Probabilistic Programs and their Verification: A Brief Survey

Abhinav Aggarwal
University of New Mexico, Albuquerque, NM USA

Abstract

This report presents a brief survey on probabilistic programs to help understand formal models of probabilistic program analysis. As fas as verification protocols are concerned, mainly Hartog's paper on extending Hoare triples for the verification is studied, the examples of which are provided in the paper. More techniques, like the ones mentioned in Hurd's thesis and several others are a part of my reading list for this semester. I also present an interesting analogy between a popular game called "Flappy bird" and the general execution of probabilistic programs. I raise two questions at the end which may shed some insight into verification of the same.

1 Formal models

The various formal models that are available have been nicely summarized in [7] and [9]. The three main ones are referred to as the standard model, relational model and the lifted model. However, before moving on to these models, some definitions should be kept in mind. As mentioned in [11], an *inductive invariant* of a loop is a property on the state that is preserved by executing the body of the loop from states in which the loop-guard is satisfied. In this regard, we use Dijsktra's and Scholten's concept of *weakest liberal precondition*, $wlp.S.Q$, which for each predicate Q on the state space of a qualitative program S, equals the predicate describing the largest set of states from which S is guaranteed to either not terminate or do so in a state satisfying Q. Katoen et al. use predicate pairs $[P, Q]$ for specifying a program S by saying that S satisfies the specification $[P, Q]$ if $P \implies wlp.S.Q$, which is the same as saying $\{P\}S\{Q\}$ in Hoare's axiomatic system.

A refinement ordering $\preceq$ is defined between programs S, T by saying that $S \preceq T$ iff T is guaranteed to satisfy all the specifications $[P, Q]$ that are satisfied by S, i.e.

$$S \preceq T \equiv (\forall Q \quad wlp.S.Q \implies wlp.T.Q)$$

This allows specifications to be treated as programs themselves by saying that $wlp.[P, Q]$ is the least program S (wrt $\preceq$) such that S satisfies $\{P\}S\{Q\}$. A more formal description of these specifications is given in [2], where in Chapter 27, a *specification* is defined as a statement that indicates in a simple way *what* is to be computed without saying *how* it is computed. In our context, this translates to knowing what the preconditions and postconditions of a program are, without knowing how the program actually takes us from the former to the latter. An appropriately designed precondition and postcondition set for a program will be able to guarantee it's termination and correctness without knowing the actual computation performed. Back et al. also provide definitions for three types of specifications : *general monotonic specification, conjunctive specification* and *disjunctive specification*. I aim to study these in detail as a part of this course to get a better understanding of the concept.

Coming back to Katoen's formulation in [11], the semantics of qualitative programs are studied using inductive invariant maps, which essentially provide invariants of all the loops in the given program, subject to the constraints that the Hoare's triples are valid for each program point. However generating valid inductive

invariant maps for a given qualitative loop involves finding solutions to a set of second order constraints of the form $\exists$ an invariant I for a given loop L. Aumotating such a process is the biggest obstacle with such a technique, where generation of invariants signs in. For probabilistic programs, a similar requirement occurs but in a different framework called the *pGCL* or the *probabilistic guarded command language*. Although Katoen gives a nice semantics of this language, a better understanding with detail is provided in [9]. Citing Dijktra's work, He et al. define the probabilistic choice operator $\oplus_p$, which indicates that if we have $P \oplus_p Q$, then we execute program P with probability $(1-p)$ and program Q otherwise. This way, every sequential program begins in an initial state and through the probabilistic choices made during the program, either doesn't terminate or does so in a set of possible states. This way, a distribution over the set of final states is generated which tells us the probability for each final state that the program will terminate in that state. This forms the basis of the standard model of probabilistic programs as descirbed by [9].

