

Reasoning about efficiency within a probabilistic μ -calculus

A.K. McIver¹

*Programming Research Group
Oxford University
UK*

Abstract

Expectation-based probabilistic predicate transformers [15] provide a logic for probabilistic sequential programs, giving access to expressions such as ‘the probability that predicate A is achieved finally’. Using expectations more generally however, we can express μ -calculus formulae for the expected path-length of a computation tree. Moreover within an expectation-based μ -calculus such efficiency measures and more conventional (but probabilistic) temporal operators [14] can be related.

Keywords: Probabilistic semantics, predicate transformers, nondeterminism, verification, refinement, imperative programming, time efficiency, complexity.

1 Introduction

Traditional treatments of program correctness do not in general extend to the analysis of efficiency: this is true both for standard and for randomised programs. In the latter case any analysis is especially challenging due to the interaction of probabilistic and nondeterministic choice, a situation commonly arising in distributed systems where the nondeterminism represents a scheduler’s action and often the role of probability is to break symmetry (*e.g.* [19]). The theme of this paper is to show how the expectation transformer model of probabilistic sequential and demonic programs, introduced as a framework for proving correctness [6,15], naturally extends to the calculation of time efficiency.

Lynch et al. [10] and de Alfaro [2] study correctness and efficiency within a framework of probabilistic automata. Both define the expected time to achieve a particular goal by reference to the probability distribution over the paths taken to reach it. Lynch is interested in proof techniques, and she presents a method for composing transitions labelled by both a time and a probability.

¹ The author is supported by the *EPSRC*.

On the other hand de Alfaro is concerned with model checking, and he defines a threshold operator for expected times similar to the probabilistic thresholds of pCTL [5]. The use of a threshold operator provides a means to express statements such as ‘the expected time to reach the goal is at most t ’.

The method we present here takes a dual approach to program semantics: rather than considering directly the probability distribution underlying computation paths, we regard the effect of a program on a ‘random variable’ as the prime observational characteristic. We call the random variables ‘expectations’, and with this view a program transforms expectations in the same way that a non-probabilistic program can be identified with a predicate transformer [3], the latter transforming predicates from post- to pre-conditions. Thus, as a transformer, standard assignments for example induce substitution of variables in expressions over random variables, and probabilistic choice transforms by averaging.

In this kind of program semantics, expected times are expressed naturally as formulae in a μ -calculus extended over a domain of real-valued functions (rather than over predicates). Sharir et al. [20] have a similar approach to program analysis also using least fixed points over real-valued functions, though their computational model is probabilistic only, not treating (demonic) non-determinism.

Our novel interpretation of a quantitative μ -calculus over a model for probabilistic programs has been used independently by Huth and Kwiatkowska [11]. They concentrate on the model checking aspects rather than the calculational possibilities which we exploit in this paper.

In Sec. 2 we describe our computational model, dwelling only briefly on a corresponding operational model, similar to Bianco and de Alfaro’s [1], since our emphasis is on programs as expectation transformers. In Sec. 3 we introduce μ -calculus formulae and some properties for two measures of efficiency; as examples we treat the expected time to reach a set of states and the expected number of visits made to a set of states. Our main concern however is calculation, and in Sec. 4 we show how those measures of efficiency are related to temporal logic operators. The prominence of that relation rests on the fact that efficiency depends on (the lengths of) the computation paths, and that temporal logic is a formalism designed particularly for reasoning about paths in a computation tree. Finally in Sec. 5 we apply our results to calculate the efficiency of a simple probabilistic algorithm abstracted from Rabin’s randomised solution to the choice-coordination problem [19].

We write $f.x$ for function f applied to argument x ; given associative, commutative, idempotent (binary) operator \parallel and function g (of compatible type) we write

$$(\parallel X : \text{Cond}.X \bullet g.X)$$

for the \parallel -composition of elements $g.X$, where X is drawn from its type and further selected under condition Cond . More generally we write $(\sigma X \bullet F.X)$

to bind (in various ways) variable X in function F ; in this paper σ will appear as one of λ , μ or ν . Other notation is introduced as we use it.

2 Expectation transformers

We use a version of a computational model [6,15] which combines both probabilistic choice and nondeterministic choice. The two choices are fundamentally different — behaviourally we argue that nothing is known about the control of the nondeterminism, whereas the distribution underlying the probabilistic choice can be observed after many repeated trials. Mathematically the difference is very apparent: probability is modelled by probability distributions — weights are assigned to each possible outcome — whereas nondeterminism is represented simply by a set of final states. For our computational model we assume a finite state space S , and we define $\mathcal{D}S$, the *discrete probability distributions* over S , to be the subset of functions from S to the non-negative reals \mathbb{R}_{\geq} (sub-)normalised to 1:

$$\mathcal{D}S := \{D: S \rightarrow \mathbb{R}_{\geq} \mid \sum_{s:S} D.s \leq 1\} ,$$

where we write ‘ $:=$ ’ for ‘is defined to be’.

Traditional models of nondeterministic programs (without probability [3]) define programs as functions from (initial) state to set of (final) states, and as discussed above the multiplicity of the set represents nondeterminism. On the other hand Kozen [8] models deterministic probabilistic programs as functions from (initial) state to a distribution over (final) states. Putting the two together thus, we define the space of nondeterministic, probabilistic programs $\mathcal{H}S$ to be the set of functions from (initial) state to *sets* of distributions over final states,²

$$\mathcal{H}S := S \rightarrow \mathbb{P}\mathcal{D}S .$$

As for standard nondeterministic models it is the multiplicity of the set that captures the lack of knowledge in how a choice is made, whilst the probability distribution represents a known measure of probability that is environment independent.

