 \xrightarrow{d} s_3$, where at least one of c, d is a non-probabilistic transition, then there is a state s_4 and transitions $s_2 \xrightarrow{d} s_4$, $s_3 \xrightarrow{c} s_4$.
- **Masking:** If there is a transition $s \xrightarrow{c?} s'$ then any transition of the form $s \xrightarrow{c!} s''$ has $s' = s''$. Masking is needed for the (forthcoming) definition of parallel composition.
- **Splitting**: if c and d are constraints and we have a transition $s \xrightarrow{(c \sqcup d)!} s'$ then there is a state s'' and transitions $s \xrightarrow{c!} s''$ and $s'' \xrightarrow{d!} s'$.
- **Transitivity**: if $s \xrightarrow{c?} s'$ and $s' \xrightarrow{d?} s''$ then there is a transition $s \xrightarrow{(c \sqcup d)?} s''$. Also, if $s \xrightarrow{c!} s'$ and $s' \xrightarrow{d!} s''$ then there is a transition $s \xrightarrow{(c \sqcup d)!} s''$.
- **Saturation**: if there is a transition $s \xrightarrow{c?} s'$, then there are transitions $s' \xrightarrow{d!} s'$ for every d such that $c \supseteq d$. Thus every query resolved by the constraint solver is represented by an action in the transition system.
- **Tau-finiteness condition:** There are at most finitely many τ transitions from any state.

The sole source of non–confluence in a CLTS is the (possible) non-commutativity of conflicting probabilistic transitions — non-probabilistic transitions commute with all other transtions. It also ensures that adding new information cannot disable an already enabled transition, and ensures the content of Lemmas (3.5,3.6) of [54]. These conditions ensure that the set of output constraints leading out of a state forms a directed set; and ensure that the set of output constraints in the labels on all the transitions between any two states forms a directed set.

Path-equivalence: A CLTS comes equipped with a state-indexed equivalence relation $\equiv_s$ on paths starting from the same state s in a CLTS: intuitively two paths are equivalent if they agree on the random variables. Each equivalence class comes associated with a probability, corresponding to the choice of the random variables. $\equiv_s$ satisfies conditions such as:

- If t is a non-probabilistic transition and $p \cdot t$ is a path starting from s, then $p \cdot t \equiv_s p$.
- If paths p_1, p_2 both start in s and end in the same state s', then $p_1 \equiv_s p_2$.

We say a CLTS is finite if intuitively it encodes only finitely many probabilistic choices. Formally, we say a CLTS is finite if:

- the number of equivalence classes in $\equiv_{s_0}$, where s_0 is the start state, is finite; and
- there are no paths with infinitely many occurrences of τ

For a finite CLTS, the number of equivalence classes in any state is finite, and the probability of any equivalence class is strictly positive. For the purposes of this paper, we restrict our attention to finite CLTSs.

Example 4.3 *The CLTS for* `true` *is constructed as follows. We will assign labels to the states for notational convenience. For each constraint c, there is a state with label c. The start state is the state with label* `true`*. A state with label c has self-loops labeled $d!$ for all finite constraints d such that $c \supseteq d$. There is transition labeled $e?$ from a state with label c to a state with label d if d is equivalent to $c \wedge e$. For any state s, $\equiv_s$ consists of one equivalence class consisting of all paths starting from s. This class has probability* 1.

Example 4.4 *The CLTS for a program that emits c is obtained from the CLTS for* `true` *as follows. For each transition with label $d?$ such that $c \supseteq d$, add a transition labeled $d!$ with the same source and target states.*

The following example extends the ideas of examples 4.4 and 4.4 to all determinate **cc** programs.

Example 4.5 *Let $(E, \sqsubseteq)$ be an algebraic lattice in which* `false` *is a compact element. Then, any continuous closure operator f*[4] *can be encoded as a CLTS as follows. Start with a copy of the CLTS for* `true`*. Close the CLTS under the addition of following transitions: add a transition $d!$ from a state with label c to a state with label $c \bigsqcup d$ if $d \sqsubseteq f(c)$.*

Example 4.6 *The CLTS for the program $c +_{0.5} d$ (emit one of c or d with probability 0.5 each) is as follows. The set of states is the disjoint union of the set of states of c, d (as given in Example 4.4) and the set of states of a copy of* `true`*(from Example 4.3). Each component of the disjoint union retains all its transitions. To each state, say with label e coming from the copy of the CLTS for* `true`*, add two probabilistic τ transitions each with probability 0.5, with target the states with the same label in CLTSs for c and d.*

The states s coming from the CLTS for c (resp. d) retain the associated $\equiv_s$. For each state s coming from the copy of `true`*, there are the following three equivalence classes in $\equiv_s$. These three equivalence classes correspond to the three possible cases of the probabilistic choice.*

1. *Choice not resolved: this equivalence class is the set of paths starting that only visit states from the copy of* `true` *and has probability number* 1.
2. *Choice resolved in favor of c: this class is the set of paths that visit some state from the copy of the CLTS for c and has probability number* 0.5.
3. *Choice resolved in favor of d: this class is the set of paths that visit some state from the copy of the CLTS for d and has probability number* 0.5.