Standard Model : This model assumes program specification as a pair of precondition and postcondition by saying that P is a program with specification $(pre(s), post(s, s'))$ if given the initial state s and a precondition $pre(s)$ satisfied by that initial state, the program P either does not terminate or does so in the final state s' that satisfies the postcondition $post(s, s')$. With this specification, the refinement ordering over the programs becomes

$$(pre1, post1) \preceq (pre2, post2) \equiv (pre1 \wedge \neg(post1; \neg pre2), post1; post2)$$

Hence, a standard program *refines* another if it terminates more often and less non-deterministically than the other. Now when the program has indeed terminated, we have a probability distribution over the set of final states. This distribution, say $f : \mathcal{P}(S_\perp) \to [0, 1]$, basically maps every subset of the set of states $S_\perp$ to the interval $[0, 1]$ such that

1. $f(\phi) = 0$ and $f(S_\perp) = 1$
2. $f(\bigcup_{i \in I} x_i) = \sum_{i \in I} f(x_i)$ if all x_i are disjoint.

For any state $s \in S_\perp$, a point distribution η_s is defined as $\eta_s(X) = 1$ if $s \in X$ and 0 otherwise. This way, a refinement ordering between the distributions is defined by saying that one probability distribution refines another if it assignes higher probabilities to all sets of proper states. Subsequently, a non-deterministic program P refines Q if from every starting state, P results only in distributions that are refinements of distributions produced by Q. However, keep in mind that the standard model does not contain any notion of probabilistic choice, it only allows pure non-deterministic choice.

Relational model : The next model as described by [9] is the one in which we focus on the fact that the result of the execution of a program P starting at an initial state s can be described as a set of probability distributions over the final states (including $\perp$).

$$P : S_\perp \to 2^D$$

Here, D represents the set of all probability distributions over $S_\perp$. Note that every program P that satisfies this specification can be considered valid if it behaves chaotically whenever the input is improper i.e. $P(\perp) = D$. The paper also defines two important notions of *up-closure* and *convex-closure* of probability distributions, which are heavily used for analysis later. However, with this definition of programs, we define the set of all programs as

$$PROGS = \{P : S_\perp \to 2^D \mid P(\perp) = D\}$$

and say that a program $P \in PROGS$ is *worse than* (or less determined than) a program $Q \in PROGS$ if for every initial state s, program P gives a bigger set of possible distributions over the final states, i.e.

$$P \preceq Q \equiv (\forall s \in S_\perp \quad Q(s) \subseteq P(s))$$

This makes $(PROGS, \succeq)$ a complete lattice. Two important theorems in the semantics of this relational model are given below. The first one is for the probabilistic choice operator:

$$(P \oplus_p Q)(s) \equiv \{(1-p)f + pg \mid f \in P(s) \wedge g \in Q(s)\}$$

Similarly, for the sequential composition, execution of P gives the set $P(s)$ of distributions over intermediate states (invisible to the outside). Every distribution over the final states of $(P;Q)(s)$ is the result of picking for every intermediate state t a distribution in $Q(t)$ and taking the weighted sum of these distributions, the weights being the probabilities assigned to the intermediate states by some distribution in $P(s)$, i.e.

$$(P;Q)(s) \equiv \{\sum_{t \in S_\perp} f(t)h_t \mid f \in P(s) \wedge (\forall t \in S_\perp, h_t \in Q(t))\}$$

A number of laws are obeyed by the constructs in the relational model, details of which can be found in [9]. However, an important concept demonstrated in this paper is the relation between relational model and standard model where the conversion from one to another is given by a Galois connection $(\downarrow, \uparrow)$. Where on one hand, the standard model (SM) tells us which final states are possible, the relational model (RM) tells us what final states occur with what probabilities. The definitions of these two functions is given as follows:

$$\uparrow P = (pre(s), post(s, s') \text{ where,}$$

$$pre(s) = (\forall f \in P(s) \quad f(\perp) = 0)$$
$$post(s, s') = (\exists f \in P(s) \quad f(s' > 0)$$

and then for conversion from SM to RM, we have,

$$\downarrow (pre, post).\ \perp = D$$
$$\downarrow (pre, post).s = cc.\{\eta_{s'} \mid post(s, s')\} \triangleleft pre(s) \triangleright D$$

where, cc stands for convex closure.