The aim of this paper is to calculate time efficiency using a ‘probabilistic program logic’ derived from a space of monotonic ‘expectation transformers’. Rather than using predicates (boolean valued functions of S) to investigate the result sets of programs, we use Kozen’s idea of *expectations* or real-valued functions of the state space, which we denote by $\mathcal{P}S$. Within $\mathcal{P}S$ the original predicates can still be found as $\{0,1\}$ -valued functions of the state space. We call such expectations *standard* and we say that (standard) G *holds* at a state

² We need to impose certain constraints on the result sets so that for example we can define recursion by a least fixed point over a continuous program-to-program function. The details [6,15] are unnecessary to understand this paper.

s if and only if $G.s = 1$. In what follows we reserve G to denote standard expectations, and use A, B for general (not necessarily standard) expectations.

When using expectations, programs are identified with transformers taking post- to pre-expectations: if from initial state s program $prog$ (in \mathcal{HS}) yields final distribution $prog.s$, then viewed as a transformer it takes post-expectation $post$ to a pre-expectation evaluating at s to $\int_{prog.s} post$, the expectation of random variable $post$ over distribution $prog.s$ — which itself being a function of s is of the same type as $post$.³ Because we are looking for upper bounds on expected path lengths, when nondeterminism is present we interpret it as maximum,⁴ giving the general definition as follows.

Definition 2.1 *Let program $prog$ in \mathcal{HS} take initial states in S to sets of final distributions over S . The greatest pre-expectation of $prog$ with respect to post-expectation $post$ in \mathcal{PS} is defined*

$$wp.prog.post.s \quad := \quad (\sqcup D: prog.s \cdot \int_D post) .$$

□

There is an important special case of Def. 2.1: since when $post$ is standard $\int_D post$ is just the probability that $post$ holds with respect to D , the expression $wp.prog.post.s$ is then the greatest probability that $post$ can hold after execution of $prog$ from s .

We generalise implication, a relation between predicates, to ‘probabilistic implication’ (\Rightarrow), a relation between expectations; it is defined (with its variants) over \mathcal{PS} by

\Rightarrow ‘everywhere no more than’

\equiv ‘everywhere equal to’

\Leftarrow ‘everywhere no less than’ .

For c in \mathbb{R}_{\geq} we denote by \underline{c} the constant expectation evaluating everywhere to c and we define arithmetic operators such as addition, multiplication and maximum (\sqcup) between expectations by lifting them pointwise. For q in $[0, 1]$ we denote by $_q \oplus$ the operator that q -averages: if A, B are in \mathcal{PS} then $A_q \oplus B$ is the expectation that evaluates to $q \times A.s + (1-q) \times B.s$ at state s . Finally we also mention fixed points, which are introduced later. If F a monotone function $\mathcal{PS} \rightarrow \mathcal{PS}$ (with respect to \Rightarrow), we denote by $(\mu X \cdot F.X)$ the least solution in \mathcal{PS} of $F.A \Rightarrow A$. We summarise these definitions in Fig. 1.

³ Most authors write $\int_{s \in S} f(s) d\mu$ for the integral of a function $f: S \rightarrow \mathbb{R}_{\geq}$ over a measure μ on S ; here however we use $\int_{\mu} f(s) ds$ in which s is bound and μ is free [15, p.330]. In any case, $\int_{\mu} f$ is the expected value of f over μ .

⁴ In [15] the nondeterminism was interpreted as minimum, but here we require upper bounds on efficiency, and for this application maximising is therefore more appropriate.

<i>implication</i>	$A \Rightarrow A' \text{ iff } (\forall s: S \cdot A.s \leq A'.s)$
<i>maximum</i>	$(A \sqcup A').s = A.s \sqcup A'.s$
<i>minimum</i>	$(A \sqcap A').s = A.s \sqcap A'.s$
<i>addition</i>	$(A + A').s = A.s + A'.s$
<i>scaling</i>	$(cA).s = c \times A.s$

probabilistic transition

$$\text{For } prog \text{ in } \mathcal{HS}, \quad wp.prog.A.s = (\sqcup D: prog.s \cdot \int_D A)$$

least fixed point

For F a monotone function

$$\text{in } \mathcal{PS} \rightarrow \mathcal{PS}, \quad (\mu X \cdot F.X) = (\sqcap A: \mathcal{PS}; F.A \Rightarrow A \cdot A)$$

Terms A, A' are expectations in \mathcal{PS} , and c is a scalar in \mathbb{R}_{\geq} and s is a state in S . Note also that if t in $\mathcal{PS} \rightarrow \mathcal{PS}$ is healthy (see Fig. 2) then $t = wp.prog$ for some $prog$ in \mathcal{HS} .

Fig. 1. Summary of the semantics of the quantitative logic.

In what follows we concentrate on expectation transformers, *step*, that satisfy the ‘healthiness properties’ in Fig. 2. They are similar to Morgan’s [15] who showed that they characterise the *wp*-behaviours of implementable probabilistic/nondeterministic programs — in short, if t is healthy then $t = wp.prog$ for some program $prog$ in \mathcal{HS} . Note that Kozen’s (only deterministic) logic necessarily satisfies all the properties of Fig. 2, and because it has no nondeterminism, suplinearity can be strengthened to linearity — an equality rather than the weaker inequality given here. Indeed the linearity property of Kozen’s logic follows from the linearity of expectation operators in elementary probability theory, and thus provides the hallmark of determinism. Moreover we regard the plurality of a nondeterministic program’s result set as a ‘generalised distribution’ with a correspondingly generalised expectation operator and as such only satisfying the weaker suplinearity property in Fig. 2.