Consistent states: A consistent state of a CLTS is intuitively a state which does not already entail inconsistency. The inconsistency of such a state is witnessed by a "maximal" path that does not have a `false`! labeled transition. We formalize this idea below.

Let P be a path from state w_0. Let the transitions in P be $t_0, t_1 \ldots$ with transition t_i (of label e_i) going from state w_i to state w_{i+1}. Then, a path P' from w_0 is a τ extension of P if $P' \neq P$ and the following hold: (1) the transitions in P' are $\tau, t'_0, t'_1, \ldots$, with t'_i going from w'_i to w'_{i+1}, (2) t'_i has the same label as t_i, and

[4] $f : E \rightarrow E$ is a closure operator if f is monotone, $x \sqsubseteq f(x)$ and $f(f(x)) = f(x)$.

(3) $\forall i$ there is a transition labeled τ from w_i to w'_i. A path P, all of whose suffixes have no τ extension, is called τ–maximal.

Let s be a state of a CLTS. We say that s is **consistent** if there is a path from s which satisfies the following: (1) all transitions on the path are labeled ! or τ, (2) the path is τ–maximal (3) if any $c!$ transition is enabled on the path, a $d!$ transition is taken for some $d \supseteq c$ (4) `false`! does not occur on the path. A state that is not consistent is called **inconsistent**. Note that the closure conditions ensure that any state reachable from an inconsistent state remains inconsistent.

Example 4.7 *In the CLTS for* `true` *and c, the only inconsistent state is the one with label* `false`*. In the CLTS for c* $+_{0.5}$ *d, the only inconsistent states are the three states with labels* `false`.

A state s of a CLTS is *observable* if s is the start state, or s is a consistent state with no outgoing τ transitions.

Finite CLTS IO. Let O be the set of paths from the start state s_0, which satisfy the following: (1) the first transition on the path is labeled ?, all others are labeled ! or τ, (2) the path is τ–maximal (3) if any $c!$ is enabled on the path, a $d!$ transition is taken for some $d \supseteq c$ (4) `false`! does not occur on the path (5) no prefix of the path is in O.

Now for an input a, consider the subset $O(a)$ of O whose paths start with $a?$. The output of each path is $\bigsqcup c$, for all $c!$ on the path. This is the set of outputs on a. The probability of an output o is $\sum\{Prob(Q) \mid Q \in \equiv_{s_0}, \exists p \in Q \cap O(a), \text{output of } p = o\}$ normalized by $\sum\{Prob(Q) \mid Q \in \equiv_{s_0}, O(a) \cap Q \neq \emptyset\}$.

Weak bisimulation. A path P in a CLTS has label $(c?)$(resp. $c!$) if it is of the form $\tau^*(c_1?)\tau^*(c_2?)\tau^* \ldots (c_n?)\tau^*$ (resp. $\tau^*(c_1!)\tau^*(c_2!)\tau^* \ldots (c_n!)\tau^*$) and c is equivalent to $c_1 \wedge c_2 \wedge \ldots \wedge c_n$.

We first define the probability of reaching a non–empty countable set of observable states S from a given state s on paths labeled $c?$. Let X be the set of all consistent states reachable from the state s by paths having label $c?$. Let P be the set of all paths ending in consistent states whose initial state is s and no state other than the final one is in X. Let $p = \sum\{Prob(Q) \mid Q \in \equiv_s, P \cap Q \neq \emptyset\}$.

Let P' be the set of all paths whose initial state is s, the final state is in S, no state other than the final one is in S, and whose label is $c?$. Then the probability of reaching S from s on $c?$ is defined as 0 if $p = 0$; if $p \neq 0$, it is defined as $(1/p) \times (\sum\{Prob(Q) \mid Q \in \equiv_s, P' \cap Q \neq \emptyset\})$.

A similar definition holds for the probability of reaching a non–empty countable set of consistent states S from a given state s on paths labeled $c!$.

We view CLTSs modulo the equivalence induced by the following definition of probabilistic weak bisimulation (modeled on the definition for strong bisimulation in [43]). This definition relies on **tau-finiteness** to ensure that a state can reach only countably many states on $\tau^\star$.

