ELSEVIER

Theoretical Computer Science 282 (2002) 191–219

Theoretical Computer Science

www.elsevier.com/locate/tcs

Quantitative program logic and expected time bounds in probabilistic distributed algorithms

A.K. McIver *, [1]

Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Abstract

In this paper we show how quantitative program logic (Morgan et al., ACM Trans. Programming Languages Systems 18 (1996) 325) provides a formal framework in which to promote standard techniques of program analysis to a context where probability and nondeterminism interact, a situation common to probabilistic distributed algorithms. We show that overall expected time can be formulated directly in the logic and that it can be derived from local properties of components. We illustrate the methods with an analysis of expected running time of the probabilistic dining philosophers (Lehmann and Ravin, Proc 8th Annu. ACM. Symp. on principles of Programming Languages, ACM, New York, 1981, p. 133). © 2002 Published by Elsevier Science B.V.

1. Introduction

Distributed systems consist of a number of independent components whose interleaved behaviour typically generates much nondeterminism; the addition of probability incurs an extra layer of complexity. Our principal aims here are to illustrate how, using 'quantitative program logic' [15], familiar techniques from standard programming paradigms easily extend to the probabilistic context, and that they can be used to analyse not only correctness, but expected running times as well.

Our techniques use a quantitative logic interpreted over a probabilistic transition system in which the expressions are real- (rather than Boolean-) valued functions of the state. The transition system combines nondeterministic choice together with probabilistic choice, making the following distinction between the two. Nondeterministic choice represents some arbitrary selection between several possible programs. The selection

* Fax: +44-1865-273-839.

E-mail address: anabel@comlab.ox.ac.uk (A.K. McIver).

[1] McIver is supported by the EPSRC.

0304-3975/02/$ - see front matter © 2002 Published by Elsevier Science B.V.
PII: S0304-3975(01)00049-4

could depend on many criteria that are either unknown or irrelevant (at least for correctness) and anyway are probably unquantifiable: nondeterminism abstracts from them all. Modelling nondeterminism in this sense is crucial for the analysis of "true concurrency" which is a recognised phenomenon of distributed systems, where the arbitrariness of choice can be traced to unknown timing delays, for example. In this sense its behaviour is different to probabilistic choice: if the former represents "true concurrency", the latter represents "pure randomness", which (unlike nondeterminism) we assume to be both quantifiable and impervious to eccentricities prevailing in the environment.

Other authors [17, 3, 24] treat nondeterminism and probability in this way, and standard models of Markov decision processes [6] also have these attributes, where "nondeterminism" corresponds to the abstraction of "scheduling policies", and indeed others have drawn attention to the similarity [4].

The analysis of such systems (aside form temporal properties) is often operational, whereas our approach has more similarities with Dijkstra/Hoare logic and *wp*-style reasoning. To make the quantitative logic expressive enough to encode information about the probabilistic choices specified in the programs, we provide an interpretation in terms of 'probabilistic Hoare triples' as follows. If A is a real-valued function of the state and *post* is a predicate (over the state) then we say that the probabilistic Hoare triple

$$\{A\}\ prog\ \{post\}$$

is *satisfied* provided that for all intial states s, the program *prog* establishes predicate *post* with probability 'at least $A.s$' [15]. Because of nondeterminism 'at least' is usually the best that we can do, and is a robust estimate against all ways to resolve the nondeterminism. We set out the details in terms of a probabilistic *wp* in Section 2.

There are several advantages to this formal approach. The first is its notational clarity—as in ordinary program logic, properties (even if they are probabilistic) can be described easily. Perhaps of greater consequence though is that the logic provides a meticulous and pragmatic interface between the programming language and the underlying mathematics, and mathematical properties can be deduced directly from the program text. In programs that have nondeterminism and probability, that can very often be an error-prone activity.

An additional and pleasant feature of the quantitative logic is the ease at which standard program techniques extend smoothly. (Examples include compositional reasoning, μ-calculus treatments of temporal logic and fairness which can be found in the general literature [2, 12].) Many proofs reduce to simple arithmetical arguments which apply not only to correctness, but to "performance" as well, since the extended type (of reals rather than Booleans) encourages us to formulate and reason about both correctness and "performance" within the same logical framework.

The techniques here significantly expand the applicability of previous contributions on this theme. Elsewhere [14] we gave μ-calculus-style definitions of expected time for a program to satisfy a set of states but unfortunately the context there is limited and, in particular, is insufficient to be applied to many distributed applications where fairness is a major component to correctness arguments. One contribution of this paper

is to show how to augment those ideas with fairness; we test the ideas on the well-known probabilistic dining philosophers [13] which has been extensively studied in many works [17, 18] though none use Hoare-style logic.

We begin, in Section 2, by outlining the theoretical foundations on which our later analyses depend. We define a model of probabilistic computation which is flexible enough to accommodate the kind of nondeterminism evident in distributed algorithms. Once that is settled, we can 'read off' greatest and least event-probabilities via a probabilistic program logic. In Section 3 we move on to performance, defining it as the the expected number of execution steps (of a fixed program) needed until some goal is realised; we show how this notion can be formulated as a fixed point expression in the logic. Next in Sections 4 and 5 we turn to practicalities and consider how to reason under the imposition of 'weak fairness'; that condition is vital for the correctness of some algorithms however it presents some difficulties for performance. Finally in Section 6, we test the the theory by analysing the randomised dining philosophers [13]; we discover that no more than 34 steps are required for some philosopher to eat, independently of the number seated at the dining table.

We uniformly use S for the state space and $\bar{S}$ for discrete probability distributions over S (normalised real-valued functions of S). We also use ':=' for 'is defined to be' and '.' for function application. We lift ordinary arithmetic operators pointwise to operators between functions: addition ($+$); multiplication ($\times$); maximum ($\sqcup$) and minimum ($\sqcap$). For a real-valued function F and a scalar q we write qF for the function 'the result of F scaled by factor q'. Other notation is introduced as needed.

2. Program logic and estimating probabilities

Operationally, we model probabilistic sequential programs [16, 10] (cf. also [3, 17]) as functions from (initial) state to *sets* of distributions (called 'result sets') over (final) states. Intuitively, that describes a computation proceeding in two (indivisible) stages: a nondeterministic choice, immediately followed by a probabilistic choice, where the probability refers to the possible final states reachable from a given initial state. This is an "abstract" combination of probability and nondeterminism—in principle, all distributed (concurrent) programs can be modelled as an element of $\mathscr{H}S$, thus we use it as a means to test soundness of our proof techniques.

Definition 2.1. The space of probabilistic, demonic programs $(\mathscr{H}S, \sqsubseteq)$ is defined[2]

$$\mathscr{H}\mathscr{S} := S \to \mathbb{P}\bar{S}.$$

[2] In fact, the whole space $S \to \mathbb{P}\bar{S}$ is not appropriate for modelling probabilistic programs, and we find that we must impose closure conditions on the result sets in order to model reasonable computational behaviour; compactness is one such closure, and it guarantees feasibility of recursive definitions. The details of this and the other closure conditions are set out elsewhere [16] and are unnecessary to understand the present context.

We order programs $P, Q \in \mathscr{H}S$:

$$P \sqsubseteq Q \text{ iff } (\forall s : S \cdot P.s \supseteq Q.s).$$

The relation $\sqsubseteq$ (between programs)—*program refinement*—classifies programs in terms of the extent of nondeterministic behaviour. Programs are more refined than others if they affect reduced nondeterminism, and deterministic programs, the most refined programs of all, only engage in pure randomised computations. In this view nondeterminism can be understood as an abstraction of the mechanism underlying the arbitrary selection over some range of probability distributions; however the agent making that selection can only influence the weights of the probabilistic transitions, not their *actual* resolution once those weights have been picked.

An alternative way to understand a program P in $\mathscr{H}S$ is as a Markov decision process. P represents a set of available agent-strategies (or policies), each one corresponds to a probabilistic assignments in $P.s$ for a given initial s. Once the agent has settled on his strategy, the program evolves just like an ordinary Markov process. Nondeterminism can therefore be construed as abstracting from all agent-strategies.

Nondeterminism is often met as 'abstraction' in specifications, where the agent is cast as a programmer as he chooses an implementation; but when the character of the agent is demonic, adapting to the particular 'run-time' environment as he makes his choice, his behaviour is identical to that of an 'adversary scheduler' assumed of many distributed algorithms [17]. We shall discuss this application to schedulers in more detail in Section 4.

Unlike other authors we shall not use the operational model for program analysis; as a concise description of computations it provides an invaluable means to understand the probabilistic context, however in practice, we can do better by using the quantitative program logic introduced elsewhere [16]: it has an exact interpretation over $\mathscr{H}S$ (and therefore relevant in the present computational setting) and (as we shall see) forms the basis for an attractive analytical and specification tool.

The idea, first suggested by Kozen [11] for deterministic programs, is to extract information about probabilistic computations by considering 'expected values'. Ordinary program logic [7] identifies preconditions that guarantee post conditions; in contrast, for probabilistic programs, the *probability*, rather than the certainty of achieving a post condition is of interest, and Kozen's insight was to formulate that as the result of averaging certain real-valued functions of the state over the final distributions determined by the program. Thus, the quantitative logic we use is an extension of Kozen's (since our programs are both nondeterministic and probabilistic) but is similarly based on *expectations* (real-valued functions of the state rather than Boolean-valued predicates). We denote the space of expectations by $\mathscr{E}S(:= S \to \mathbb{R})$, and we define the semantics of a probabilistic program r as $wp.r$, an *expectation transformer* [16].