We now give an interpretation of the language of guarded commands, augmented with probabilistic choice, as healthy expectation transformers. For ordinary program operators the similarity to ordinary predicate transformers is marked, except that the interpretation of nondeterministic choice (\sqcup) is to maximise rather than to minimise whilst the new operator, probabilistic choice (${}_q\oplus$), q -averages the results of its operands. As mentioned earlier

<i>suplinearity</i>	$c_1(t.A_1) + c_2(t.A_2) \Leftarrow t.(c_1A_1 + c_2A_2)$
<i>distribution of constants</i>	$t.(A + \underline{c}) \equiv t.A + \underline{c}$
<i>feasibility</i>	$t.A \Rightarrow \sqcup A$
<i>monotonicity</i>	$A_1 \Rightarrow A_2 \quad \text{implies} \quad t.A_1 \Rightarrow t.A_2$
<i>scaling</i>	$t.(cA) \equiv c(t.A)$
<i>continuity</i>	$t.(\sqcup \mathcal{A}) \equiv (\sqcup A: \mathcal{A} \bullet t.A)$

We denote by A, A_1, A_2 expectations in \mathcal{PS} , by c, c_1, c_2 scalars in \mathbb{R}_{\geq} and by \mathcal{A} a \Rightarrow -directed subset of \mathcal{PS} .

Fig. 2. Summary of properties characterising a ‘healthy’ expectation transformer t .

we can, for example, discover the probability that a program will achieve a post-condition by calculating the pre-expectation applied to a standard post-expectation (corresponding to the post-condition). But a post-expectation can be *any* real-valued function, and in particular if its value at a (final) state encodes the length of the ‘computation path’ extending from an initial state s , then with our above interpretation we find the average over all possible path lengths — the expected path-length — as our pre-expectation at s .

To fix the idea of ‘computation path’ we shall use the name *step* to denote an atomic program generating a computation tree in which transitions correspond to a single invocation of the program. We are interested in calculating the number of those transitions in a computation tree that corresponds either to a specific (terminating) process (as in the next example) or more generally to an eternal looping process. In the latter case we concentrate on segments of the whole tree, and we will in Sec. 3 calculate the expected number of transitions (*step*’s) required to achieve satisfaction of a (state) predicate.

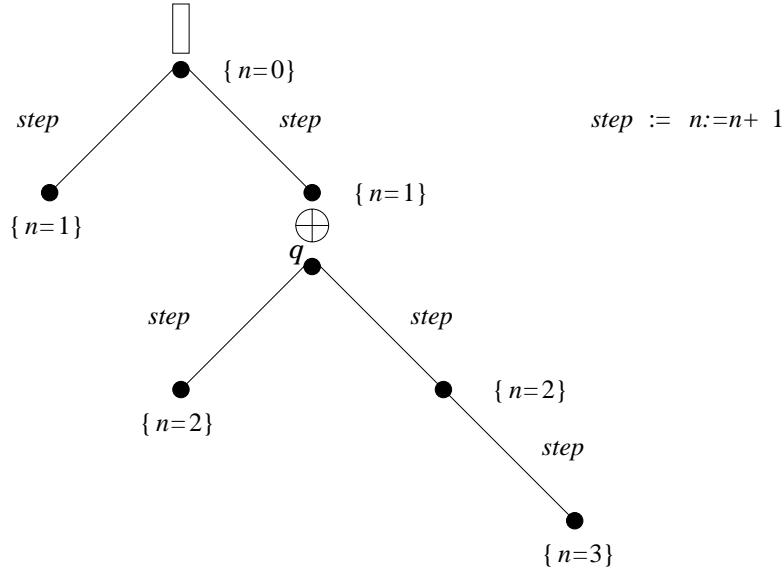
Consider for now a *step* defined simply as

$$step \quad := \quad n := n + 1 ,$$

and (equivalently) the computation tree Fig. 3 corresponding to the program

$$\begin{aligned}
 \text{Tree} &:= \quad step \\
 (1) \quad &\sqcup \quad step; \\
 &\quad \quad \quad step \oplus_q (step; step) .
 \end{aligned}$$

The program *Tree* first nondeterministically chooses either to terminate after a single invocation of *step*, or to terminate after respectively two (probability



We interpret \square as a nondeterministic and $q \oplus$ as a probabilistic branching, the latter selecting the left branch with probability q and the right with probability $1-q$.

Fig. 3. The computation tree for the program *Tree*.

q) or three (probability $1-q$) invocations. The expected number of invocations of *step* in *Tree*, or equivalently the expected length of the computation paths in Fig. 3, is given by $wp.\text{Tree}.n$ where for convenience we write just n for the random variable taking a state s to the value of n it holds. (For example if the *program variable* n in program *step* takes the value 3 at state s , then the *random variable*, also denoted n in the expressions below, evaluates to the *real number* 3 at s .)

We note first that the expectation-transformer semantics of *step* itself is given by

$$(2) \quad wp.\text{step}.(f.n) \equiv f.(n+1) ,$$

where $f.n$ is an expectation over the whole state (and we have chosen to make explicit its dependence on n).