Definition 4.8 *Given two CLTSs C_1 and C_2, an equivalence relation, R, on the disjoint union of their observable states is called a* ***bisimulation*** *if (1) their start states are related by R, and (2) whenever two states s_1 and s_2 are R-related, then for any finite constraint c and any R-equivalence class of states S the probability of reaching S from s_1 on $c?/c!$ is the same as the probability of reaching S from s_2 on $c?/c!$.*

Example 4.9 *Consider the CLTS for c* $+_{0.5}$ *c (based on example 4.6, but with c taking the role of d too). Consider also the CLTS for c from example 4.4. These two CLTSs are bisimilar. The witnessing binary relation relates all consistent states with the same label.*

Example 4.10 *Consider the CLTS for c* $+_{0.5}$ `false` *(based on example 4.6, but with* `false` *taking the role of d) and the CLTS for c (example 4.4). These two CLTSs are bisimilar. The states of* `false` *do not need to be part of the relation because they are all inconsistent. Thus, the witnessing binary relation relates (1) all consistent states with the same label coming from the CLTS for c with the same label, and (2) the start state of the CLTS for c* $+_{0.5}$ `false` *and c.*

4.4 An algebra of CLTSs

Given the definition of a CLTS, we now describe a Probabilistic cc algebra of CLTS, *i.e.* each Probabilistic cc combinator is interpreted as a function on CLTSs. The construction for `true` (example 4.3) and c (example 4.4) have been already described. We describe the basic intuitions in the inductive cases. In each case, the transition system needs to be closed under the closure conditions to get a CLTS.

if c **then** A. Consider the CLTS for `true`(example 4.3), and restrict it to the states with labels d such that $d \not\supseteq c$, by removing all states with labels d such that $d \supseteq c$. Add transitions labeled $e?$ from a state with label e' (in the altered copy of `true`) to a state of A reached by a transition $d?$ from the start state of A, if $d \supseteq c$ and d is equivalent to $e' \wedge e$. The new start state is the start state of the copy of `true` and the CLTS is restricted to its reachable states. Two paths in the resulting CLTS are equivalent if their restriction to the states of A are equivalent; the probability numbers of an equivalence class of paths are inherited from the CLTS for A (the equivalence class(es) of paths contained completely in the copy of `true` have probability number 1).

Parallel composition. Parallel composition proceeds by a product construction that mixes aspects of asynchronous and synchronous products. We first form the product of the sets of states. Define transitions as follows:

$$\frac{s_1 \xrightarrow{c!} t_1 \quad s_2 \xrightarrow{c?} t_2}{(s_1, s_2) \xrightarrow{c!} (t_1, t_2)} \qquad \frac{s_1 \xrightarrow{c?} t_1 \quad s_2 \xrightarrow{c!} t_2}{(s_1, s_2) \xrightarrow{c!} (t_1, t_2)}$$

$$\frac{s_1 \xrightarrow{c?} t_1 \quad s_2 \xrightarrow{c?} t_2}{(s_1, s_2) \xrightarrow{c?} (t_1, t_2)}$$

$$\frac{s_1 \xrightarrow{\tau} t_1}{(s_1, s_2) \xrightarrow{\tau} (t_1, s_2)} \qquad \frac{s_2 \xrightarrow{\tau} t_2}{(s_1, s_2) \xrightarrow{\tau} (s_1, t_2)}$$

We draw the readers attention to two technical points. Firstly, our handling of τ transitions means that the probabilities at a state can add up to more than 1. For example, let each parallel component have a state with two tau transitions of probability 0.5; the resulting product state has four τ transitions each with associated number 0.5. This peculiarity does not affect the results of this paper; indeed, it can be fixed by a slightly more involved syntactic construction that we will not describe in this paper. Secondly, unlike in traditional process algebras, the $c!$ transition and the $c?$ transition do not "cancel" each other. Rather the broadcast style captured by the CLTS construction captures the cc intuition that if one process tells c, then this constraint is in the global store and in effect is broadcast to all other processes. The *masking* condition is necessary to ensure that $c!$ transitions in parallel composition are deterministic, while *saturation* ensures that all necessary synchronizations happen. The set of equivalence classes of paths in the product CLTS is the product of the sets of equivalence classes of the two component CLTSs. Two paths in the product are equivalent if their projections on the two component systems are equivalent; their probability is the product of the two respective probabilities due to Property 2.5.

New variables. The CLTS for **new** X **in** A is constructed in the following stages.

From the CLTS for A remove all transitions labeled $c?$ where $\exists_X c \neq c$, and delete any states not connected to the start state. This step prevents the process from receiving any X-information from the environment.