Definition 2.2. Let $r : S \to \mathbb{P}(\bar{S})$ be a program taking initial states in S to sets of final distributions over S. Then the *least possible*[3] pre-expectation at state s of program r, with respect to post-expectation A in $\mathscr{E}S$, is defined

$$wp.r.A.s := \sqcap \left\{ \int_F A \mid F : r.s \right\},$$

where $\int_F A$ denotes the integral of A with respect to distribution F.[4]

Another way to understand $wp.r.A.s$ is the minimum expected cost over all agent-strategies in a Markov decision process determined by r, and with respect to cost function A; we postpone that discussion until Section 7.

In the special case that A is $\{0,1\}$-valued—a *characteristic* expectation—we may identify a predicate which is *true* exactly at those states where A evaluates to 1, and then the above interpretation makes $wp.r.A.s$ the least possible probability that r terminates in a state satisfying that predicate. In more conventional notation still, Definition 2.2 gives a precise meaning to the (probabilistic) Hoare triple

$$\{wp.r.A\} \quad r \quad \{A\}, \tag{1}$$

where the triple is satisfied if r guarantees that A is established from initial state s with probability at least $wp.r.A.s$.

To economise on notation we often pun a characteristic expectation with its associated predicate, saying that 's satisfies A' when strictly speaking we mean $A.s = 1$. The context should dispel confusion, however. Other distinguished functions are the constants $\underline{1}$ and $\underline{0}$ evaluating everywhere to 1 and 0, respectively, and thus corresponding to *true* and *false*. We also write $\neg A$ for the negation of A (equivalent to $\underline{1} - A$).

By taking the minimum over a set of distributions in Definition 2.2, we are adopting the demonic interpretation for nondeterministic choice, and for many applications it is the most useful, since it generalises the 'for all' modality of transition semantics [1]. Thus if $wp.r.A.s = p$ say for some real p then *all* (probabilistic) transitions in $r.s$ ensure a probability of at least p of achieving A. In respect of program refinement, reducing the nondeterminism increases that least result.

We use *probabilistic implication* $\Rrightarrow$, a relation between expectations, and with its variants $\Lleftarrow$ and $\equiv$, is defined to extend ordinary Boolean implication:

$\Rrightarrow$ 'everywhere no more than'

$\equiv$ 'everywhere equal to'

$\Lleftarrow$ 'everywhere no less than'.

[3] This interpretation is the same as the *greatest guaranteed* pre-expectation used elsewhere [16].

[4] In fact $\int_F A$ is just $\sum_{s:S} \alpha.s \times F.s$ because S is finite and F is discrete [8]. We use the $\int$-notation because it is less cluttered, and to be consistent with the more general case.

scaling	$t.(c' \cdot A) \equiv c' \cdot (t.A)$
constant distribution	$t.(A + \underline{c}) \equiv t.A + \underline{c}$
subadditive	$(t.A_1) + (t.A_2) \;\Rrightarrow\; t.(A_1 + A_2)$
feasibility	$\underline{\sqcap A} \Rrightarrow t.A \Rrightarrow \underline{\sqcup A}$
continuity	$t.(\sqcup \mathscr{A}) \equiv (\sqcup A{:}\,\mathscr{A} \bullet t.A)$
$\Rrightarrow$*-monotonicity*	$A_1 \Rrightarrow A_2$ implies $t.A_1 \Rrightarrow t.A_2$

The constants c, c' are in $\mathbb{R}$ and c' is nonnegative. The expectations A_1, A_2 are in $\mathscr{E}S$. These properties are collectively referred to as sublinearity. A transformer t is sublinear if and only if $t = wp.r$ for some program r in $\mathscr{H}S$ [15].

Fig. 1. Axioms of transformers corresponding to operational programs.

do nothing	$wp.\mathbf{skip}.A := A$
assignment	$wp.(x := E).A := A[E/x]$
probabilistic choice	$wp.(r_p \oplus r').A := p(wp.r.A) + (1 - p)(wp.r'.A)$
nondeterministic choice	$wp.(r \,[]\, r').A := wp.r.A \sqcap wp.r'.A$
sequential composition	$wp.(r; r').A := wp.r.(wp.r'.A)$
conditional choice	$wp.(r \;\mathbf{if}\; B \;\mathbf{else}\; r').A := B \times wp.r.A \;+ \neg B \times wp.r'.A$
refinement	$r \sqsubseteq r'$ iff $(\forall A \in \mathscr{E}S \bullet wp.r.A \Rrightarrow wp.r'.A)$

A is in $\mathscr{E}S$ and E is an expression in the program variables. The expression $A[E/x]$ denotes replacement of variable x by E in A. The real p satisfies $0 \leqslant p \leqslant 1$, and pA means 'expectation A scaled by p'. Finally B is Boolean-valued when it appears in a program statement but is interpreted as a $\{0,1\}$-valued expectation in the semantics.
The semantics is essentially the same as for ordinary predicate transformers [7] except for probabilistic choice, which is defined as the weighted average of the pre-expectations of its operands, and nondeterminism, which is demonic and selects the minimum probability of success.

Fig. 2. Probabilistic *wp* semantics. Nondeterminism is interpreted demonically.

And now the above observations imply that given any A in $\mathscr{E}S$,

$$r \sqsubseteq r' \;\text{ iff }\; wp.r.A \Rrightarrow wp.r'.A. \tag{2}$$

Here, rather than considering $\mathscr{H}S$ directly we study their realisation as transformers, because in principle (again) all distributed programs can be considered as transformers via Definition 2.2. In practice, we shall use *wp* formulations to generate annotations of program text like (1) above, and Fig. 2 sets out how to do that for a simple programming language, suitable for the application we consider here. The axiomatisation of $\mathscr{H}S$ (in terms properties of transformers) is set out in Fig. 1.

Returning now to our application—expected time bounds— upper bounds have more significance; we define the dual of $wp.r$, generalising the 'exists' modality [1], interpreting nondeterminism as maximum and thus angelically. (cf. upper and lower probability estimates [3].)

Definition 2.3. Let $r : S \to \mathbb{P}(\bar{S})$ be a program, taking initial states in S to sets of final distributions over S. Then the *greatest possible* pre-expectation at state s of program r, with respect to post-expectation A in $\mathscr{E}S$, is defined

$$\widetilde{wp}.r.A.s := \sqcup \left\{ \int_F A \mid F : r.s \right\}.$$

The consequences of nondeterminism may seem a little strange at first, for example for characteristic expectation A, we find that $wp.r.A + wp.r.\neg A$ can be less than 1, apparently counter to the experience from probability theory where one would always expect that the probability of the occurrence of either A or $\neg A$ (that is the probability of *any* event) is certain. The usual connection between the probability of A and that of $\neg A$ however can be recovered once we factor in the nature of angelic and demonic nondeterminism: a small calculation from Definition 2.3 and 2.2 yields the equality

$$\widetilde{wp}.r.\neg A + wp.r.A \equiv \underline{1}, \tag{3}$$

which confirms the reasoning that in resolving the nondeterminism to maximise the chance of terminating in $\neg A$ (namely $\widetilde{wp}.r.\neg A$), the chance of terminating in A is at the same time diminished to the minimum possible (namely $wp.r.A$).

An apt illustration of the utility of the logic is revealed in its treatment of nondeterminism—post-expectations are transformed in a goal-directed fashion and the details of mathematical proofs often reduce to simple arithmetic arguments. That last point is formalised by 'sublinearity', summarised in Fig. 1, and the real surprise is that these simple (sub)distribution properties characterise, in logical terms, 'what it is to be a probabilistic computation'; a transformer t is sublinear exactly when it is equal to $wp.r$ for some program r in $\mathscr{H}S$ [16], and theorems about transformers whose proofs appeal only to sublinearity correspond to theorems about programs in $\mathscr{H}S$.

A is in $\mathscr{E}S$ and E is an expression in the program variables. The expression $A[E/x]$ denotes replacement of variable x by E in A. The real p satisfies $0 \leqslant p \leqslant 1$, and pA means 'expectation A scaled by p'. Finally B is Boolean-valued when it appears in a program statement but is interpreted as a $\{0,1\}$-valued expectation in the semantics.

The semantics is essentially the same as for ordinary predicate transformers [7] except for probabilistic choice, which is defined as the weighted average of the pre-expectations of its operands, and nondeterminism, which is demonic and selects the minimum probability of success.

We finish this section with a 'tutorial' exercise in the use of the logic; we consider the program set out in Fig. 3, for which we calculate the least possible probability that the variable b is set to *true* after a single execution. From Definition 2.2 that is given by $wp.\text{Chooser}.\{b = true\}$, where $\{e = v\}$ denotes the predicate (i.e. characteristic expectation) 'e is equal to v'. From Fig. 2 we see that in order to evaluate the

Chooser := $(b := true)$ **if** $\{b = true\}$ **else** $(b := true)[](b := true \ {}_{1/2}\oplus\ b := false)$

If b is *false* initially then it can be either set to *true* unconditionally, or only with probability $1/2$: the choice between those two options is resolved nondeterministically.

Fig. 3. Randomised Chooser with a Boolean-valued variable b.

conditional choice in Chooser, we need to consider each of the options separately—i.e. the case that the program starts with b set to *false*, and the case that it does not.