Thus, when we go from SM to RM through the function $\downarrow$, and then come back to SM through $\uparrow$, we get back to where we started because all the information of non-detemrinistic choice is preserved by probabilistic choice. However, when we go from RM to SM through $\uparrow$ and then come back to RM through $\downarrow$, we get a less refined program back due to loss of information at the non-deterministic part, i.e. given that $(\uparrow, \downarrow)$ forms a Galois connection, we have

$$\uparrow (\downarrow (pre, post)) = (pre, post)$$
$$\downarrow (\uparrow Q) \preceq Q$$

Lifted model : This model, as first described in [10], is defined for GCL (*guarded command language*). It contains the probabilistic choice operator but not the non-deterministic choice. The proper states are evaluations, which are like pseudo-distributions that do not need to sum upto one. This is achieved for probability distributions by adding an improper state $\perp$ to the set of proper states. Thus, every program is a function from $S_\perp \to D$. In this model, all such programs are deterministic, in the sense that given an initial state, there is exactly one probability distribution over the final states. Accordingly, we lift the pointwise ordering operator $\leq$ as follows:

$$P \preceq Q \equiv (\forall s \in S \quad P(s) \leq Q(s)$$

This way, if we call $P_D = [S_\perp \to D]$ as the set of all programs that are valid in the lifted model, then $(P_D, \preceq)$ forms a complete partial order (CPO). Construction of probabilistic powerdomains is now possible over this CPO. To express non-determinism in this model, we use the concepts of convex closed and up-closed sets of deterministic programs. Let P_L be defined as

$$P_L = \{P \in 2^{P_D} \mid P \text{ is convex closed and up-closed } \}$$

A CPO on this model is constructed using the Smyth ordering on convex closed and up-closed sets, which in our case simplifies to inclusion ordering, i.e.

$$P \preceq Q \equiv Q \subseteq P \quad \forall P, Q \in P_L$$

Hence, we can now say that non-determinism in the lifted model is a complile-time decision where the compiler choses P or Q before any probabilistic choice is made, as opposed to the non-determinism in the relational model, which is a real-time decision where the choice of final distribution is made during execution. Non-determinism cannot take advantage of probabilistic choice in the lifted model.

Finally, I discuss the model of probabilistic programs as discussed in Hurd's thesis [7]. Although some more study needs to be done in this area, the basics of this model are now presented. Hurd uses HOL (higher order logic) to specify and verify any program equipped with a source of random bits, which are assumed to be independent, identically distributed Bernoulli(1/2) random variables. Suppose we have a probabilistic function $\tilde{f} : \alpha \to \beta$ that takes as input an element of type α and uses a random number generator to output a result of type β. Define a specification function $B : \alpha \times \beta \to \mathbb{B}$ for $(\tilde{f})$ as a predicate on pairs of elements from α to β. Say that a particular function application $\tilde{f}(a)$ satisfies the specification if $B(a, \tilde{f}(a))$ is true. Since $\tilde{f}$ is probabilistic, the application $\tilde{f}(a)$ may meet the requirements on one instance and fail it on another. Hurd then extends this definition of a function to higher order logic (HOL) as follows:

$$f : \alpha \times \mathbb{B}^\infty \to \beta \times \mathbb{B}^\infty$$

which explicityly takes as input a sequence $s \in \mathbb{B}^\infty$ of random bits in addition to an argument $a \in \alpha$, uses some of the ramdom bits in a calculation exactly mirroring $\tilde{f}(a)$ and then passes back a sequence of unused bits with the result. We then use mathematical measure theory to define a probabilistic measure function $\mathbb{P} : \mathcal{P}(\mathbb{B}^\infty) \to [0, 1]$ from sets of bit sequences to real numbers between 0 and 1. Thus, the natual question *'for a given a, with what probability does $\tilde{f}$ satisfy the specification?'* then becomes *'for a given a, what is the probability measure of the set $\{s \mid B(a, fst(fas))\}$'*. Modelling probabilistic programs this way has several advantages, as listed by Hurd.