Replace all (output) labels $c!$ with $\exists_X.c!$ to prevent output of X-information. This however may violate the determinacy condition on CLTSs, so we collapse certain states. We define an equivalence relation S on the remaining states of A as follows: S is defined inductively as the smallest equivalence relation satisfying (a) $sSs', s \xrightarrow{(\exists x.c)?} s_1, s' \xrightarrow{(\exists x.c)?} s_1'$, then $s_1 S s_1'$ and (b)$sSs', s \xrightarrow{(\exists x.c)!} s_1, s' \xrightarrow{(\exists x.c)!} s_1'$, then $s_1 S s_1'$ and (c) $sSs', s \xrightarrow{\tau} s_1, s' \xrightarrow{\tau} s_1'$, the paths from the start state s_0 of A to s_1, s_1' are in the same equivalence class of paths in $\equiv_{s_0}$, then $s_1 S s_1'$. We quotient the remaining states of A under the relation S.

The start state of the resulting CLTS is the (equivalence class of) the start state of A; and the equivalence relation on paths is inherited from A.

choose X **from** $\{0, 1\}$ **in** P. This corresponds to **new** X **in** $(P, X = 0) +_{0.5} (P, X = 1)$, so the CLTS for it will be very similar to that in example 4.6. Let P_0 be the CLTS corresponding to $P, X = 0$, and P_1 be the CLTS corresponding to $P, X = 1$. Construct a CLTS A as follows. A contains the disjoint union of the CLTSs of `true`, P_0 and P_1. To each state coming from `true` and with label c, add two τ transitions each with probability 0.5; one going to the target of the $c?$ transition from the start state of P_0 and the other going to the the target of the $c?$ transition from the start state of P_1. The start state of `true` becomes the start state. The set of equivalence classes of paths from a state s coming from the copy of true are determined by the choice made on X (see example 4.6) and are as follows:

1. Choice not resolved: this equivalence class is the set of paths starting that only visit states from the copy of `true`and has probability number 1.
2. Choice $X = 0$: this class is the set of paths that visit some state from P_0 and has probability number 0.5.
3. Choice $X = 1$: this class is the set of paths that visit some state from P_1 and has probability number 0.5.

The required CLTS is given by **new** X **in** A, *i.e.* hide the variable X in A.

The algebra of CLTSs serves as a suitable target for the semantics of finite Probabilistic cc programs because of the following theorem.

Theorem 4.11 *Weak bisimulation is a congruence on CLTS.*

4.5 Recursion

In analogy with the treatment of recursion in the operational semantics, we treat an infinite Probabilistic cc process as a countable set of finite CLTSs. Intuitively, the set associated with a program P can be viewed as the set of CLTSs corresponding to the elements of App(P). We first define an ordering $\leq$ on CLTS. Let L, L' be CLTSs.

Definition 4.12 $L \leq L'$ *if the state/transition set of* L *is a subset of the state/transition set of* L' *and*

- *The start states of* L *and* L' *are the same.*
- *If* t *is a* τ *transition in* L' *whose target is in* L*, then the source of* t *is in* L *and* t *is a transition in* L*.*
- *If* $p_1 \equiv_s p_2$ *in* L' *for paths of* L *starting from* s*, then* $p_1 \equiv_s p_2$ *in* L *with same probability number.*

Our idea is to model (potentially infinite) Probabilistic cc programs by countable directed (wrt $\leq$) sets of CLTSs.

Example 4.13 *Recall the program of Example 2.9.*

$$\begin{aligned} U(l,u,z) ::\ & z \in [l,u], \\ & \textbf{choose } X \textbf{ from } \{0,1\} \textbf{ in } [\\ & \quad \textbf{if } X = 0 \textbf{ then } U(l,(u+l)/2,z), \\ & \quad \textbf{if } X = 1 \textbf{ then } U((u+l)/2,u,z)] \end{aligned}$$

Recall that this program can be visualized as a full binary branching tree, where the branches correspond to the two possible values for the random variable of each recursive call. The directed set corresponding to this program is induced by the finite prefixes of this tree satisfying the condition that every non-leaf node has exactly 2 children — such a prefix corresponds to some element of the set of syntactic finite approximants of the operational semantics (section 3). Indeed, every such prefix forms the the skeleton for the associated CLTS where each node is equipped with a (self-loop) transition with label $c!$ where c is the finest associated interval, *e.g.* the left (resp. right) child of the root node has a self loop of the form $(z \in [0, 0.5])!$ (resp. $(z \in [0.5.1])!$).