Continuing now from (1) and noting that (as usual) the wp semantics of sequential composition is composition of transformers, we calculate as follows:

$$\begin{aligned}
 & wp.\text{Tree}.n \\
 \equiv & wp.\text{step}.n \sqcup \quad \quad \quad \square \text{ maximises arguments} \\
 & \quad wp.(step; (step_{q \oplus} (step; step))).n \\
 \equiv & (n + \underline{1}) \sqcup \quad \quad \quad (2) \\
 & \quad wp.(step; (step_{q \oplus} (step; step))).n \\
 \equiv & (n + \underline{1}) \sqcup \quad \quad \quad \text{sequential composition} \\
 & \quad wp.\text{step}.(wp.(step_{q \oplus} (step; step))).n \\
 \equiv & (n + \underline{1}) \sqcup \quad \quad \quad q \oplus q\text{-averages arguments} \\
 & \quad wp.\text{step}.(q \times wp.\text{step}.n + (1-q) \times wp.(step; step).n)
 \end{aligned}$$

$$\begin{aligned}
&\equiv (n + \underline{1}) \sqcup \text{wp.step.}(q \times (n + \underline{1}) + (1-q) \times (n + \underline{2})) && \text{sequential composition; (2)} \\
&\equiv (n + \underline{1}) \sqcup (q \times (n + \underline{2}) + (1-q) \times (n + \underline{3})) && (2) \\
&\equiv q \times (n + \underline{2}) + (1-q) \times (n + \underline{3}) . && \text{arithmetic}
\end{aligned}$$

Hence, evaluating the pre-expectation for $n = 0$ initially, we see that the greatest expected path length is $3-q$.

In the following sections we generalise the ideas of this simple example, using the *wp* program logic to express general formulae for calculating the time complexity of arbitrary looping programs.

3 Counting steps

We continue our study of efficiency by considering in general the expected time it takes to achieve some standard predicate G by iterating an (atomic) program *step*. Rather than assuming that *step* already contains an explicit counting variable as we did for (1), for motivation we proceed instead by introducing a fresh variable n whose role is simply to count the number of iterations required to terminate the following loop (compare Hehner's approach [7]):

$$(3) \quad \text{Loop} := \text{do } \overline{G} \rightarrow \text{step}; n := n + 1 \text{ od} ,$$

where $\overline{G} := \underline{1} - G$. The final output of *Loop* is a set of distributions over the new state space $S \times \mathbb{N}$, and if n is initially 0 then its final value records the number of iterations (or the length of the computation tree) until termination. We want to know the *expected value* of n finally and as for (1) we must average over all final distributions of *Loop*. As before that average value is *wp.Loop.n*; taking (3) as our starting point, we seek to eliminate n under the assumption that it is initially 0. For brevity, dropping the prefix '*wp*' and thus equating the language with its *wp* semantics we deduce that *Loop.n* is a fixed point of the transformer

$$(4) \quad (\lambda X \cdot \overline{G} \times n + G \times \text{step}; (n := n + 1).X) .$$

Now continuing, we have

$$\begin{aligned}
&\text{Loop.n} \\
&\equiv \overline{G} \times n + G \times (\text{step}; n := n + 1).(\text{Loop.n}) && \text{from (4)} \\
&\equiv \overline{G} \times n + G \times \text{step.}(\text{Loop.}(n + \underline{1})) && \text{assignment; see below} \\
&\equiv \overline{G} \times n + G \times \text{step.}((\text{Loop.n}) + \underline{1}) . && \text{distribution of constants}
\end{aligned}$$

For the deferred justification we argue that since neither G nor *step* depend on n , we must have the equality

$$n := n + 1; \text{Loop} \equiv \text{Loop}; n := n + 1 .$$

Calculating the weakest pre-expectation with respect to the right hand side and applying the assignment to n yields *Loop.(n + 1)* as required.

Now initialising n to zero, and noting that $step$ does not change n , we see that the expected number of steps E (equal to $Loop.n$ when $n = 0$), satisfies

$$E \equiv \overline{G} \times step.(E + \underline{1}) ,$$

an equation that does not refer to n explicitly; hence we are encouraged to define the number of steps to achieve G as the least fixed point of a function that accumulates.

Definition 3.1 *For a system defined by $step$ and standard expectation G , the worst (largest) expected time to reach G is defined*

$$\Delta G := (\mu X \cdot \overline{G} \times step.(\underline{1} + X)) ,$$

where the least fixed point is taken over the domain of non-negative valued expectations. \square

Def. 3.1, the expectation ΔG is undefined⁵ on those states for which termination of (3) is not guaranteed: along computation paths for which G is never attained, the time to reach G is deemed infinite. In fact the result is only finite at initial states for which ‘almost all’ (probability 1) paths reach G in a finite time, and in such cases there is only one fixed point which makes the use (informally at least) of the least fixed point appropriate.

Our second basic quantity is a measure of the number of times a set of states is visited on repeated invocations of $step$. This time we define a transformer that applies to general expectations, though it specialises to the number of visits when applied to standard ones.

Definition 3.2 *For a system defined by $step$ and for A in \mathcal{PS} , the expected number of visits to A is defined*

$$\#A := (\mu X \cdot A + step.X) .$$

\square

As for Δ , we use the least fixed point to define $\#$ since it correctly calculates the the limit of (increasing) accumulated visits. Def. 3.2 is simpler than Def. 3.1, and being defined for general rather than just standard expectations it satisfies more properties. In particular, as the next lemma shows, the fact that $\#$ satisfies suplinearity implies that it is a generalised expectation operator — though unlike $step$, which averages over generalised distributions of the final state delivered by a single invocation, the generalised distributions associated with $\#$ are over the paths in the (infinite) computation tree.