We calculate the '$b = false$' case first. In this straightforward example, it is easy to see that a strategy that opts for the left branch of the nondeterministic choice results in b being set to *true* with probability 1, whereas the strategy that opts for the right branch sets it to *true* with probability only $1/2$, thus the minimum probability (over all such strategies) must be at least greater than the minimum of these two extremes and so is also $1/2$. The calculation below represents the formal verification of that fact.

$$
\begin{aligned}
& wp.((b := true)[](b := true \ {}_{1/2}\oplus\ b := false)).\{b = true\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{nondeterministic choice} \\
\equiv\ & wp.(b := true).\{b = true\} \sqcap wp.(b := true \ {}_{1/2}\oplus\ b := false).\{b = true\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{assignment} \\
\equiv\ & \{true = true\} \sqcap wp.(b := true \ {}_{1/2}\oplus\ b := false).\{b = true\} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{see below} \\
\equiv\ & \underline{1} \sqcap (wp.(b := true \ {}_{1/2}\oplus\ b := false).\{b = true\}) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{probabilistic choice} \\
\equiv\ & \underline{1} \sqcap (1/2(wp.(b := true).\{b = true\}) + 1/2(wp.(b := false).\{b = true\})) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{assignment} \\
\equiv\ & \underline{1} \sqcap (\{true = true\}/2 + \{false = true\}/2) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{see below} \\
\equiv\ & \underline{1} \sqcap (1/2 \times \underline{1} + 1/2 \times \underline{0}) \\
\equiv\ & \underline{1/2}. \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{arithmetic}
\end{aligned}
$$

For the deferred justifications, we use the equivalences $\{true = true\} \equiv \underline{1}$ and $\{false = true\} \equiv \underline{0}$.

A similar (though easier) calculation follows for the '$b = true$' case, resulting in $wp.(b := true).\{b = true\} \equiv \underline{1}$. Putting the two together with the rule for conditional choice we find

$$wp.\text{Chooser}.\{b = true\} \equiv \{b = true\} + \{b = false\}/2 \tag{4}$$

implying that there is a probability of at *least* $1/2$ of achieving $\{b = true\}$ if execution of Chooser begins at $\{b = false\}$ and of (at least) 1 if execution begins at $\{b = true\}$.

In contrast, we can calculate the *greatest possible* probability of reaching $\{b = false\}$ using Definition 2.3:

$$\left.\begin{array}{ll} & \widetilde{wp}.\text{Chooser}.\{b = false\} \\ \equiv & \quad \underline{1} - wp.\text{Chooser}.(\underline{1} - \{b = false\}) \quad (3) \\ \equiv & \quad \underline{1} - wp.\text{Chooser}.\{b = true\} \;\; b \text{ is Boolean-valued} \\ \equiv & \quad \underline{1} - (\{b = true\} + \{b = false\}/2) \quad (4) \\ \equiv & \quad \{b = false\}/2, \end{array}\right\} \tag{5}$$

yielding a probability of at *most* 1/2 if execution begins at $\{b = false\}$ and 0 if it begins at $\{b = true\}$—there is no execution from (initially) $\{b = true\}$ to (finally) $\{b = false\}$.

To sum up, we have considered minimum and maximum probabilities using wp and $\widetilde{wp}$ for a single execution of a program, demonstrating the two extreme strategies (demonic and angelic) for resolving nondeterministic choice. In the next section we extend this logical viewpoint to temporal-style semantics.

3. Computation trees and fixed points

In this section we consider arbitrarily many executions of a fixed program denoted $\bigcirc$, for that is how many distributed algorithms may be modelled. Later we shall interpret $\bigcirc$ (or $\tilde{\bigcirc}$) as a specific wp.prog (or $\widetilde{wp}$.prog) for some program prog, but for the moment we adopt an abstract view. Ordinary program semantics of such systems are computation trees [1], with each arc of the tree representing a transition determined by $\bigcirc$. A *path* in the tree represents the effect of some number of executions of $\bigcirc$, and is defined by a sequence whose entries are (labelled by) the states connecting contiguous arcs. When (the interpretation of) $\bigcirc$ contains both probabilistic and nondeterministic choices, a set of distributions over computation paths may be generated in the following way. To generate a single such distribution first the nondeterministic choices are resolved according to some strategy, and then the state is updated to be consistent with the remaining probabilistic transitions. This procedure is iterated, possibly by employing different strategies for resolving the nondeterminism on each separate iteration. Clearly, with the nondeterminism removed and replaced by explicit strategies, the probabilities over the finite paths may be calculated directly, and they determine a well-defined probability distribution over all paths. (It is usually called the Borel measure over cones [8], and the essentials of this construction appear in other works [17, 3].) The set of distributions that can be generated from a particular interpretation of $\bigcirc$ can be found by varying the possible ways to choose the strategies. We call the set of distributions so-generated a probabilistic tree (generated by a given interpretation of $\bigcirc$).

Our aim for this section is, as for the state-to-state transition model, to extract probabilistic information about the path-distributions in probabilistic trees by interpreting 'path formulae' (defined with expectation transformers) over the computation trees.

least possible eventually	$\Diamond A := (\mu X \cdot A \sqcup \bigcirc X)$
greatest possible eventually	$\tilde{\Diamond} A := (\mu X \cdot A \sqcup \tilde{\bigcirc} X)$
least possible always	$\Box A := (\nu X \cdot A \sqcap \bigcirc X)$
greatest possible always	$\tilde{\Box} A := (\nu X \cdot A \sqcap \tilde{\bigcirc} X)$
least possible time to B	$\Delta B := (\mu X \cdot \underline{0}$ **if** B **else** $(\underline{1} + \bigcirc X))$
greatest possible time to B	$\tilde{\Delta} B := (\mu X \cdot \underline{0}$ **if** B **else** $(\underline{1} + \tilde{\bigcirc} X))$

B is $\{0,1\}$-valued and A is nonnegative-valued.

Fig. 4. Expectation operators with respect to a distribution over the computation tree generated by $\bigcirc$.

The result of the interpretation however should correspond with the following intuition. Consider a μ-calculus formula ϕ which describes some temporal property say. Another way to think of ϕ is as a function f_ϕ from computation paths to $\{0,1\}$, mapping a path pt to 1 if the path satisfies ϕ and zero otherwise. We call such a function a 'path function'. (For example [23].) Similarly, though more generally, other kinds of formulae may be associated with real-valued path-functions; for example the definition in Fig. 4 gives a μ-calculus-style formula which has such an interpretation that maps paths to the 'distance' until the first time that some predicate (A) is satisfied along the path. (We give some details of this below.) Having introduced the idea of formulae as path-functions it is now easy to give an intuition over probabilistic trees.

The idea is that given a formula ϕ we can construct a function $\bar{f}_\phi$ from path-distributions to the reals given by

$$\overline{f_\phi}.t := \int_t f_\phi,$$

where t is a path-distribution. This gives the expected value of the function f_ϕ with respect to t. When ϕ corresponds to a temporal property $\bar{f}_\phi.t$ calculates the proportion of paths (with respect to t) that satisfy the property; when ϕ corresponds to the formula in Fig. 4 then it calculates the expected path-length until A is reached. Finally, we take the minimum value $\bar{f}_\phi.t$ over all distributions in the computation tree to obtain a 'least expected value'. This, then is our intended interpretation of ϕ over a probabilistic tree.

It turns out that we can approach that interpretation more directly using the quantitative logic. All we need is to interpret $\bigcirc$ as wp.prog (corresponding to the program prog that generates the probabilistic tree), and to find the fixed points with respect to $\Rrightarrow$, in appropriate subsets of $\mathscr{E}S$. We use μ or ν to denote, respectively, least or greatest fixed points; their existence is assured in complete spaces that have least and greatest elements. In particular in the nonnegative reals the least element is 0, whilst for the greatest element we add 'infinity' as a special value. That leads to promoted least and greatest elements in $\mathscr{E}_{\geqslant}S$ (the nonnegative-valued expectations) as $\underline{0}$ and the

constant 'everywhere infinite' function. The partially ordered space $(\mathscr{E}S, \Rrightarrow)$ is also complete (when restricted to expectations with a fixed bound).

The details that this 'logical interpretation' corresponds to the minimum expected value described above is set out elsewhere [14, 15].

Our first example of fixed point definitions occur in our generalisation of the familiar temporal properties 'eventually' and 'always'; it turns out that the standard formulations [15] of both properties have straightforward extensions in the quantitative logic by replacing $\vee$ and $\wedge$ in the ordinary μ-calculus expressions by $\sqcup$ and $\sqcap$, respectively, and interpreting $\bigcirc$ as an expectation transformer. The resulting operators, $\Diamond$ and $\Box$ (set out in Fig. 4), when applied to a characteristic expectation A return the (least possible) probability (rather than the certainty) that eventually or always A holds of the paths in the computation tree. These will be invaluable later for encoding invariant properties of the system.

Next, we introduce our basic time-bound operator, denoted by Δ, which like the other temporal operators, also has a concise fixed point formulation (Fig. 4).[5] If A is characteristic then ΔA returns the expected length of the paths in the computation tree until a state is reached that satisfies the predicate represented by A. In the context of probabilistic programs it corresponds to the expected *number* of (repeated) executions of $\bigcirc$ required until A holds. To make the link between the fixed-point expression[6] for ΔA in Fig. 4 and the expected length of the computation path to reach A we unfold the fixed point once: if A holds it returns 0—no more steps are required to achieve A along the path; otherwise we obtain a '1+'—at least one more step is required to reach A.

We use this to define a function F from paths to reals in the following way. Let pt be a path and '$s \in \neg A$' return 1 if s satisfies $\neg A$ and 0 otherwise. Let *head* and *tail* be the path operators that take the head and tail of a path. F can now be defined as the limit

$$\begin{aligned} F.pt \quad := \\ & \sqcup\{n \geqslant 1 \cdot (head.pt \in \neg A) \times (1+ \\ & \qquad (head\ tail.pt \in \neg A) \times (1 + \ldots ((head\ tail^n.pt \in \neg A)\} \end{aligned}$$

which can be seen to correspond to an unfolding of the function ΔA with $\bigcirc$ interpreted at moving one place along the path. It is clear that F does indeed calculate the minimum distance from the head of pt until A is satisfied.

Finally when interpreted in the quantitative logic, with $\bigcirc$ interpreted as *wp*.prog and the least fixed point with respect to $\Rrightarrow$ then the effect is to calculate the least expected path-length (that is the least expected value $\int_t F$ over all path-distributions t in the probabilistic tree generated by prog). A formal justification is given elsewhere [14].