Example 4.14 *Consider the following modified variant of the program of Example 2.9.*

$$\begin{aligned} U(l,u,z) ::\ & z \in [l,u], \\ & \textbf{choose } X \textbf{ from } \{0,1,2,3\} \textbf{ in } [\\ & \quad \textbf{if } X = 0 \textbf{ then } U(l,(u+3l)/4,z), \\ & \quad \textbf{if } X = 1 \textbf{ then } U((u+3l)/4,(u+l)/2,z), \\ & \quad \textbf{if } X = 2 \textbf{ then } U((u+l)/2,(3u+l)/4,z), \\ & \quad \textbf{if } X = 3 \textbf{ then } U((3u+l)/4,u,z)] \end{aligned}$$

The directed set has similar intuitions to the one from example 4.13. This program can be visualized as a full quarternary branching tree, where the branches correspond to the four possible values for the random variable of each recursive call. Consider the the finite prefixes of this tree satisfying the condition that all non-leaf nodes have exactly 4 children. As in example 4.13, each node is equipped with a (self-loop) transition with label $c!$ where c is the finest associated interval, eg. the children of the root node have self loops of the form $(z \in [0, 0.25])!$, $(z \in [0.25.0.5])!$, $(z \in [0.5, 0.75])!$, $(z \in [0.75, 1])!$. The directed set corresponding to this program consists of the CLTSs built out of the skeleton transition system encoded in these trees.

IO relation. The IO relation for directed sets of CLTSs is defined as a limit of the IO relation of the elements of the set following the ideas of Section 3 — the directed set of CLTSs takes the place of the directed set of syntactic approximations in the definitions of Section 3.

Weak bisimulation. First some notation. Let $D = \{A_i\}$ be a directed set of CLTSs. Define the LTS $\mathcal{D}$ whose states and transitions are the union of the set of states of the CLTSs A_i[5]. Let its state set be X.

Let $S \subseteq X$. We first define S_{A_i}, the projection of S on A_i. Let s be a state of A_i. Then $s \in S_{A_i}$, if in $\mathcal{D}$ s reaches some state of S by a path of the form $\tau^\star$, and no other state of A_i is on this path.

We now define the probability of reaching a non–empty countable set of states $S \subseteq X$ from a given state s on paths labeled $c?$ (resp. $c!$). Let s be a state in A_k and $k \leq s$. Let p_{A_i}, the (unnormalized) probability of reaching S_{A_i} from s on a path labeled $c?$ (resp. $c!$) [6] We define the probability of reaching S from s on a path labeled $c?$ (resp. $c!$) as the limit of the net $\{p_{A_i}\}$. In the absence of normalization, this limit always exists.

Definition 4.15 *Let D and E be two directed sets of CLTSs, and let $\mathcal{D}$ and $\mathcal{E}$ be their union LTSs. A partial equivalence relation, R, on the disjoint union of the states in $\mathcal{D}$ and $\mathcal{E}$ is called a **bisimulation** if*

- *Let s be a state such that s is not related to s by R. Then, every $\tau^\star$ path from s can be extended to a $\tau^\star$ path ending in a state t such that t R t; every τ^∞ path from s includes a state t such that t R t, where these paths are in the respective LTSs.*
- *Whenever two states s_1 and s_2 are R-related, then for any finite constraint c and any R-equivalence class of states S the probability of reaching S from s_1 on $c?/c!$ is the same as the probability of reaching S from s_2 on $c?/c!$. Two states are **bisimilar** if there is a bisimulation relating two states.*

Two CLTSs are bisimilar if there is a binary relation between their state sets, satisfying the above conditions such that the initial states are bisimilar.

In the above definition, the partiality of the equivalence relation captures the idea that all internal configurations need not be matched. This idea was captured by the restriction to observable states in the finite case. This issue is illustrated by the following example.

Example 4.16 *There is a bisimulation relating the trees of example 4.13 and example 4.14. The witnessing partial equivalence relation relates the nodes of example 4.14 to the corresponding nodes of example 4.13. The other nodes of example 4.13 (*e.g.* the children of the root node, and every alternating level of nodes from thereon) are not included in the equivalence relation.*

The following non–example illustrates the issues further.

Example 4.17 *No two distinct nodes of the tree described in Examples 4.13 can be in the same equivalence class of a bisimulation relation, since their possible outputs distinguish them. Similarly, no two distinct nodes of Example 4.14 can be in the same equivalence class of a bisimulation relation.*

The absence of normalization in the computation of probability numbers of our programs shows up in the following example. (recall that for finite recursion free programs A, the semantics of Section 4.3 validates the equational law $A +_{0.5} \texttt{false} = A$.)

Example 4.18 *The programs $A +_{0.5} \texttt{false}$ and A are not bisimilar in general.*

[5]Note that $\mathcal{D}$ may not be CLTS as it may not satisfy τ-finiteness.

[6]p_{A_i} is defined following the definitions of section 4.3. Let P' be the set of all paths in A_i whose initial state is s, the final state is in S_{A_i}, no state other than the final one is in S, and whose label is $c?$ (resp. $c!$). Then, $p_{A_i} = (\sum\{Prob(Q) \mid Q \in \ \equiv_s, P' \cap Q \neq \emptyset\})$.

Algebra of directed sets of CLTSs. The following lemma lifts the Probabilistic cc algebra to sets. It allows us to lift the Probabilistic cc combinators to sets of CLTSs by just extending them "pointwise" — *e.g.* the parallel composition of two sets is the set of CLTSs got by performing the defined composition on all possible pairs from the two sets.