1. Since the formalization is in HOL, no other programming language is needed to express the programs.
2. Since the notation is the same as that used in pure functional programming languages, the monadic notation can be borrowed for this case as well. This will help to elegantly express programs and easily transfer programs to and from an execution environment.
3. Formal support of only one probability space (sequence of IID Bernoulli(1/2) bits) is required for the theory of formal verification of probabilistic programs.

Using this semantics, we can then reason about programs that do not necessarily terminate, but terminate with probability 1. This means that the set of sequences that cause a probabilistic program to terminate is an event and has probability 1. A more in-depth study of this concept will be done as a part of this course.

Having looked at various models of formalizing probabilistic programs, I will now list some verification techniques to check the correctness of probabilistic programs.

2 Verification techniques

A bunch of verification techniques are listed in Hurd's thesis [7] and [8]. A detailed study of these techniques will be done as part of this reading course.

1. The first technique is to use first-order deductive provers, which use algorithms such a resolution or model elimination to perform general first-order proof search. In practice, they are extremely useful for finishing off easy goals but quickly get swamped when confronted with deeper problems.

2. The next technique is to use conditional reqriters, which take theorems of the form

$$\forall v \quad C(v) \implies (A(v) = B(v))$$

and rewrite instances of A in the goal to corresponding B. For each instance, the condition C must be proved before the rewrite can take place (usually by a decision procedure or by a recursive call to the conditional rewriter).

3. The third technique is to use decision procedures that always succeed in either proving or efuting the goal. Hurd extensively used decision procedures for Presburger arithmetic (the class of formulas that include addition, subtraction and comparison operators but no multiplications). The main limitation of decision procedures is that most goals do not naturally fall into a decidable class and must first be reduced using other tools.

4. Hurd also defines the notion of probabilistic while-loops and establishes the condition that this loop terminates with probability 1. More details of this technique will be studied in this course.

5. More techniques using linear algebra [23], probabilistic inference [6], linear invariant generation [11] and predicate transformers [20] are discussed in detail in the respective papers. However, these will be studied in detail in this course.

6. The main technique, using Hoare-triples [3] and their extension is presented in this report, specially through examples in the next section. In a nutshell, Hartog and Vink define a language $\mathcal{L}_{pif}$ with conditional and probabilistic choice. They provide a denotational semantics for this language and discuss probabilistic predicates and Hoare logic for correctness in $\mathcal{L}_{pif}$. Further, a relationship with weakest preconditions is established and completeness theorems for Haore logic discussed. Most importantly, the probabilistic predicates used in their logic retain their truth value interpretation and determinisitc predicates are extended to arithmetical functions yielding the probability that the predicate holds. This way,the logical operators do not have to be extended.

To illustrate the use of these Hoare triples for verification of probabilistic programs, axiomatic rules in [3] are used for the examples in the next section. Keep in mind that the analysis presented differs from that in the original text and one extra assumption has been made for example 3, which has been listed with the context.

3 Examples

In this section, several examples for verification of probabilistic programs using a Hoare like logic are presented. The programs are taken from [3] and [9], but the analysis is done independently.

Exmaple 1 : Consider the program below. The aim to check the probability with which the values of x and y are the same. As we will see, the sequence in which the programs are executed can affect the final result.

$P = (x := 0) \oplus (x := 1)$
$Q = (y := 0) \oplus_{1/2} (y := 1)$

The nature of non-deterministic computation differs from probabilistic computation where the semantics of making a choice are concerned. While the former makes choice indepedent of any statistical formulations, the latter usually follows a fixed distribution or "rule" to make this choice. To understand this difference better, assume that the non-deterministic choice is made by some adversary who knows that we are trying to get the values of x and y to match. Hence, the moment this adversary gets to know anything about the value of y as set by program Q, it can make a choice to set the value of x that does not match this value

of y and thus, make sure that these two values are never equal. The probabilistic choice, however, cannot make this biased decision because it is bound to follow the distribution that y be assigned one of the two values with equal probablity. Hence, when P runs before Q, we have $Pr\{x = y\} = 1/2$, but when Q runs before P, this probability becomes zero. A more formal derivation of these values can be found in [9], where the authors use a formal semantics for probabilistic programs to compute these probabilities.