[5] Its formulation, and the Δ notation, were suggested by Carroll Morgan.

[6] This has equivalent formulation

$$\Delta B \equiv (\mu X \cdot \neg B \times (\underline{1} + \bigcirc X)).$$

feasibility	$\tilde{\Diamond} A \Rrightarrow \underline{1}$
duality	If $A \Rrightarrow \underline{1}$ then $\tilde{\Diamond} A \equiv \underline{1} - \Box(\underline{1} - A)$
invariants	If $I \Rrightarrow \bigcirc I$ and $I \Rrightarrow A$ then $I \Rrightarrow \Box A$

A, A' and I are nonnegative-valued expectations in $\mathscr{E}_{\geqslant} S$.

Fig. 5. Some properties of the path operators.

In general, the μ-calculus expressions correspond to the 'for all' or 'exists' fragments of temporal logic [1] according to whether they are defined with $\bigcirc$ or $\tilde{\bigcirc}$. For example $\Diamond A$ defined with $\tilde{\bigcirc}$ returns the maximum possible probability that a path satisfies eventually A. Also $\tilde{\triangle} A$ gives an upper bound on the number of steps required to reach A.

All the properties of Fig. 5 follow from the definitions under the assumption that $\bigcirc$ is sublinear.

The invariant law deserves special mention, since it extends the notion of ordinary program invariants to the probabilistic context: just as in ordinary program logic, invariants capture a static predicate about the transition system, so probabilistic invariants capture static *probabilistic properties* about the probabilistic transition system.

Definition 3.1. An expectation I in $\mathscr{E}_{\geqslant} S$ is said to be an *invariant* of $\bigcirc$ provided that

$$\bigcirc I \Lleftarrow I.$$

When I takes arbitrary values, the invariant law says that the probability that A always holds along the paths with initial state s is at least $I.s$. This property is fundamental to our treatment of 'rounds' in the next section, where we also present a lengthier account of invariants.

We end this section with another tutorial-style example of a calculation involving $\tilde{\triangle}$. Again we use the program Chooser set out in Fig. 3 above, (hence $\tilde{\bigcirc}$ is interpreted as $\widetilde{wp}$.Chooser), and we wish to calculate $\tilde{\triangle}\{b = true\}$ an upper bound on the expected number of times Chooser must be executed until b is set to *true*. In a simple case where there is a probability of success on each execution (specifically here if Chooser sets b to *true*) elementary probability theory implies that the expected time to success is the result of summing over a geometric distribution; in contrast the calculation below shows how to find that time using our program logic, in which strategies are automatically resolved appropriately. We note first that $\tilde{\triangle}\{b = true\}$ evaluated at '$b = true$' is 0 (for b is already set to *true*). Thus we know that $\tilde{\triangle}\{b = true\} \equiv q\{b = false\}$, for some nonnegative real q which we must determine (where recall that we use qA to mean 'A

scaled by q'). With that in mind, we reason

$$
\begin{array}{lll}
 & \tilde{\triangle}\{b=true\} & \\
 & & \text{definition } \tilde{\triangle}\text{; Fig. 4} \\
\equiv & \neg\{b=true\}\times(\underline{1}+\widetilde{wp}.\text{Chooser}.(\tilde{\triangle}\{b=true\}))\tilde{\triangle}\{b=true\} \equiv q\{b=false\} & \\
\equiv & \{b=false\}\times(\underline{1}+\widetilde{wp}.\text{Chooser}.(q\{b=false\})) & \\
 & & \text{see below} \\
\equiv & \{b=false\}\times(\underline{1}+q(\widetilde{wp}.\text{Chooser}.\{b=false\})) & \\
 & & \text{from (5)} \\
\equiv & \{b=false\}\times(\underline{1}+q\{b=false\}/2)\ . &
\end{array}
$$

For the deferred justification we are using the scaling property (Fig. 1) which also holds for $\widetilde{wp}$. Chooser and which allows us to distribute the scalar q.

Now evaluating at '$b=false$' we deduce from the above equality that

$$q = 1 + q/2,$$

giving $q=2$, and (unsurprisingly) an *upper bound* of 2 on the number of executions of Chooser required to achieve success.

Note that the application of $\widetilde{wp}$. Chooser automatically resolves the nondeterminism to optimise the goal of calculating the maximum time to reach $\{b=true\}$; operationally the effect of that resolution strategy is to select the probabilistic branch rather than the alternative in which b is immediately set to *true*. The calculation above yields the bound that dominates *all* strategies.

So far we have defined the expected time bound for a tree built from an atomic step $\bigcirc$, interpreted as a single program. (In the example it is Chooser.) A seemingly different setting is provided by distributed systems, the subject of this paper. Distributed systems are constructed from a number of individual processors, each independently executing their local programs, albeit as part of some overall collaborative effort. Analysing the system boils down to making sense of the composite behaviour, which is some interleaving of the processors' individual execution steps made in the local programs. The details of the interleaving are managed by the particular scheduling policy chosen as part of the implementation of the system, and it is here that the underlying concurrency in the system reveals nondeterminism.

The first step in the analysis of an algorithm is to abstract from as many of those details as possible whilst maintaining all the schedule-specific restrictions required to preserve correctness. In this paper we impose weak fairness on the scheduler which means that eventually each process is scheduled and is allowed to execute its local program.[7] (This reduces to the usual definition if we assume that all processes are continuously enabled [2].) It is this condition that prevents us using too weak an

[7] Sometimes this is called "fairmerge".

interpretation for $\bigcirc$. For example were we to model every step of the system's computation sequence as a nondeterministic choice over the processors' programs (even though examination of an arbitrary finite fragment of the execution sequence might suggest that behaviour) we would be unable to prove that the dining philosophers' protocol terminated and our estimate of its performance would only be a useless infinite upper bound. We must use a stronger interpretation for $\bigcirc$.

As others do [17], we define a 'round' to be an interval of the *system*'s execution sequence in which each local algorithm has been scheduled at least once, and we count those by associating a round's behaviour with $\tilde{\bigcirc}$ in the expression for $\tilde{\Delta}$.

This interpretation of a round is equivalent to the assumption of (weak) fairness (at the end of a round each process has been scheduled at least once, conversely in a fair system rounds always exist), so by carefully inferring properties of a round only from its abstract formalisation (given below), we shall be able to bound the expected time to eating of the dining philosophers under every implementation of a fair scheduling policy. That is the subject of the next section.

4. Fair adversary schedulers and probabilistic invariants

We consider a distributed system composed of N processors, each one with the ability to execute independently programs $P_1, \ldots, P_N$ respectively, and indeed the system as a whole evolves by independent execution of (an atomic step) of one of the P_i. It is the job of the scheduler to choose the order in which the (atomic steps of the) P_i's are scheduled. An unconstrained scheduler may choose any order at all, whilst one constrained by a notion of fairness has less freedom. In an unconstrained system we allow any P_i to be selected at every step; that is equivalent to modelling an execution step *of the system* as $[]_{1 \leqslant i \leqslant N} P_i$. (Here we see where [] abstracts from the scheduler's choice.) As we have explained, that is too weak for the dining philosophers, because repeated executions of $[]_{1 \leqslant i \leqslant N} P_i$ will produce all possible sequences, the fair as well as the unfair. The next stage is to reduce the scheduler's choice in such a way that only fair sequences are generated. Another way of saying that is to define a program Round such that

(a) if Round terminates then the observed effect on the state is the same as if each P_i has executed at least once, and
(b) Round must terminate.

Any program (Round) satisfying both (a) and (b) represents the minimum requirements on a fair scheduler. Thus if we interpret $\bigcirc$ as *wp*.Round in our operators then the correct time bounds will be calculated, because we will be calculating (correctly) executions of Round rather than of $[]_{1 \leqslant i \leqslant N} P_i$.

There are a number of ways to specify Round (specifically *wp*.Round satisfying (a) and (b). One possibility is to specify exactly the fair execution sequences of the P_i contained in Round; another is to characterise (a) and (b), declare that Round must satisfy them and analyse their consequences. For weak fairness, the latter is the

easiest approach, for as we shall see, it allows us to restrict our analysis to checking invariants.

Other notions of fairness satisfy more complicated invariant properties [20], and probabilistic fairness [5] would need both a probabilistic choice between the programs P_i and a specification of Round, thus would not simplify the analysis in this context.

We impose the two conditions (a) and (b) above separately. Considering (b) first, we define the action of an unconstrained scheduler, which we denote by the program P^*, only confining its activity to terminating sequences. It turns out that it is easy to describe P^* exactly: it is the greatest fixed point of the function

$$\left(\lambda X\cdot \mathbf{skip}[]\left(\mathop{[]}_{1\leqslant i\leqslant N} P_i\right);X\right),$$

since P^* is not disturbed by prefixing with $\mathbf{skip}[](\mathop{[]}_{1\leqslant i\leqslant N} P_i)$ and [] allows the scheduler free choice amongst the P_i at each step. Since it does allow the scheduler free choice P^*, another way to think of it is as the weakest program (with respect to $\sqsubseteq$) that terminates after some arbitrary iterations of the P_i. The next definition formalises this as a transformer and the operational justification is set out elsewhere [15].

Definition 4.1. The weakest program that terminates after some number of executions of $P_1,\ldots,P_n$ is defined by

$$wp.P^*.A := (\mu X \cdot A \sqcap wp.(\mathop{[]}_{1\leqslant i\leqslant N} P_i.X),$$

for any A in $\mathscr{E}S$.

Next, we turn to condition (a) above, and constrain P^*, by selecting those computations that contain at least one execution of each P_i—it is easy to see that (for each i) those sequences have (at least) the same effect as the program $P^*;P_i;P^*$; put another way, we must have that Round is a refinement of $P^*;P_i;P^*$. This then defines a Round.