Lemma 4.19 *All operations in the algebra of CLTS are monotone with respect to $\leq$.*

The earlier theorem that weak bisimulation is a congruence gets lifted to directed sets of CLTSs:

Lemma 4.20 *Weak bisimulation is a congruence on directed sets of CLTS.*

4.6 Correspondence and conservativity results

Theorem 4.21 (Adequacy) *(Directed sets of) CLTS modulo weak bisimulation is sound for reasoning about* Probabilistic cc *with respect to observations of IO relations.*

The key step in this theorem is to show computational adequacy — the operational and CLTS IO relations coincide. Our proof exploits the set construction in the CLTS semantics for recursive programs to reduce the proof to the case of finite recursion free programs. For this case, the proof is carried out by using a standard cc style argument to reduce finite recursion free programs to the following normal form — random variable declarations at the outside enclosing a body with all local variable declarations outside a body built out of tells, composition and asks.

The directed sets CLTS semantics is not complete, *i.e.* not fully abstract because it does not handle normalization of probability numbers.

The theory is consistent with limit observing semantics of determinate cc [54, Pg. 344] — the output `false` of [54] corresponds to absence of output in our treatment of determinate cc. Example 4.5 motivates:.

Theorem 4.22 *CLTS modulo weak bisimulation is a fully abstract semantics for determinate* cc *with respect to observations of IO relation.*

Corollary 4.23 (Conservativity) Probabilistic cc *is conservative over* cc.

5 Related work

Our integrated treatment of probability and the underlying concurrent programming paradigm is inspired by [9, 2], *e.g.* conditions 2.2 and 2.5 are directly borrowed from these papers. The treatment of recursion and associated semantics are not explored in these papers. Our work falls fundamentally into the realm of study initiated in these two papers, with our contribution being the integration of programming language and concurrency theory methods.

The role of probability has been extensively studied in the context of several models of concurrency. Typically, these studies have involved a marriage of a concurrent computation model with a model of probability.

(1) Probabilistic process algebras add a notion of randomness to the underlying process algebra model. This theory is quite comprehensive and these extensive studies have been carried out in the traditional framework of (different) semantic theories of (different) process algebras (to name but a few, see [32, 41, 43, 34, 5, 61, 18]) *e.g.* bisimulation, theories of (probabilistic) testing, relationship with (probabilistic) modal logics etc. Recently, these theories have

been shown to extend nicely to continuous distributions [15, 22]. We have imported powerful machinery to analyze labeled transition systems from this area of probabilistic process algebra into our work.
(2) The work of Jane Hillston [37] develops a process algebra, PEPA, for compositional performance modeling. The probabilities enter through the fact that each action has a duration chosen according to an exponential distribution. A natural question for us is to encode her process algebra in Probabilistic cc. This would lead us into the integration of explicit continuous time into our model.
(3) The verification community has been very active and there has been significant activity in developing model checking tools for probabilistic systems, for example [13, 6, 20, 38]. Our work is not directedly related but should be seen as a complementary tool.
(4) Probabilistic Petri nets [45, 62] add Markov chains to the underlying Petri net model. This area has a well developed suite of algorithms for performance evaluation. Example 2.8 shows how to represent such nets in Probabilistic cc.
(5) Probabilistic studies have also been carried out in the context of IO Automata [58, 63]. The reader will have noticed the influence of IO-automata on the definition of CLTS.

Our work differs in the following aspects. Formally, our work is based on the cc model. More importantly perhaps, our work focuses on the execution aspect of stochastic models, in contrast to the specification focus of the above. Thus, our model remains determinately probabilistic and we integrate probability more deeply into the language, *e.g.* the cc paradigm is exploited to build and specify joint probability distributions of several variables. This forces us to explore the semantic issues associated with the interaction of constraints, recursion and random variables; issues not treated in the above theories. On the other hand, our work does not currently incorporate the sophisticated reasoning methodologies of the above formalisms. Our semantic study based on classical techniques, labeled transition systems and bisimulation, makes us hopeful that this technology can be adapted to Probabilistic cc.

The development of probabilistic frameworks in knowledge representation has been extensive [51]. Our earlier examples motivate how to express the joint probability distributions of Bayesian networks within Probabilistic cc making Probabilistic cc a simple notation for describing Bayesian networks. However, Probabilistic cc does not allow the direct manipulation of conditional probability assertions as in the logics of [49, 23]. The work in this genre that is most closely related to our work is the work on (lazy first order) stochastic functional programming languages [42]. The key technical contribution of that paper is an algorithm that computes the output distribution exactly when all possible execution paths terminate. Our paper differs in the choice of the underlying concurrent computing idiom — recall our earlier arguments for the importance of concurrency. [42] also does not handle probability distributions when the underlying computation does not terminate. However, we hope to be able to adapt the very general (program analysis style) ideas in the algorithm of that paper to develop reasoning methods for Probabilistic cc.