Lemma 3.3 *$\#$ is monotonic, scaling, suplinear and continuous. Moreover if $step$ is deterministic then for A, B in \mathcal{PS} ,*

$$\#(A + B) \equiv \#A + \#B .$$

⁵ Strictly speaking we are using the completion of \mathcal{PS} , allowing expectations to take ‘infinite’ values [16], though we will persist in referring to such values as undefined over the original definition for \mathcal{PS} .

Proof: We prove only suplinearity — the other properties are proved similarly. Since $\#$ is defined as a least fixed point, we proceed by using the least fixed point property over (the completion of) \mathcal{PS} .⁶ For A, B in \mathcal{PS} we substitute the expression $\#A + \#B$ into the fixed point equation for $\#(A + B)$:

$$\begin{aligned}
& (A + B) + \text{step}(\#A + \#B) && \text{Def. 3.2 for } \#(A + B) \\
\Rightarrow & (A + B) + \text{step}.\#A + \text{step}.\#B && \text{step suplinear} \\
\equiv & (A + \text{step}.\#A) + (B + \text{step}.\#B) \\
\equiv & \#A + \#B . && \text{fixed points for } \#A, \#B
\end{aligned}$$

Hence we deduce that $\#(A + B) \Rightarrow \#A + \#B$.

Note that suplinearity of $\#$ holds even if step is not deterministic. \square

The next lemma gives a relation between Δ and $\#$.

Lemma 3.4 *If G is standard then*

$$\Delta G \quad \Rightarrow \quad \#\overline{G} ,$$

with equality if G is an invariant of step . \square

Lem. 3.4 states that the number of steps it takes to reach G is always at most the number of visits to \overline{G} — each time \overline{G} remains true after invoking step both the number of visits to \overline{G} (so far) and the time taken (so far) to reach G are increased. (If in addition escape from G is impossible, once it is achieved, then a re-visit to \overline{G} is also impossible at which point both the functions in Def. 3.2 and Def. 3.1 stop accumulating — in that case it can be shown that $\#G \equiv \Delta\overline{G}$.)

In this section we have introduced expectation transformers that count the average number of executions of step . In the next section we further investigate the transformers' properties within the broader context of the modal μ -calculus for probabilistic temporal logic generally.

4 Temporal logic and efficiency within a modal μ -calculus

Elsewhere [14,17] we have already generalised the traditional temporal operators ‘eventually’ (\Diamond) and ‘always’ (\Box) to our present context, using predicate transformer-style characterisations [9] and thus giving access to expressions such as ‘the probability that A is established eventually’. We now fit our new operators Δ and $\#$ into place among the others — all are specific μ - or ν -idioms and as such can be readily related.

We begin by recalling the fixed point formulation for ‘eventually’ in the probabilistic context [17, p.14].

Definition 4.1 *For expectation A in \mathcal{PS} and computation defined by step , we*

⁶ The *least fixed point property* is that if f is a monotone function on a domain (D, \leq) and if $f.x \leq x$ then also $\mu f \leq x$, where μf denotes the least fixed point of f .

define eventually A

$$\Diamond A \quad := \quad (\mu X \cdot A \sqcup \text{step}.X) \text{ .}$$

□

Def. 4.1 defines a healthy expectation transformer, moreover it can be shown [17] that $\Diamond G$ (for standard G) evaluated at a state s is the (maximum) proportion of paths rooted at s for which G holds at least once. With that interpretation we can see informally how \Diamond and $\#$ might be related: if it is impossible to reach A from a state s , no matter how many times step executes or how the nondeterminism is resolved, then the expected number of visits made to A starting at s should also be zero. But in fact we find an even tighter relationship between $\#$ and \Diamond , namely that

$$\Diamond(G \times \#G) \equiv \#G \text{ ,}$$

as set out below in Lem. 4.2 (d), which we explain (informally) as follows.

It has been shown that each temporal formula can be interpreted operationally as a program [17], and in addition when the operators are nested in an expression, that the formula corresponds to a sequential composition of programs, one for each operator in the expression. It turns out that $\#$ and Δ have similar operational interpretations and thus an expression $\Diamond(G \times \#G)$ can be interpreted as a program that first of all (since \Diamond is the left-most operator) seeks to satisfy G , and once it does, the continuing behaviour is to calculate the subsequent expected return visits (from there). The equivalence above states that such a program is the same as one which simply calculates the expected number of visits to G directly.

To see how this works, consider for the moment the state space \mathbb{N} over which step is defined

$$\text{step} \quad := \quad n := n - 1 \text{ ,}$$

and let G be the predicate $1 \leq n \leq 10$ — write $[1 \leq n \leq 10]$ for the standard expectation that is 1 when that predicate holds and is 0 otherwise [12]. With this simple example, we can see immediately that

$$(5) \quad \#G \quad \equiv \quad \underline{0} \sqcup n \sqcap \underline{10} \text{ ,}$$

for if $n \leq 0$ initially then G will never be reached; conversely if $n \geq 1$ initially, then G will certainly be reached, continuing to be satisfied until n decreases to zero — taking at most 10 steps to do so — after which G will never be satisfied again. An equally easy observation is that for any expectation A over this step ,

$$(6) \quad \Diamond A \quad \equiv \quad (\sqcup k \leq n \cdot A.k) \text{ ,}$$

because $\Diamond A$ evaluated at a state returns the maximum value of A that repeated invocations of step can reach. Continuing now we have

$$\begin{aligned} & \#G \\ \equiv & \underline{0} \sqcup n \sqcap \underline{10} \\ \equiv & (\sqcup k \leq n \cdot ([1 \leq n \leq 10] \times (\underline{0} \sqcup n \sqcap \underline{10})).k) \end{aligned} \tag{5}$$

$$\equiv \quad \Diamond(G \times \#G) . \quad (6); (5)$$

The next lemma sets out some other existing relationships between the operators $\#$ and Δ .