Definition 4.2. Let Round be some program implementing a scheduling policy of a distributed system defined by the programs $P_1,\ldots,P_N$. We say that Round satisfies the *weak fairness* condition provided that

$$P^*;P_i;P^* \sqsubseteq \text{Round}$$

for all $1\leqslant i\leqslant N$.

Definition 4.2 says, above all else, that the observed effect on the state after execution of any program Round can be explained by arguing that each P_i must have been executed at least once—before or after those executions nothing is known beyond that the state may be changed only by execution of the P_i, hence the sandwiching with P^*. Having settled weak fairness we turn to proofs; our main technique will be 'probabilistic invariants'.

Recall Definition 3.1 that probabilistic invariants supersede the notion of ordinary program invariants in that we have replaced ordinary implication by probabilistic implication. Indeed when I is characteristic, it being a probabilistic invariant means much the same as before: execution of prog from a state satisfying I results in a final state also satisfying I. More interesting are probabilistic invariant properties which often cannot be encoded by characteristic invariants; however since we allow real-valued functions in our expressions those that are characteristic often have simple formulations. An example of the latter kind is the invariance of the function [8]

$$\underline{1/2} \leqslant wp.\text{Chooser}.\{b = true\}, \tag{6}$$

with respect to Chooser. This says that from every state there is at least $1/2$ chance of setting b to *true*. (In fact (6) is equivalent to *true* anyway, and thus is trivially invariant—but the point is how easily the 'there is always a probability $1/2$' can be expressed.) In the next sections we shall see examples of invariants that do not correspond to characteristic expectations, and typically they express properties such as 'there is always a probability of at least $1/2$ of maintaining predicate Q'. Such invariants, though noncharacteristic themselves, satisfy the inequality $I \Rrightarrow Q$.

Returning now to P^*, we see that if I represents an invariant set of states with respect to $[]_{1 \leqslant i \leqslant N} P_i$, then

$$I \Rrightarrow wp.P^*.I, \tag{7}$$

since I cannot be falsified by execution of $[]_{1 \leqslant i \leqslant N} P_i$. In fact (7) holds even when I is a probabilistic invariant. With that observation we can prove the main result of this section: how to deduce properties of Round using invariants.

Lemma 4.3. *If I, I' are probabilistic invariants with respect to $[]_{1 \leqslant i \leqslant N} P_i$, and such that there is some $1 \leqslant i \leqslant N$ such that $wp.P_i.I \Lleftarrow I'$, then $wp.\text{Round}.I \Lleftarrow I'$.*

Proof. We reason as follows

$$\begin{array}{lll} & wp.\text{Round}.I & \\ \Lleftarrow & wp.(P^*; P_i; P^*).I & \text{Definitions 4.2; 2.1; (2)} \\ \equiv & wp.P^*.wp.P_i.wp.P^*.I & \text{sequential composition, Fig. 2} \\ \Lleftarrow & wp.P^*.(wp.P_i.I) & (7) \\ \Lleftarrow & wp.P^*.I' & \text{assumption; monotonicity} \\ \Lleftarrow & I' & (7). \quad \square \end{array}$$

[8] Here we lift $\leqslant$, defined on the reals, (as distinct from $\Rrightarrow$) to a function on expectations: $(a \leqslant b).s = 1$ if $a.s$ is no greater than $b.s$; otherwise $(a \leqslant b).s = 0$.

least possible visits to A	$\#A := (\mu X \cdot (\bigcirc X)$ **if** $\neg A$ **else** $A + \bigcirc X)$
greatest possible visits to A	$\widetilde{\#}A := (\mu X \cdot (\tilde{\bigcirc} X)$ **if** $\neg A$ **else** $A + \tilde{\bigcirc} X)$

A is $\{0,1\}$-valued.

Fig. 6. The expected number of visits.

Lemma 4.3 embodies the progress property inferred by fairness. If an invariant I' holds initially, and if from within that invariant some (helpful) P_i establishes a second invariant I, then I must hold at the end of the round. Fairness guarantees that P_i executes at some stage in the round, and no matter how the round begins if I' holds initially, invariance ensures that it holds at that stage, after which the now established invariant I continues to hold, no matter how the round is completed. A special case of this is that local invariants are also Round invariants; to see it we take I' to be equal to I in Lemma 4.3.

5. Counting rounds

The significance of Lemma 4.3 is that it provides our only tool for dealing with fairness. With it we can estimate upper bounds on $\widetilde{wp}.\text{Round}.A$ by finding an invariant I such that $\underline{1} - I \Rrightarrow A$ and then converting it to an upper bound on $\widetilde{wp}.\text{Round}$ using (3):

$$
\begin{aligned}
& \widetilde{wp}.\text{Round}.A \\
\Rrightarrow\ & \widetilde{wp}.\text{Round}.(\underline{1} - I) \quad \text{monotonicity} \\
\equiv\ & \underline{1} - wp.\text{Round}.I \quad (3) \\
\Rrightarrow\ & \underline{1} - I. \quad I \text{ Invariant; Lemma 4.3.}
\end{aligned}
$$

For our application to performance that is still not enough because it is not the case that $\tilde{\Delta}A$ can be formulated as such a simple function of an invariant; in this section we consider the consequences of using an alternative performance measure that can be.

We introduce a second measure ($\tilde{\#}$) set out in Fig. 6 which counts steps in a different way. As we shall see, it provides the required access to Lemma 4.3 as well as naturally encouraging the prover to divide up the state space into manageable chunks via its *sup*linearity property.

Informally, $\tilde{\#}A$ counts the (maximum) expected number of times that A ever holds along paths in the computation tree. As from ΔA, we can understand $\tilde{\#}A$ by unfolding: if A holds in the current state on a path, it is deemed to be one more visit to A; similarly unfolding the fixed point once reveals a corresponding '1+' in that case. (This operator

suplinearity	$\tilde{\#}(A+B) \Rrightarrow \tilde{\#}A + \tilde{\#}B$
visit-reach	$\tilde{\Delta}\neg A \Rrightarrow \tilde{\#}A$ with equality if $\neg A \Rrightarrow \bigcirc\neg A$
visit-eventually	$\tilde{\#}A \Rrightarrow \tilde{\Diamond}\, A/(1-p)$, where $p := (\sqcup s{:}A \cdot \tilde{\bigcirc}(\tilde{\Diamond}\, A).s)$.

A is $\{0,1\}$-valued. In the visit-eventually rule, p is the greatest possible probability that A is ever revisited; if $p=1$ that upper bound is formally infinite.

Fig. 7. Some properties of expected visits.

is well known from standard probability theory and $\tilde{\#}$ is the generalisation to include nondeterminism.) The new performance operator is related to $\tilde{\Delta}\neg A$ by observing that for characteristic A the number of times A holds on the path is at least as great as the length of the path until $\neg A$ holds. Other properties of $\tilde{\#}$ are set out in Fig. 7. (Note that with this notation we have returned briefly to the abstract notions of Section 3.)

The promised link between performance and local invariants is found in the visit-eventually rule. It generalises a result from Markov processes [8] which says that the expected number of visits to A is the probability that A is ever reached ($\tilde{\Diamond}\, A$) conditioned on the event that it is never *re*visited (probability $1-p$). The importance of the visit-eventually rule in the context of distributed algorithms is that an upper bound on $\tilde{\Diamond}\, A/(1-p)$ may be calculated from upper bounds on both $\tilde{\Diamond}\, A$ and $\tilde{\bigcirc}\tilde{\Diamond}\, A$, both of which are implied by *lower bounds* on invariants, since from Fig. 5 we have $\tilde{\bigcirc}A \equiv \underline{1} - \Box(\underline{1}-A)$, and that $\Box(\underline{1}-A)$ is an invariant.

To see this in action, we consider again Chooser of Fig. 3, and repeat the calculation of $\tilde{\#}\{b=true\}$, but this time taking the indirect route of the visit-eventually rule and interpreting all occurrences of $\tilde{\bigcirc}$ as $\widetilde{wp}$.Chooser. First the visit-reach rule tells us that $\tilde{\Delta}\{b=true\} \equiv \tilde{\#}\{b=false\}$, since $\{b=true\}$ is invariant with respect to Chooser. Next, we use the visit-eventually rule and discover that we need to calculate $\tilde{\Diamond}\{b=false\}$ and $\widetilde{wp}$.Chooser.$\tilde{\Diamond}\{b=false\}$. Since $\{b=true\}$ is an invariant, we deduce immediately that $\tilde{\Diamond}\{b=false\} \equiv \{b=false\}$, from which we can read off $\widetilde{wp}$.Chooser.$\tilde{\Diamond}\{b=false\}$ as $\{b=false\}/2$. That gives $p=1/2$ and an overall upper bound of $2\{b=false\}$, consistent with the exact calculation in Section 3.

Finally, we draw together the notions of rounds, performance and invariants, and we prove a theorem specific to the distributed context; it forms the main result of this section.

Theorem 5.1. *Consider a distributed system defined by processes $P_1,\ldots,P_N$ arbitrated by a fair scheduler. Given an expectation A and local invariants I, I' such that $A \Rrightarrow \underline{1} - I$ and wp*.Round.$I \Lleftarrow I'$, *the maximal possible number of times that A holds* (*after*

execution of Round) $\tilde{\#}A$ *is given by*

$$\tilde{\#}A \Rrightarrow \underline{1/(1-q)},$$

where $q := 1 - (\sqcap s : A \cdot I'.s)$ *and* $\tilde{\bigcirc}$ *is defined to be* $\widetilde{wp}$.Round *in the definition of* $\tilde{\#}$.