References

[1] H. Abelson, T. F. Knight, and G. J. Sussman. Amorphous computing manifesto. http://www-swiss.ai.mit.edu/ switz/amorphous/white-paper/amorph-new/amorph-new.html, 1996.

[2] A. Aghasaryan, R. Boubour, E. Fabre, C. Jard, and A. Benveniste. A petri net approach to fault detection and diagnosis in distributed systems. Technical Report PI-1117, IRISA — Institut de recherche en informatique et systemes aleatoires – Rennes, 1997.

[3] Rajeev Alur, Lalita Jategaonkar Jagadeesan, Joseph J. Kott, and James E. Von Olnhausen. Model-checking of real-time systems: A telecommunications application. In *Proceedings of the 19th International conference on Software Engineering*, pages 514–524, 1997.

[4] Robert B. Ash. *Real Analysis and Probability*. Academic Press, 1972.

[5] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. Axiomatizing probabilistic processes: Acp with generative probabilities. *Information and Computation*, 121(2):234–255, 1995.

[6] Christel Baier, Ed Clark, Vasiliki Hartonas-Garmhausen, Marta Kwiatkowska, and Mark Ryan. Symbolic model checking for probabilistic processes. In *Proceedings of the 24th International Colloquium On Automata Languages And Programming*, number 1256 in Lecture Notes In Computer Science, pages 430–440, 1997.

[7] A. Benveniste and G. Berry, editors. *Another Look at Real-time Systems*, volume 79:9, September 1991.

[8] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE* [7], pages 1270–1282.

[9] Albert Benveniste, Bernard C. Levy, Eric Fabre, and Paul Le Guernic. A calculus of stochastic systems for the specification, simulation, and hidden state estimation of mixed stochastic/nonstochastic systems. *Theoretical Computer Science*, 152(2):171–217, Dec 1995.

[10] A. Berlin, H. Abelson, N. Cohen, L. Fogel, C-H. Ho, M. Horowitz, J. How, T. F. Knight, R. Newton, and K. Pistel. Distributed information systems for mems. Technical report, Xerox Palo Alto Research Center, 1995.

[11] G. Berry. Preemption in concurrent systems. In R. K. Shyamasundar, editor, *Proc. of FSTTCS*, pages 72–93. Springer-Verlag, 1993. LNCS 761.

[12] G. Berry and G. Gonthier. The ESTEREL programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87 – 152, November 1992.

[13] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Proceedings of the 15th Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1026 in Lecture Notes In Computer Science, pages 499–513, 1995.

[14] Patrick Billingsley. *Probability and Measure*. Wiley-Interscience, 1995.

[15] Richard Blute, Josée Desharnais, Abbas Edalat, and Prakash Panangaden. Bisimulation for labelled markov processes. In *Proceedings of the Twelfth IEEE Symposium On Logic In Computer Science, Warsaw, Poland.*, 1997.

[16] J. Dean Brock and W. B. Ackerman. Scenarios: A model of nondeterminate computation. In J. Diaz and I. Ramos, editors, *International Colloquium on Formalization of Programming Concepts*, pages 252–259. Springer-Verlag, 1981. Lect. Notes in Comp. Sci. 107.

[17] Bjorn Carlson, Vineet Gupta, and Tadd Hogg. Controlling agents in smart matter with global constraints. In *Proceedings of the AAAI Workshop on Constraints and Agents*, July 1997.

[18] R. Cleaveland, S. A. Smolka, and A. Zwarico. Testing preorders for probabilistic processes. *Lecture Notes in Computer Science*, 623, 1992.

[19] D. Coore, R. Nagpal, and R. Weiss. Paradigms for structure in an amorphous computer. Technical Report AI Memo 1614, MIT, 1997.

[20] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[21] Johan de Kleer and Brian C. Williams. Diagnosis with behavioral modes. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1324–1330, August 1989.

[22] Josee Desharnais, Abbas Edalat, and Prakash Panangaden. A logical characterization of bisimulation for labeled markov processes. In *Proceedings of the 13th IEEE Symposium On Logic In Computer Science, Indianapolis*. IEEE Press, June 1998.

[23] R. Fagin, J.Y.Halpern, and N. Megiddo. A logic for reasoning about probabilities. *Information and Computation*, 87:78–128, 1990.

[24] Erann Gat. Towards principled experimental study of autonomous mobile robots. *Autonomous Robots*, 2:179–189, 1995.

[25] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with SIGNAL. In *Proceedings of the IEEE* [7], pages 1321–1336.