The fact that non-deterministic computation can be viewed as the action of an adversary is to analyse the behavior in the worst case. Recall that the semantics of non-deterministic computation allow us to view it as a convex hull of all possible probabilistic choices which is why assigning to probabilities to final outputs makes no sense. We can only talk of worst and the best cases.

Example 2 : This example will illustrate how to calculate the postcondition for probabilistic choice using Hoare like semantics as discussed in [3].

Precondition : $[x = 1]$
$x := x + 1 \oplus_{1/3} x := x + 2$

The forward semantics for probabilistic choice operator are very straight forward. For the first choice, the post condition is $[x - 1 = 1]$, which is equivalent to $[x = 2]$. For the second choice, we have $[x - 2 = 1]$, which is equivalent to $[x = 3]$. Thus, keeping in mind that the first choice is made with probability $2/3$, the final postcondition is given as $[Pr\{x = 2\} = 2/3 \wedge Pr\{x = 3\} = 1/3]$. We can also write this distribution using the point-notation for discrete sets, i.e. $[\eta_{s;x=2} = 2/3 \wedge \eta_{s;x=3} = 1/3]$.

Example 3 : Let us now see a detailed example from [3]. The analysis is done differently from the paper, but it conforms to the axiomatic rules as presented by the authors.

int $ss[1 \ldots N], k, t$
$t := 0$, $k := 1$
while $(k \leq N)$ **do**
  $t := (t + ss[k]) \oplus_p$ `skip` ▷ Some elements are not added
  k := k+1
end while
Postcondition : $[Pr\{t = \sum_{i=1}^{N} ss[i]\} \geq p^N]$

We compute the loop invariant for the while loop first. Taking hint from the postcondition given, the appropriate loop invariant is given as $I = [Pr\{t = \sum_{i=1}^{k-1} ss[i]\} \geq p^{k-1}]$. Check that it is true just before the body of the loop, i.e. when $t = 0$ and $k = 1$. Now, using Hoare's triples, the condition to be true just after the loop terminates is given by $I \wedge \neg G$, where $G = [k \leq N]$, the loop condition.

$$I \wedge \neg G = [Pr\{t = \sum_{i=1}^{k-1} ss[i]\} \geq p^{k-1} \wedge k > N] \implies [Pr\{t = \sum_{i=1}^{N} ss[i]\} \geq p^N]$$

Hence, the postcondition is verified. The condition true just before the loop is I, which, when traced back through the assignment statement, gives $I_{t=0,k=1}$, giving $[Pr\{0 = \sum_{i=1}^{0} ss[i]\} \geq p^0] \implies \mathit{true}$. For the body of the loop, we see that $[I \wedge P] = [Pr\{t = \sum_{i-1}^{k-1} ss[i]\} \geq p^{k-1} \wedge k \leq N]$ when acts as a precondition to

$t := (t + ss[k]) \oplus_p \texttt{skip}$ gives

$$[Pr\{t - ss[k] = \sum_{i=1}^{k-1} ss[i]\} \geq p^{k-1} \wedge (k \leq N) \oplus_p Pr\{t = \sum_{i=1}^{k-1} ss[i]\} \geq p^{k-1} \wedge (k \leq N)] \implies$$
$$[Pr\{t = \sum_{i=1}^{k} ss[i]\} \geq p^{k-1} \wedge (k \leq N) \oplus_p Pr\{t = \sum_{i=1}^{k-1} ss[i]\} \geq p^{k-1} \wedge (k \leq N)] \implies$$
$$[Pr\{t - ss[k] = \sum_{i=1}^{k-1} ss[i]\} \geq p^{k-1} \oplus_p Pr\{t = \sum_{i=1}^{k-1} ss[i]\} \geq p^{k-1}]$$