Proof. Using the notation of the visit-eventually property of Fig. 7 we see that an upper bound on $\tilde{\#}A$ is given by upper bounds on both $\tilde{\Diamond}A$ and p. By appealing to feasibility (Fig. 5) and arithmetic we deduce immediately that $\tilde{\#}A \Rrightarrow \underline{1/(1-q)}$ for any $q \geqslant p$. All that remains is to calculate the condition on q. We begin by estimating an upper bound for $\widetilde{wp}$.Round.$(\tilde{\Diamond}A)$.

$$\left.\begin{array}{lll} & wp.\text{Round}.(\tilde{\Diamond}A) & \text{Fig. 7} \\ \Rrightarrow & \widetilde{wp}.\text{Round}.(\tilde{\Diamond}(\underline{1}-I)) & \text{monotonicity, Fig. 5; } A \Rrightarrow \underline{1}-I \\ \equiv & \widetilde{wp}.\text{Round}.(\underline{1}-\Box(\underline{1}-(\underline{1}-I))) & \text{duality, Fig. 5} \\ \equiv & \widetilde{wp}.\text{Round}.(\underline{1}-\Box I) & \text{arithmetic} \\ \Rrightarrow & \widetilde{wp}.\text{Round}.(\underline{1}-I) & I \Rrightarrow \Box I\text{; invariants, Fig. 5} \\ \equiv & \underline{1}-wp.\text{Round}.I & \text{Definition 2.3} \\ \Rrightarrow & \underline{1}-I' & \text{assumption} \end{array}\right\} \quad (8)$$

Next we bound q:

$$\begin{array}{lll} & q \geqslant p & \\ \text{if} & q \geqslant (\sqcup s : A \cdot \widetilde{wp}.\text{Round}(\tilde{\Diamond}A).s) & \text{definition } p\text{, Fig. 7} \\ \text{if} & q \geqslant (\sqcup s : A \cdot (\underline{1}-I').s) & (8) \\ \text{if} & q \geqslant 1-(\sqcap s : A \cdot I'.s), & \text{arithmetic} \end{array}$$

as required. □

To show that I' is established after execution of Round, we just need it to be established by one of the P_i.

Corollary 5.2. *Let* $P_1, \ldots, P_N$ *be a distributed system as in Theorem* 5.1, *and let* I, I' *be local invariants and* A *an expectation such that* $A \Rrightarrow \underline{1} - I$ *and* $I' \Rrightarrow wp.P_i.I$ *for some* i. *In that case*

$$\tilde{\#}A \Rrightarrow 1/(1-q),$$

where q *is as in Theorem* 5.1.

Proof. From Lemma 4.3 and the assumptions, we deduce that $wp.\text{Round}.I \Lleftarrow I'$. The result then follows directly from Theorem 5.1. □

In this section we have investigated how to estimate performance of distributed algorithms is terms of counting 'rounds'. The result is the following tractable method: to find the expected time until some predicate A holds, consider instead the expected number of total visits to its complement $\neg A$, which should satisfy some invariant property allowing application of Theorem 5.1. If necessary, to make the task of finding invariants easier, $\neg A$ can be further decomposed using the suplinearity rule. Note that the upper bound on $\tilde{\Delta}A$ achieved with this method will be reasonably good if A is an invariant of $\bigcirc$.

The choice of decomposition (the only creative part of the exercise) of course depends on the details of the particular algorithm; and the only advantage of using the method and techniques suggested here is that the formal proof obligations are straightforward to verify.

6. Performance of the dining philosophers

In this section we illustrate our operators above by considering Rabin and Lehmann's randomised solution [13] to the well-known problem of the dining philosophers. The problem is usually presented as a number of philosophers $P_1,\ldots,P_N$ seated around a table, who variously think (T) or eat (E). In order to eat they must pick up two forks, each shared between neighbouring philosophers, where the ith philosopher has left, right neighbours, respectively, P_{i-1} and P_{i+1} (with subscripts numbered modulo N). A solution to the problem is a distributed protocol guaranteeing that some philosopher will eventually eat (in the case that some philosopher is continuously hungry). Randomisation is used here to obtain symmetry in the sense that philosophers execute identical code—any non-random solution cannot both guarantee eating and be symmetrical [13].

The aim for this section is to calculate upper bounds on the expected time until some philosopher eats, and since we are only interested in the time to eat we have excluded the details following that event. The algorithm set out in Fig. 8 represents the behaviour of the ith philosopher, where each atomic step is numbered. A philosopher is only able to execute a step provided he is scheduled and when he is, he executes exactly one of the steps, without interference from the other philosophers. Fig. 8 then describes a philosopher as follows: initially, he decides randomly which fork to pick up first; next he persists with his decision until he finally picks it up, only putting it down later if he finds that his other fork is already taken by his neighbour. We have omitted the details relating to the shared fork variables, and for ease of presentation we use labels T, E, l, r, etc., to denote a philospher's state, rather than the explicit variables' values they imply. Thus, for example, if P_i is in state L_i or P_{i-1} is in state R_{i-1}, it means the variable representing the shared fork (between P_i and P_{i-1}) has been set to a value that corresponds to 'taken'. The distributed system can now be defined as repeated executions of the program $[]_{1\leqslant i\leqslant N}P_i$, together with the weak fairness condition, Defintion 4.2.

```
1. if  T_i          → l_i 1/2 ⊕ r_i
2. []  (l_i ∨ r_i)  → if  (l_i ∧ ¬R_{i−1}) → L_i
                       []  (l_i ∧ R_{i−1})  → l_i
                       []  (r_i ∧ ¬L_{i+1}) → R_i
                       []  (r_i ∧ L_{i+1})  → r_i
                       fi
3. []  (L_i ∨ R_i)  → if  (L_i ∧ ¬L_{i+1}) → E_i
                       []  (L_i ∧ L_{i+1})  → 𝕃_i
                       []  (R_i ∧ ¬R_{i−1}) → E_i
                       []  (R_i ∧ R_{i−1})  → ℝ_i
                       fi
4. []  (𝕃_i ∨ ℝ_i)  → T_i
   fi
```

The state T_i represents thinking, $l_i(r_i)$ that a philosopher will attempt to pick up the left (right) fork next time he is scheduled, $L_i(R_i)$ that he is holding only the left (right) fork, $\mathbb{L}_i(\mathbb{R}_i)$ that he will put down the left (right) fork next time he is scheduled and E_i that he eats. The use of state as a Boolean means 'is in that state'; as a statement it means 'is set to that state'.

Fig. 8. The ith philosopher's algorithm [13].

Let Eat be the predicate where some philosopher is eating,

$$\text{Eat} := (\exists i \cdot E_i).$$

The performance of the Dining Philosophers is now given by $\tilde{\triangle}$Eat, the expected number of rounds that the algorithm must execute before some philospher eats. And since we are only interested in rounds we interpret $\tilde{\bigcirc}$ as $\widetilde{wp}$.Round.

Our principal tools for calculating $\tilde{\triangle}$Eat will be the visit-reach rule, Theorem 5.1 and suplinearity of $\widetilde{\#}$; we shall use the latter property to decompose the state space into chunks for which guessing invariants to use in Theorem 5.1 (or Corollary 5.2) is easy.

We can identify a number of landmark invariant philosopher configurations which represent significant progress on the way to Eat. In this section we examine only one however, denoted by Block and defined below. In the appendix we give a table of invariants which can be used to construct a full formal proof of the upper bound of performance. All that remains is a verification of the conditions of Theorem 5.1, and those techniques are well illustrated by the calculation of $\widetilde{\#}(\neg\text{Block})$ in Lemma 6.1 below.

Turning now to some details, the invariant Block represents the configuration in which two 'almost adjacent' philosophers are either trying for or have already picked up 'opposite first forks'. 'Almost adjacent' means that the two philosophers (P_i and P_j) are separated by thinkers (T_k holds whenever k lies 'clockwise' between i and j), and 'opposite first forks' means that philosopher i is attempting to pick up, or already has picked up his left fork ($l_i \vee L_i$) whilst philosopher j is in the same situation except in

respect of his right fork $(l_j \vee L_j)$.

$$\text{Block} := (\exists i,j \cdot (l_i \vee L_i) \wedge (r_j \vee R_j) \wedge (\forall k \mid \text{between}(i,k,j) \cdot T_k)) \vee (\exists i \cdot E_i),$$

where

$$\text{between}(i,k,j) = (\exists n,n' \mid n+n' < N \cdot (i+n=k) \,\mathbf{mod}\, N \wedge (k+n'=j) \,\mathbf{mod}\, N),$$

identifies the philosophers lying 'clockwise' between i and j.

To see that Block is invariant, we write it as the union of a *set* of functions B_{ij} given by

$$B_{ij} := (l_i \vee L_i) \wedge (r_j \vee R_j) \wedge (\forall k \mid \text{between}(i,k,j) \cdot T_k),$$

which together are 'almost permuted' by execution by the P_k. Execution either leaves B_{ij} alone, establishes Eat or establishes one of B_{ik} or B_{kj}. Thus their union (which is equal to Block) is left invariant overall.

Notice that the length of a Block can be defined as the shortest distance between almost adjacent philosophers with opposite forks, and the special case that the distance is zero corresponds to the predicate Eat. The dining philosophers algorithm can thus be seen as implementing a procedure that (eventually) reduces that distance once it is established.