Lemma 4.2 *For standard expectation G , and state s in S , the following hold:*

- (a) *if G is reachable with certainty from everywhere ($\Diamond G \equiv \underline{1}$) then $\#G$, the expected number of visits to G , is everywhere infinite;*
- (b) *it is impossible to reach G from s ($\Diamond G.s = 0$) if and only if the expected number of visits $\#G.s$ is also zero;*
- (c) *if $\Diamond G.s$, the maximum probability of reaching G from s , is strictly less than 1 then $\Delta G.s$, the expected time to get there from s , is infinite;*
- (d) *the expected number of visits to G is determined by the probability of ever reaching G and the number of return visits to G , once there ($G \times \#G$):*

$$\Diamond(G \times \#G) \equiv \#G ,$$

which is equivalent to the easier

$$(e) \quad \Diamond(\#G) \equiv \#G .$$

□

Note that the implication in (c) can be strengthened to an equivalence if *step* is deterministic, though it does rely on S being finite: for it is well known that in the symmetric random walk on the integers each state will be visited eventually with probability 1, but still the expected time to get there is infinite.

We end this section by investigating $\#G$ in the case that G identifies a particular state s . Thus we define $\{s\}$ in \mathcal{PS} by

$$\begin{aligned} \{s\}.s' &= 1 \quad \text{if } s = s' , \\ &0 \quad \text{otherwise} . \end{aligned}$$

In that case the number of visits to s can be calculated directly from the probability of eventual return.

Lemma 4.3 *For s in S ,*

$$\#\{s\} \equiv \Diamond\{s\}/(1-p) ,$$

where $p := (\text{step}.\Diamond\{s\}).s$ is the probability of ever returning to s .

Proof: We suppose that $\#\{s\}$ is defined everywhere (for otherwise we have $(\text{step}.\Diamond\{s\}).s = 1$ making $1-p$ zero and then the lemma holds trivially). Let $N := \#\{s\}.s$ be the expected number of visits to $\{s\}$ starting from s . Writing cA for the scalar multiplication of expectation A by scalar c , we now reason

$$\begin{aligned} &\#\{s\} \\ \equiv &\Diamond(\{s\} \times \#\{s\}) && \text{Lem. 4.2 (d)} \\ \equiv &\Diamond(N\{s\}) && \{s\} \times \#\{s\} \equiv (\#\{s\}.s)\{s\} \equiv N\{s\} \\ \equiv &N(\Diamond\{s\}) . && \Diamond \text{ scales} \end{aligned}$$

Now substituting $N(\Diamond\{s\})$ directly into the least fixed point equation for $\#\{s\}$ (Def. 3.2) we have

$$\begin{aligned} N(\Diamond\{s\}) &\equiv \{s\} + \text{step}.(N(\Diamond\{s\})) \\ \text{hence } N(\Diamond\{s\}) &\equiv \{s\} + N(\text{step}.\Diamond\{s\}) . \end{aligned} \quad \text{step scales}$$

Feasibility of \Diamond and step then gives us that

$$\text{step}.\Diamond\{s\} \Rightarrow \sqcup\{s\} \equiv \underline{1} ,$$

from which we deduce that $\Diamond\{s\}.s = 1$. Now we evaluate both sides of the above equivalence at s ; then putting $p := (\text{step}\Diamond\{s\}).s$ we conclude finally that

$$N = 1 + Np ,$$

which gives $N = 1/(1-p)$ as required. \square

Although the result is well known for deterministic random walks in elementary probability theory [4], the importance of Lem. 4.3 — and of this approach more generally — is to show that it is true in the presence of non-determinism as well.

5 Rabin's choice coordination

We now apply the results of the previous section to analyse a probabilistic algorithm. Choice coordination is a typical problem of distributed networks and may be stated as follows: a set of users (or processors) must come to a unanimous agreement (between several options) given that their only means of communication is via a global variable. The problem is made difficult by the users being unnamed and identical. Rabin [19] uses probability to break the symmetry in the system, and his algorithm terminates with probability 1. We have used the expectation-based probabilistic temporal logic to verify that fact [13], and in this section we analyse the expected time for it to happen.

The probabilistic structure underlying Rabin's solution is very simple, and in Fig. 4 we present the algorithm in a form abstracted to the level of that structure only. We examine the case when the choice is between two options. More detail relating Fig. 4 with Rabin's algorithm is given elsewhere [13].

There are three notable states *decide*, *wait* and *collect*, and the algorithm cycles around them, terminating⁷ (for us) in *collect*. In *decide* one of the users makes a choice the outcome of which either terminates the algorithm (probability 1/2) or returns it to *wait* (probability 1/2). In *wait* an internal step must be taken to get back to *decide* from where one of the users is forced to choose probabilistically to terminate or to return to *decide*.