[26] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Probabilistic concurrent constraint programming. In Antoni Mazurkiewicz and Jozef Winkowski, editors, *CONCUR'97: Concurrency Theory*, Lecture notes in computer science, vol 1243. Springer Verlag, August 1997.

[27] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1-2):3–50, 1998.

[28] Vineet Gupta, Radha Jagadeesan, Vijay Saraswat, and Daniel Bobrow. Programming in hybrid constraint languages. In Panos Antsaklis, Wolf Kohn, Anil Nerode, and Sankar Sastry, editors, *Hybrid Systems II*, volume 999 of *Lecture notes in computer science*. Springer Verlag, November 1995.

[29] Vineet Gupta, Vijay Saraswat, and Peter Struss. A model of a photocopier paper path. In *Proceedings of the 2nd IJCAI Workshop on Engineering Problems for Qualitative Reasoning*, August 1995.

[30] N. Halbwachs. *Synchronous programming of reactive systems*. The Kluwer international series in Engineering and Computer Science. Kluwer Academic publishers, 1993.

[31] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language LUSTRE. In *Proceedings of the IEEE* [7], pages 1305–1320.

[32] H. Hansson and B. Jonsson. A calculus for communicating systems with time and probabilities. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 278–287. IEEE Computer Society Press, 1990.

[33] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[34] S. Hart and M. Sharir. Probabilistic propositional temporal logics. *Information and Control*, 70:97–155, 1986.

[35] M. Hayden and K. Birman. Probabilistic broadcast. Technical Report TR96-1606, Cornell University, 1996.

[36] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Constraint processing in cc(fd). Technical report, Computer Science Department, Brown University, 1992.

[37] Jane Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, University of Edinburgh, 1994. To be published as a Distinguished Dissertation by Cambridge University Press.

[38] Michael Huth and Marta Kwiatkowska. Quantitative analysis and model checking. In *proceedings of the 12 IEEE Symposium On Logic In Computer Science*, pages 111–122. IEEE Press, 1997.

[39] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*, pages 167–186. MIT Press, 1991.

[40] C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 186–195, Asilomar Conference Center, Pacific Grove, California, 1989.

[41] Bengt Jonsson and Wang Yi. Compositional testing preorders for probabilistic processes. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 431–441, San Diego, California, 1995.

[42] D. Koller, D. McAllester, and A. Pfeffer. Effective bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, pages 740–747, 1997.

[43] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, September 1991.

[44] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[45] M. Ajmone Marsan. Stochastic petri nets: an elementary introduction. In *Advances in Petri Nets 1989*, pages 1–29. Springer, June 1989.

[46] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[47] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[48] R. Nagpal and D. Coore. An algorithm for group formation and finding a maximal independent set in an amorphous computer. Technical Report LCS-TR-737, MIT, 1998.

[49] N. J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28:71–87, 1986.

[50] P. Panangaden. The expressive power of indeterminate primitives in asynchronous computation. In P. S. Thiagarajan, editor, *Proceedings of the Fifteenth Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes In Computer Science, 1995. Invited Lecture.

[51] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan-Kaufmann Publishers, 1988.

[52] J. R. Russell. Full abstraction for nondeterministic dataflow networks. In *Proceedings of the 30th Annual Symposium of Foundations of Computer Science*, pages 170–177, 1989.

[53] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6):475–520, November/December 1996. Extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[54] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, pages 333–352, January 1991.

[55] Vijay A. Saraswat. The Category of Constraint Systems is Cartesian-closed. In *Proc. 7th IEEE Symp. on Logic in Computer Science, Santa Cruz*, 1992.

[56] Vijay A. Saraswat. *Concurrent constraint programming*. Doctoral Dissertation Award and Logic Programming Series. MIT Press, 1993.

[57] Dana Scott. Lattice theory, data types and semantics. In Randall Rustin, editor, *Formal Semantics and Programming Languages*, pages 65–106. Prentice Hall, 1972.

[58] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science, 1995. Also appears as technical report MIT/LCS/TR-676.

[59] G. E. Shilov and B. L. Gurevich. *Integral, Measure and Derivative: A Unified Approach*. Prentice-Hall Inc., 1966.

[60] Gert Smolka, M. Henz, and J. Werz. *Constraint Programming: The Newport Papers*, chapter Object-oriented programming in Oz. MIT Press, 1994.

[61] R. van Glabbeek, S.A. Smolka, and B.U. Steffen. Reactive, generative, and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 1995.

[62] N. Viswanadham and Y. Narahari. *Performance Modeling of Automated Manufacturing Systems*. Prentice-Hall Inc., 1992.

[63] S.-H. Wu, S.A. Smolka, and E. Stark. Compositionality and full abstraction for probabilistic i/o automata. *Theoretical Computer Science*, 1996. Preliminary version in CONCUR94.