In terms of overall performance, we apply suplinearity to divide the problem into two smaller ones.

$$\begin{array}{lll}
 & \tilde{\triangle}\text{Eat} & \\
\equiv & \widetilde{\#}\neg\text{Eat} & \text{Fig. 7} \\
\equiv & \widetilde{\#}(\neg\text{Eat} \sqcap (\text{Block} \sqcup \neg\text{Block})) & \\
\equiv & \widetilde{\#}(\neg\text{Eat} \sqcap \text{Block} \;+\; \neg\text{Block}) & \text{disjoint predicates} \\
\Rrightarrow & \widetilde{\#}(\neg\text{Eat} \sqcap \text{Block}) \;+\; \widetilde{\#}(\neg\text{Block}). & \text{suplinearity; Fig. 7}
\end{array}$$

The result tells us that it is sufficient to estimate upper bounds on the number of returns to ¬Block, (equivalently, since Block is invariant, the time to establish Block) and similarly the time to establish Eat given that Block holds. In the latter case we have indeed reduced the problem since we can consider a smaller set of states (those that satisfy Block). In the remainder of this section we calculate $\widetilde{\#}(\neg\text{Block})$.

Lemma 6.1. $\widetilde{\#}(\neg\text{Block}) \Rrightarrow \underline{22}$.

Proof. Again we decompose the problem. The terms B_i below are defined in Fig. 9:

$$\begin{array}{lll}
 & \widetilde{\#}\neg\text{Block} & \\
\equiv & \widetilde{\#}(\bigsqcup_{0\leqslant i\leqslant 6} B_i) & \text{Fig. 9} \\
\equiv & \widetilde{\#}(\sum_{0\leqslant i\leqslant 6} B_i) & B_i \text{ are pairwise disjoint} \\
\Rrightarrow & \sum_{0\leqslant i\leqslant 6} \widetilde{\#}B_i. & \text{suplinearity, Fig. 7}
\end{array}$$

$$
\begin{aligned}
B_0 &:= (\forall i \cdot T_i)\\
B_1 &:= (\exists i \cdot (l_i \vee L_i \wedge T_{i+1}) \vee (r_i \vee R_i \wedge T_{i-1}))\\
B_2 &:= (\exists i \cdot (l_i \vee L_i \wedge \mathbb{R}_{i+1}) \vee (r_i \vee R_i \wedge \mathbb{L}_{i-1})) \wedge \neg B_1\\
B_3 &:= (\exists i \cdot T_i) \wedge B\\
B_4 &:= (\exists i \cdot \mathbb{R}_i \vee \mathbb{L}_i) \wedge \neg B_3 \wedge B\\
B_5 &:= (\exists i \cdot L_i \vee R_i) \wedge \neg(B_3 \vee B_4) \wedge B\\
B_6 &:= (\forall i \cdot l_i) \vee (\forall i \cdot r_i),
\end{aligned}
$$

where

$$
\begin{aligned}
B &:= ((\forall i \cdot l_i \vee L_i \vee \mathbb{L}_i \vee T_i) \vee\\
&\quad (\forall i \cdot r_i \vee R_i \vee \mathbb{R}_i \vee T_i)) \wedge \neg B_0.
\end{aligned}
$$

$\neg$Block can be decomposed into seven (pairwise) disjoint satisfiable predicates $B_0, B_1, \ldots, B_6$.

Fig. 9. The decomposition of $\neg$Block.

Next, we calculate $\widetilde{\#}(B_i)$ for the various predicates B_i. From Theorem 5.1, for each B_i, we need to find invariants I_i, I_i' satisfying the conditions

$$B_i \Rrightarrow \underline{1} - I_i \tag{9}$$

$$I_i' \Rrightarrow wp.\mathrm{Round}.I_i. \tag{10}$$

The invariants we use are set out in Fig. 10; we verify the conditions only for B_1; the other cases are similar.

For B_1 we take I_1 to be Block and I_1' to be Block $\sqcup\, B_1/2$.

Since we already know that Block is an invariant, we concentrate on Block $\sqcup\, B_1/2$, showing that it is both invariant and that it satisfies (10).

A typical instance of a configuration that implies Block $\sqcup\, B_1/2$ is given by

$$I'_{(i,l)} := \mathrm{Block} \sqcup ((l_i \vee L_i) \wedge T_{i+1})/2$$

and indeed, as we shall see, such configurations account for all of Block $\sqcup\, B_1/2$. The point of identifying $I'_{(i,l)}$ is that it is very easy to verify our desired properties for it. Specifically, we show that $I'_{(i,l)}$ is invariant and that (10) holds in the form

$$I'_{(i,l)} \Rrightarrow wp.\mathrm{Round}.\mathrm{Block}. \tag{11}$$

For example to prove (11), we reason

$$
\begin{aligned}
& I'_{(i,l)}\\
\equiv\ & \mathrm{Block} \sqcup ((l_i \vee L_i) \wedge T_{i+1})/2 \quad \text{definition}\\
\equiv\ & \mathrm{Block} \sqcup wp.P_{i+1}.((l_i \vee L_i) \wedge r_{i+1}) \quad \text{see below}\\
\Rrightarrow\ & \mathrm{Block} \sqcup wp.P_{i+1}.\mathrm{Block} \quad \text{definition of Block; monotonicity}\\
\equiv\ & wp.P_{i+1}.\mathrm{Block}. \quad \text{invariance of Block}
\end{aligned}
$$

For the deferred justification, we use the easy equivalence

$$wp.P_{i+1}.((l_i \vee L_i) \wedge r_{i+1}) \equiv ((l_i \vee L_i) \wedge T_{i+1})/2.$$

A	I	I'	p	$1/(1-p)$
B_0	Block $\sqcup B_1/2$	$I \sqcup B_0/2$	$1/2$	2
B_1	Block	Block $\sqcup B_1/2$	$1/2$	2
B_2	Block $\sqcup$ $(B_0 \sqcup B_1)/2$	$I \sqcup B_2/2$	$1/2$	2
B_3	Block $\sqcup$ $(B_1 \sqcup B_2 \sqcup B_0)/2$	$I \sqcup B_3/4$	$3/4$	4
B_4	Block $\sqcup$ $(B_1 \sqcup B_2 \sqcup B_0)/2 \sqcup$ $B_3/4$	$I \sqcup B_4/4$	$3/4$	4
B_5	Block $\sqcup$ $(B_1 \sqcup B_2 \sqcup B_0)/2 \sqcup$ $(B_3 \sqcup B_4)/4$	$I \sqcup B_5/4$	$3/4$	4
B_6	Block $\sqcup$ $(B_1 \sqcup B_2 \sqcup B_0)/2 \sqcup$ $(B_3 \sqcup B_4 \sqcup B_5)/4$	$I \sqcup B_6/4$	$3/4$	4

Expectations in columns labelled A, I, I' satisfy the relationships $A \Rrightarrow \underline{1} - I$, $I \Rrightarrow wp.\text{Round}.I$ and $I' \Rrightarrow wp.\text{Round}.I$.

Fig. 10. Invariants for Lemma 6.1.

A similar calculation shows that $I_{(i,l)}$ is invariant with respect to $[]_{1 \leqslant i \leqslant N} P_i$, hence we can appeal to Lemma 4.3, using the above instance of P_{i+1} in establishing Block to deduce (11).

Similar reasoning verifies that indeed Round establishes Block from within the dual set of invariants $I_{(i,r)}$.

But now (10) and invariance must also hold for Block $\sqcup B_1/2$, since we have the equivalence

$$\text{Block} \sqcup B_1/2 \equiv \left(\bigsqcup_{(0 \leqslant i \leqslant N)} I'_{(i,l)} \right) \sqcup \left(\bigsqcup_{(0 \leqslant i \leqslant N)} I'_{(i,r)} \right),$$

and since we have shown that those properties belong to the component configurations $I_{(i,l)}$ and $I_{(i,r)}$. Monotonicity of $wp.\text{Round}$ implies the claim.

Finally, we can at last appeal to Theorem 5.1 to obtain a value of $1/2$ for q and therefore an upper bound of $\underline{2}$ for $\widetilde{\#}B_1$.

The invariants for the other B_i can be treated similarly, and are set out in Fig. 10. The last column in that figure gives the expected visit to each subset, giving the overall upper bound of 22.

7. Conclusion

In this paper we have shown how ordinary correctness techniques of distributed algorithms can be applied to probabilistic programs by using quantitative program logic, and that the methods apply even in the evaluation of performance.

This treatment differs from other approaches to performance analysis of probabilistic algorithms [17, 3, 9, 21] in that we do not refer explicitly to the distribution over computation paths; neither do we factor out either the nondeterminism or the probability as a first step; nor even do we analyse the behaviour of the composed system. Instead we use a goal-directed approach, focussing on the property under consideration; the expectation transformer approach automatically resolves the nondeterminism optimally to suit the problem.

Earlier approaches to probabilistic analysis of programs using expectations [22, 11] do not treat nondeterminism and thus are not applicable to distributed algorithms like this at all.

As noted in Section 2, the explicit use of agent-strategies occurs again in decision problems formulated as Markov decision processes [6]. Indeed the similarity between Markov decision processes becomes increasingly tangible when we observe that the key ideas of 'probabilistic action', 'set of of policies' and 'cost' underlying decision processes correspond, respectively, to 'result-distribution', 'nondeterminism' and 'post-expectation' in the quantitative logic. Moreover, the determination of extreme average payoffs corresponds to our demonic (or angelic) interpretation of nondeterminism, finessing an ungainly explicit reference to the policy by "quantifying it away".

To see this more clearly, (as an example) we re-instate the explicit reference to strategies in the program Chooser. Let G be a Boolean-valued function of the state, which will model an agent's strategy. Next we replace the abstract '$P[]Q$' by the Boolean choice 'P **if** G **then** Q' to obtain the following (deterministic) program.

$$(b := true) \textbf{ if } \{b = true\}$$
$$\textbf{else}$$
$$(b := true) \textbf{ if } G \textbf{ else } (b := true \,{}_{1/2}\!\oplus b := false).$$

With nondeterminism removed, the above program is essentially an ordinary Markov process, and for any (cost) function A we can calculate the expected payoff, which we denote by $E.[G.\text{Chooser}].A$, as follows:

$$E.[G.\text{Chooser}].A := \begin{array}{l} A \textbf{ if } \{b = true\} \\ \textbf{else} \\ A \textbf{ if } G \textbf{ else } \displaystyle\int_F A, \end{array}$$

where F is a discrete distribution defined by $F.s := 1/(2 \times |S|)$, and $|S|$ denotes the size of the state space, S.