Denoting by *step* a single invocation of the system in Fig. 4, we calculate $\Delta\{\text{collect}\}$ as follows:

$$\begin{aligned} &\Delta\{\text{collect}\} \\ \equiv &\#(\{\text{decide}\} + \{\text{wait}\}) \end{aligned} \quad \text{collect is invariant, Lem. 3.4}$$

⁷ The notion of 'termination' is applied loosely here — it is modelled as 'skipping forever'.

if *collect* \rightarrow *collect*
if *decide* \rightarrow *wait*_{1/2} \oplus *collect*
if *wait* \rightarrow *decide*

Writing *step* for the above program, we calculate directly the ‘next-time’ properties:

$$\begin{aligned}
 \text{step}.\{ \text{collect} \} &\equiv \{ \text{decide} \} / 2 + \{ \text{collect} \} \\
 \text{step}.\{ \text{decide} \} &\equiv \{ \text{wait} \} \\
 \text{step}.\{ \text{wait} \} &\equiv \{ \text{decide} \} / 2
 \end{aligned}$$

As usual, we write $\{s\}$ for the expectation that is 1 at state s and 0 elsewhere.

Fig. 4. A single step of the probabilistic behaviour of Rabin’s choice coordination.

$$\begin{aligned}
 &\equiv \# \{ \text{decide} \} + \# \{ \text{wait} \} && \text{step is deterministic, Lem. 3.3} \\
 &\equiv \Diamond \{ \text{decide} \} / (1-p) + \Diamond \{ \text{wait} \} / (1-q) , && \text{Lem. 4.3}
 \end{aligned}$$

where in the last step we have defined p and q as

$$p := \text{step}(\Diamond \{ \text{decide} \}).\text{decide} \quad \text{and} \quad q := \text{step}(\Diamond \{ \text{wait} \}).\text{wait} .$$

Of course in principle we would prefer to restrict ‘by inspection’ only to the determination of the immediate next-time properties as set out in Fig. 4 — for that can be regarded as merely transliteration. From there we could, using the methods of [14] prove rigorously the equalities in (7) below. In this case, that would distract from the main argument and we proceed directly (and informally) to calculate those eventualities.

By inspecting Fig. 4 we see that reaching *decide* is impossible from *collect* and certain from *wait*, thus $\Diamond \{ \text{decide} \}$ evaluates to 0 at *collect* and to 1 elsewhere. As for $\Diamond \{ \text{wait} \}$, to reach *wait* from *decide* is determined in a single step — the probabilistic choice resolves either to terminate in *collect* with the result that *wait* is never satisfied (probability 1/2), or alternatively it resolves to satisfy *wait* immediately (also probability 1/2). Thus it follows that 1/2 of all paths rooted from *decide* will satisfy *wait* eventually, which explains the 1/2 multiplier in (7): we have shown that

$$\begin{aligned}
 (7) \quad \Diamond \{ \text{decide} \} &\equiv \{ \text{decide} \} + \{ \text{wait} \} , \\
 \Diamond \{ \text{wait} \} &\equiv \{ \text{decide} \} / 2 + \{ \text{wait} \} .
 \end{aligned}$$

Now continuing, we calculate p and q by working directly from Fig. 4 and the expressions in (7):

$$\begin{aligned}
 \text{step}(\Diamond \{ \text{decide} \}) &\equiv \{ \text{decide} \} / 2 + \{ \text{wait} \} , \\
 \text{step}(\Diamond \{ \text{wait} \}) &\equiv \{ \text{decide} \} / 2 + \{ \text{wait} \} / 2 ,
 \end{aligned}$$

from which we read off p and q as 1/2 and 1/2 respectively. Finally substituting

the expressions in (7) into the expression for $\Delta\{\text{collect}\}$ we see that

$$\begin{aligned}
& \Delta\{\text{collect}\} \\
\equiv & \diamond\{\text{decide}\}/(1-p) + \diamond\{\text{wait}\}/(1-q) \\
\equiv & 2(\diamond\{\text{decide}\}) + 2(\diamond\{\text{wait}\}) & p, q = 1/2, 1/2 \\
\equiv & 2(\{\text{decide}\} + \{\text{wait}\}) + 2(\{\text{decide}\}/2 + \{\text{wait}\}) & (7) \\
\equiv & 3\{\text{decide}\} + 4\{\text{wait}\} .
\end{aligned}$$

Thus on average it takes 3 invocations of the *step* in Fig. 4 starting from *decide*, and (since an extra transition must be made to *decide*) 4 steps from *wait*.

6 Conclusion

In this paper we have made two contributions: the first is to show how using a model of computation based on expectation transformers we are able to obtain expressions for measuring time efficiency of programs within an extended μ -calculus. Secondly we have shown how time efficiency relates to (a still novel form of) probabilistic temporal logic.

Our technique for combining probability and nondeterminism leads to a ‘branching-time’ style semantics. Other approaches to probabilistic temporal logic [5] typically measure the proportion of paths (with respect to a path distribution generated by *step* for example) that satisfy a standard temporal logic path formula; nondeterminism then offers a *range* of behaviours, and one speaks of maximum and minimum probabilities. Here, since we confine ourselves to the simple formulae $\diamond A$, $\#A$ and ΔG , our simpler branching-time semantics is equivalent. For example $(\diamond G).s$ gives exactly the maximum proportion of paths rooted at s for which ‘eventually G ’ holds.