Notice that $[Pr\{t = \sum_{i=1}^{N} ss[i]\} \geq p^N]$ implies $[Pr\{t - ss[k] = \sum_{i=1}^{k-1} ss[i]\} \geq p^{k-1} \oplus_p Pr\{t = \sum_{i=1}^{k-1} ss[i]\} \geq p^{k-1}]$, and thus, by using the axiom for precondition strengethening, the loop body is verified. The axiom used here is based on the fact that $a \implies (a \oplus_p b)$ for all $p < 1$. Although [3] does not mention this axiom in his list, a proof of correctness without this assumption is not possible. However, pondering over this statement, it does sound logical to say that a probabilistic choice over a and b is logically weaker than a deterministic selection of a when there is no choice of b present. Thus, one can convince himself that indeed, a does imply $a \oplus_p b$ for all p, except when $p = 1$. When $p = 1$, their is no choice and we select b without any probabilistic selection.

Example 4 : Consider the following program:

bool *done* := *false*
$k := 1, x := k$
while $\neg$*done* **do**
  done := *true* $\oplus_p$ `skip` ▷ Note that the loop may not terminate
  $k := k + 1, x := x + 1$
end while
Postcondition : $\bigwedge_{n \geq 1}[Pr\{x = n\} = p^{n-1}(1 - p)]$

The loop invariant can be designed using an approach similar to what we used for the previous example. We see that x takes values according to a geometric distribution, and hence we can use $I = \bigwedge_{1 \leq i \leq k}[Pr\{x = i\} = p^{i-1}(1 - p)]$ as our loop invariant. However, using the postcondition as given above makes the analysis using Hoare triples non-trivial.

Having seen the examples above, the final section presents an interesting analogy between a popular game called *Flappy bird* and the general execution of probabilistic programs. The comparison can help provide some insight into how probabilistic programs work in general and possibly, aid in designing an efficient verification scheme for the same.

4 Flappy bird

Flappy bird is a popular mobile game which is as simple as tapping a finger on the screen in order to maneuver a small bird through a path with obstacles. The path is horizontal and the bird is constantly moving from left to right. Every tap on the screen inverts the direction of bird's motion along the vertical. The obstacles along the path are like walls standing on the ground as well as imaginary walls hanging down the sky, Thus, if we view the whole screen as the real interval $(0, 1)$, with 0 being the ground and 1 being the sky, every obstacle requires the bird to move through only some sub-interval (a, b), where $0 \leq a < b \leq 1$. The interval (a, b) is different for every obstacle and there is a fixed space between two obstacles to allow for the bird to maneuver. The aim is to make the bird cross as many obstacles as possible and reach the finish

point which is located after, say N obstacles.

So how does this relate to probabilistic or even deterministic programs, in general? Let us try to answer this question through some formal mathematics. Every computer program, deterministic or probabilistic, will start from some initial state. Even if it is not sure what that initial state is, there is a distribution over the set of all possible states from where the program is likely to start. Thus, if we denote the set (lifted, to include non-deterministic behavior) of all states by $S_\perp$ and it's power set by $\mathcal{P}(S_\perp)$, we can define a function $\phi : \mathcal{P}(S_\perp) \to [0, 1]$, from the power set of states to the continuous interval $[0, 1]$. For the time being, assume that this function is injective, but not surjective (obviously). In case of probabilistic programs, we can replace this power set by the set of all distributions on the set of states. The latter can also be used for deterministic programs, the only difference being that the distributions in this case will be point distributions.