More generally, probabilistic temporal logics do not support the evaluation of arbitrary expectations — de Alfaro [2] specifically examines efficiency, whilst he emphasises model checking rather than proof. One advantage to our axiomatic presentation (compare Fig. 2) is to provide an understanding of the relationships that exist between different operators, as well as providing notational uniformity.

The use of temporal logic here is only a first step. The treatment of the choice coordination problem in Sec. 5 is far too simplified to claim to give an accurate measure of time efficiency for the original algorithm. Being able to separate out underlying probabilistic behaviour however is a good approach to algorithm analysis, and how to relate that abstracted behaviour to the original algorithm is yet a topic for investigation. One possibility would be to assign different costs to the transitions in Fig. 4, which in reality are made up of a number of atomic steps, and moreover the probabilities associated with them are ‘eventual’ probabilities. From the original algorithm, using standard variant arguments we can deduce that each arrow in Fig. 4 must occur after at most $P \times U$ atomic steps where U is the number of users and P is the

number of options they must choose between (in the above example P is 2). Hence we would use an expectation, $2U\{decide\} + 2U\{wait\}$, to calculate the number of atomic steps before termination, but still using Fig. 4 we reason:

$$\begin{aligned} & \#(2U\{decide\} + 2U\{wait\}) \\ \equiv & 2U\#(\{decide\} + \{wait\}) && \text{scaling} \\ \equiv & 2U(4\{decide\} + 3\{wait\}) . \end{aligned}$$

Going one step further, a more accurate way to assign costs would be to use the temporal operators more carefully: for a standard expectation G the expression $\Diamond(\overline{G} \times \Diamond X)$ gives the proportion of paths that eventually satisfy first \overline{G} and then (perhaps sometime later) X . And now it is possible to show

$$\#G \Rightarrow (\mu X \cdot \Delta \overline{G} + \Diamond(\overline{G} \times \Diamond X)) ,$$

allowing us to analyse a simplified system where (as in Fig. 4) each transition represents an eventuality incurring a cost (in the above expression) of $\Delta \overline{G}$.

Finally we are able to combine temporal formulae to prove theorems such as

$$\begin{aligned} (8) \quad & \text{if } A \Rightarrow A \triangleright B \\ & \text{then } A \times \Delta C \Rightarrow \Delta B + \Diamond \Delta C , \end{aligned}$$

which reads ‘if all the paths from A must go through B then the time to reach C is the sum of the time to reach B from A and the time to reach C from B ’. Here we are using the temporal logic formula $A \triangleright B$ (A unless B) [17] which gives the proportion of paths that either always satisfy A or, as soon as they don’t, they satisfy B : the first statement in (8) thus states that all paths from A have that property. Such theorems also await future investigation.

Acknowledgements

This paper reports work carried out within a project supported by the *EPSRC*, whose other members are Carroll Morgan and Jeff Sanders. In particular Def. 3.1 was originally suggested by Carroll Morgan in the general context of probabilistic and nondeterministic programs.

References

- [1] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. *Foundations of Software Tech. and Theor. Comp. Sci.*, LNCS volume 1026:499–513, 1995.
- [2] L. de Alfaro. Temporal logics for the specification of performance and reliability. *Proceedings of STACS ’97*, LNCS volume 1200, 1997.
- [3] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1976.

- [4] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 2. Wiley, second edition, 1971.
- [5] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
- [6] Jifeng He, K. Seidel, and A. K. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2,3):171–192, January 1997.
- [7] Eric C.R. Hehner. *The Logic of Programming*. Prentice-Hall International, 1984.
- [8] D. Kozen. A probabilistic PDL. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, New York, 1983. ACM.
- [9] D. Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [10] N. Lynch, I. Saias, and R. Segala. Proving time bounds for randomized distributed algorithms. *Proceedings of 13th Annual Symposium on Principles of Distributed Algorithms*, pages 314–323, 1994.
- [11] M. Huth and M. Kwiatkowska. Quantitative analysis and model checking. *Proceedings of 12th annual IEEE Symposium on Logic in Computer Science*, 1997.
- [12] C. C. Morgan. Proof rules for probabilistic loops. In He Jifeng, John Cooke, and Peter Wallis, editors, *Proceedings of the BCS-FACS 7th Refinement Workshop*, Workshops in Computing. Springer Verlag, July 1996. <http://www.springer.co.uk/ewic/workshops/7RW>.
- [13] C. C. Morgan and A. K. McIver. The choice-coordination problem: Notes for a correctness proof. See DC96 at <http> [18].
- [14] C. C. Morgan and A. K. McIver. A temporal logic for probabilistic predicate transformers. See MM97 at <http> [18].
- [15] C. C. Morgan, A. K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.
- [16] Carroll Morgan. The generalised substitution language extended to probabilistic programs. To appear in B’98: the Second International B Conference. See B98 at <http> [18], April 1998.
- [17] Carroll Morgan and Annabelle McIver. A probabilistic temporal calculus based on expectations. In Lindsay Groves and Steve Reeves, editors, *Proceedings of Formal Methods Pacific ’97*. Springer Verlag, July 1997. See also PTL96 at <http> [18].
- [18] PSG. Probabilistic Systems Group: Collected reports. <http://www.comlab.ox.ac.uk/oucl/groups/probs/bibliography.html>.

- [19] M. O. Rabin. The choice-coordination problem. *Acta Informatica*, 17(2):121–134, June 1982.
- [20] M. Sharir, A. Pnueli, and S. Hart. Verification of probabilistic programs. *SIAM Journal on Computing*, 13(2):292–314, May 1984.