Now that we have the function ϕ ready with us, let us see what it means to move the bird from the start point to the end location by tapping appropriately across the sreen. For building this analogy, let us keep in mind the program of example 4 in the previous section. We will start by the simplest case as illustrated by this example and then generalize our analogy.

bool $done := false$
$k := 1, x := k$
while $\neg done$ **do**
  $done := true \oplus_p \texttt{skip}$
  $k := k + 1, x := x + 1$
end while
Postcondition : $\bigwedge_{n \geq 1}[Pr\{x = n\} = p^{n-1}(1 - p)]$

We see that the value of Boolean variable *done* controls the loop iterations. This value is updated probabilistically inside the loop. With each step, it is set to *true* with probabiliity $(1 - p)$ and remains *false* with probability p. However, the moment this value becomes *true*, the loop condition becomes false and we exit the loop. The program then terminates. Thus, the termination of the whole program depends on the probabilistic selection of the values to be assigned to this Boolean variable. While one choice immediately terminates the program, the other leads to another iteration of the loop, which with a non-zero arbritarily small probability can go on for large number of times. So, will the loop ever execute forever? The answer is, no. The probability that the loop executes n times is given by $p^{n-1}(1 - p)$, which decreaes rapidly with increasing n. However, in the limiting case, this probability becomes zero and hence, the termination probability becomes 1. This implies that the loop is surely not an infinite loop, but for any finitely many iterations of the loop, there is a non-zero probability of another iteration.

Now, is the correctness of program also related to the value of the variable *done*? The answer is clearly, yes. If we observe the postcondition carefully, we require the value of x to be geometrically distributed over the set $\{1, 2, 3, \dots\}$. This means that for every case in which *done* takes the value *true*, we must have this distribution of x over the finitely many values it can take. Hence, the program is correct and this correctness criteria is verified by the exhaustive set of cases when *done* is set to *true*. But where does the flappy bird come into discussion? Assume that the iteration of the loop is analogous to the interval $(0, p)$, which also represents the distribution of the value taken by the variable *done*. Thus, every time we tap on the screen to make the bird change its direction, we are basically making this probabilistic choice of assigning value to the variable *done*. If we tap in the right time, i.e. make the choice corresponding to this interval (assign value $false$ to *done*), the bird is able to cross the current obstacle (the current loop iteration ends and another iteration can now begin) and is ready for the next obstacle. If we tap outside this interval, the bird stops moving and the game ends, i.e. the loop stops execting and the program terminates. Hence, moving the bird from start to its end location is basically analogous to iteraing the loop again and again.

Generalizing this concept now, for every probabilistic choice that we encounter in the program, our interval of tapping changes. This means that every choice we make throughout the program affects the path we take next. While one choice can lead to termination, the other can lead to indefinite computation and hence, the need to make the right choice becomes paramount. So, what is the right strategy to make this decision? The answer is almost impossible to predict in general, but some restricted cases do allow for a solution. Speaking more mathematically, if we assume a function $\theta : X \to X$, for some set X, which in this case is $\mathbb{R}$, with the property that θ is monotonically increasing, i.e. $\theta(x) \geq x$ for all $x \in X$, then this function is analogous to our program execution. The input is the current choice made by the program (the place where we tapped currently) and the output is the most optimal location to tap next (most optimal choice to make) so that the program executes in the way it is expected to. Assuming that given an interval of (or a set of) choices, we can make any choice with some probability, let a function $T : X \times X \to X$ be such that it tells us where in the interval X, we actually tapped, i.e. what choice we actually made. Clearly, we have $start(X) \leq T(X) \leq end(X)$, where *start* and *end* correspond to boundaries of this interval.

Now, once this choice has been made and a tap has been done, the movement of the bird is captured through the final function $\psi : X \times X \to X \times X$, defined by $start(\psi(X)) = \theta(T(X))$ and $end(\psi(X)) = \theta(end(X))$. Thus, we now ask ourselves these two questions:

1. For our case, where $X = \mathbb{R}$, what is the value of $\lim_{n\to\infty} \psi^n(x_1, x_2)$, given some value of x_1 and x_2?
2. What is the value of $Pr\{(a, b) \subseteq \psi^n(x_1, x_2)\}$ for given a, b, n, x_1 and x_2?

The first question tells us the limiting interval given that we start by tapping in the interval (x_1, x_2), i.e. it tells us the limiting behavior of the program given the initial set of choices that we make for each probabilistic statement. The second question computes the probability that a given interval (a, b) lies in the limiting interval after n taps, i.e. the probability that a given set of states is in the set of final states in which the program will terminate after n iterations. Thus, answering these two questions can greatly help us study the verification criteria along with the termination properties of a given program.

Reading List

[1] Ackerman, N. L., Freer, C. E., and Roy, D. M. On the computability of conditional probability. *arXiv preprint arXiv:1005.3014* (2010).

[2] Back, R.-J., and Wright, J. *Refinement calculus: a systematic introduction*. springer Heidelberg, 1998.

[3] Den Hartog, J., and de Vink, E. P. Verifying probabilistic programs using a hoare like logic. *International Journal of Foundations of Computer Science 13*, 03 (2002), 315–340.

[4] Di Pierro, A., and Wiklicky, H. Probabilistic abstract interpretation and statistical testing. In *Process Algebra and Probabilistic Methods: Performance Modeling and Verification*. Springer, 2002, pp. 211–212.

[5] Freer, C. E., and Roy, D. M. Posterior distributions are computable from predictive distributions. In *AISTATS* (2010), pp. 233–240.

[6] Gulwani, S., and Jojic, N. Program verification as probabilistic inference. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 277–289.

[7] Hurd, J. *Formal verification of probabilistic algorithms*. PhD thesis, PhD thesis, University of Cambridge, 2002.

[8] Hurd, J., McIver, A., and Morgan, C. Probabilistic guarded commands mechanized in hol. *Theoretical Computer Science 346*, 1 (2005), 96–112.

[9] Jifeng, H., Seidel, K., and McIver, A. Probabilistic models for the guarded command language. *Science of Computer Programming 28*, 2 (1997), 171–192.

[10] Jones, C. *Probabilistic non-determinism*. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics., 1990.

[11] Katoen, J.-P., McIver, A. K., Meinicke, L. A., and Morgan, C. C. Linear-invariant generation for probabilistic programs. In *Static Analysis*. Springer, 2011, pp. 390–406.

[12] Keimel, K. Topological cones: Foundations for a domain theoretical semantics combining probability and nondeterminism. *Electronic Notes in Theoretical Computer Science 155* (2006), 423–443.

[13] Kozen, D. Semantics of probabilistic programs. *Journal of Computer and System Sciences 22*, 3 (1981), 328–350.

[14] Kozen, D. A probabilistic pdl. *Journal of Computer and System Sciences 30*, 2 (1985), 162–178.

[15] Mislove, M., Ouaknine, J., and Worrell, J. Axioms for probability and nondeterminism. *Electronic Notes in Theoretical Computer Science 96* (2004), 7–28.

[16] Monniaux, D. Abstract interpretation of probabilistic semantics. In *Static Analysis*. Springer, 2000, pp. 322–339.

[17] Monniaux, D. An abstract monte-carlo method for the analysis of probabilistic programs. In *ACM SIGPLAN Notices* (2001), vol. 36, ACM, pp. 93–101.

[18] Monniaux, D. Backwards abstract interpretation of probabilistic programs. In *Programming Languages and Systems*. Springer, 2001, pp. 367–382.

[19] Monniaux, D. Abstract interpretation of programs as markov decision processes. *Science of Computer Programming 58*, 1 (2005), 179–205.

[20] Morgan, C. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Refinement Workshop, Workshops in Computing. Springer Verlag* (1996).

[21] Morgan, C., McIver, A., and Seidel, K. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems (TOPLAS) 18*, 3 (1996), 325–353.

[22] Plotkin, G. D. A powerdomain for countable non-determinism. In *Automata, Languages and programming*. Springer, 1982, pp. 418–428.

[23] Sernadas, A., Ramos, J., and Mateus, P. Linear algebra techniques for deciding the correctness of probabilistic programs with bounded resources. *Preprint, SQIG-IT and IST-TU Lisbon* (2008), 1049–001.

[24] Smith, M. J. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electronic Notes in Theoretical Computer Science 220*, 3 (2008), 43–59.