To see the connection with the *wp* semantics, we observe that

$$\int_F A \equiv wp.(b := true \,_{1/2}\oplus b := false).A$$

and therefore by minimising over all strategies and comparing with Defintion 2.2 we see that

$$wp.\text{Chooser}.A = \sqcap\{E.[G.\text{Chooser}].A \mid G : S \to \{0,1\}\}.$$

Similarly, it is possible to show the correspondence to a general program $\text{prog} \in \mathscr{H}S$, by replacing the nondeterministic choice as a general choice over all strategies. This means that each fixed strategy of the agent turns the program into an ordinary Markov process as in the above example. In this way, we are able to formulate standard decision problems succinctly as expressions in the logic.

Examples include the 'first passage cost problem' [6, p. 28] and the 'expected discounted cost problem' [6, p. 20]. We discuss those in turn.

The first passage cost problem describes the task of calculating the least possible probability of ever reaching a set of states determined by a stated predicate A. The probability is in respect of an underlying transition system defined as a range of policies, dependent on the current state, and each policy selects a (probabilistic) transition. In our notation the problem corresponds to evaluating the expression

$$(\mu X \cdot A \sqcup wp.\text{prog}.X) \tag{12}$$

(Notice that from Fig. 4 that is exactly $\Diamond A$ with $\bigcirc$ interpreted as $wp.\text{prog}.$)

The expected discounted cost problem describes the task of calculating the least possible expected *discounted payoff* incurred in reaching A. Again the expected payoff is in respect of an underlying decision process. The discounted cost is defined to be $p^n A$ if it took n steps to reach A, where $0 < p \leqslant 1$, and it is called the 'discount factor'. As an expression in the quantitative logic, it is a special case of the # operator:

$$(\mu X \cdot \underline{A} + p \times wp.\text{prog}.X).$$

Having seen above that decision problems may be formulated in the quantitative logic, we are encouraged to export more techniques common to program calculi and to look for different and arguably simpler proofs to other results related to the original decision problems. (Examples of program-oriented techniques are 'healthiness conditions', 'refinement' and the 'least fixed point' property.[9])

For example, let us consider the question of the existence of 'optimal strategies' [6, Chap. 3]. For a decision problem formulated in the quantitative logic as an expression $F(wp.\text{prog})$[10] an *optimal strategy* corresponds to finding a deterministic refinement

[9] The least fixed point property is stated as follows: $f : D \to D$ is defined on a partially ordered set $(D, \leqslant)$ and if $f.\alpha \leqslant \alpha$, then $\mu f \leqslant \alpha$.

[10] For example above at (12) our F is

$$(\lambda t \cdot (\mu X \cdot A \sqcup t.X)).$$

prog′ of prog such that $F(wp.\text{prog}') \equiv F(wp.\text{prog})$. (Recall that deterministic programs have no nondeterminism, and thus represent Markov decision processes possessing a unique strategy.)

For the first passage cost problem, we show the existence of an optimal strategy as follows. Note first that the healthiness conditions of $\mathscr{H}S$ guarantee that there is a deterministic refinement $\text{prog} \sqsubseteq \text{prog}'$ such that

$$A \sqcup wp.\text{prog}'.(\Diamond A) \equiv A \sqcup wp.\text{prog}.(\Diamond A) \equiv \Diamond A, \tag{13}$$

where we use $\Diamond A$ for $(\mu X \cdot A \sqcup wp.\text{prog}.X)$; recall that we are interpreting $\bigcirc$ as $wp.\text{prog}$, and that prog only depends on the 'current state' rather than on some history variable. (The existence of an optimal strategy for a single step of the decision process would be an assumption to the original problem.)

Notice that we now have $\Diamond A$ as a fixed point of the monotone transformer $(\lambda X \cdot A \sqcup wp.\text{prog}'.X)$. We can now reason that prog′ represents an optimal strategy for the cost problem as well, namely that $(\mu X \cdot A \sqcup wp.\text{prog}'.X) \equiv (\mu X \cdot A \sqcup wp.\text{prog}.X)$. The argument now takes only four steps:

$$\begin{array}{lll} & (\mu X \cdot A \sqcup wp.\text{prog}'.X) & \\ \Leftarrow & (\mu X \cdot A \sqcup wp.\text{prog}.X) & (2);\ \text{monotonicity} \\ \equiv & \Diamond A & \text{Fig. 4};\ \bigcirc := wp.\text{prog} \\ \Leftarrow & (\mu X \cdot A \sqcup wp.\text{prog}'.X) & (13);\ \text{least fixed point property} \end{array}$$

and we can deduce the required equality.

The existence of an optimal strategy for the discounted cost problem is proved with the same reasoning. It will be interesting to discover whether any other results to decision problems have equally simple proofs in the quantitative logic.

Appendix. More invariants

Besides Block we use the (characteristic) invariants AdjBlock and Double, where

$$\text{Block} \Leftarrow \text{AdjBlock} \Leftarrow \text{Double},$$

and

$$\text{AdjBlock} := (\exists i \cdot (l_i \wedge r_{i+1}) \wedge \neg(\mathbb{L}_{i-1} \wedge \mathbb{R}_{i+2})) \vee \text{Double}$$

$$\text{Double} := (\exists i \cdot (L_i \wedge (r_{i+1} \vee R_{i+1}) \vee (R_{i+1} \wedge (l_i \vee L_i)) \vee (\exists i \cdot (l_i \wedge r_{i+1} \wedge (l_{i-1} \vee L_{i-1} \vee r_{i+2} \vee R_{i+2})) \vee \text{Eat}.$$

From within Block it takes no more than 2 rounds to establish AdjBlock, from there it takes no more than 8 rounds to establish Double and then at most 2 rounds to

$$D_0 := (\exists i \bullet (L_i \wedge (r_{i+1} \vee R_{i+1})) \vee ((L_i \vee l_i) \wedge R_{i+1})) \wedge \neg\text{Eat}$$
$$D_1 := (\exists i \bullet (l_i \wedge r_{i+1}) \wedge ((r_{i+2} \vee R_{i+2} \vee l_{i-1} \vee L_{i-1})) \wedge \neg(D_0 \vee \text{Eat})$$

$\neg$Eat $\wedge$ Double can be decomposed into two (pairwise) disjoint predicates D_0, D_1. The definitions assume that Double holds. The invariants used to calculate the expected number of visits to each D_i are given in the table below.

A	I	I'	p	$1/(1-p)$
D_0	Eat	$I \sqcup D_0$	1	1
D_1	Eat $\sqcup D_0$	$I \sqcup D_1$	1	1

Fig. 11. Invariants for the expected time to Eat from Double.

$$C_0 := (\exists i \bullet (l_i \wedge r_{i+1}) \wedge (T_{i-1} \vee T_{i+2})) \wedge \neg(\text{Double})$$
$$C_1 := (\exists i \bullet l_i \wedge r_{i+1} \wedge \mathbb{R}_{i-1} \vee \mathbb{L}_{i+2}) \wedge \neg(\text{Double} \vee C_0)$$
$$C_2 := (\exists i \bullet l_i \wedge r_{i+1} \wedge R_{i-1} \vee L_{i+2}) \wedge \neg(\text{Double} \vee C_0 \vee C_1)$$
$$C_3 := (\exists i \bullet l_i \wedge r_{i+1} \wedge r_{i-1} \wedge l_{i+2}) \wedge \neg(\text{Double} \vee C_0 \vee C_1 \vee C_2)$$

$\neg$Double$\wedge$AdjBlock can be decomposed into four (pairwise) disjoint predicates $C_0, C_1, \ldots, C_6$. The definitions assume that AdjBlock holds. The invariants used to calculate the expected number of visits to each C_i are given in the table below.

A	I	I'	p	$1/(1-p)$
C_0	Double	$I \sqcup C_0/2$	1/2	2
C_1	Double $\sqcup C_0/2$	$I \sqcup C_1/2$	1/2	2
C_2	Double $\sqcup$ $(C_0 + C_1)/2$	$I \sqcup C_2/2$	1/2	2
C_3	Double+ $(C_0 \sqcup C_1 \sqcup C_2)/2$	$I \sqcup C_3/2$	1/2	2

Fig. 12. The decomposition of $\neg$Double $\wedge$ AdjBlock.

establish eating, together with the time to establish Block (Lemma 6.1) gives a total of 34 rounds expected to observe eating.

The invariants used to establish the expected number of steps needed to satisfy Eat and Double are set out, respectively, in Figs. 11 and 12.

References

[1] M. Ben-Ari, A. Pnueli, Z. Manna, The temporal logic of branching time, Acta Inform. 20 (1983) 207–226.

[2] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988.

[3] L. de Alfaro, Temporal logics for the specification of performance and reliability. Proc. STACS '97, Lecture Notes in Computer Science Vol. 1200, 1997.

[4] L. de Alfaro, Computing minimum and maximum reachability times in probabilistic systems, in: CONCUR 99, Lecture Notes in Computer Science, Springer, Berlin, 1999, pp. 1–11.

[5] L. de Alfaro, From fairness to chance, Electron. Notes in Theoret. Comput. Sci. 22 (1999) http://www.elsevier.nl/locate/encts/volume22.html.

[6] C. Derman, Finite State Markov Decision Processes, Academic Press, New York, 1970.

[7] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall International, Englewood Cliffs, NJ, 1976.

[8] W. Feller, An Introduction to Probability Theory and its Applications, 2nd ed., Vol. 2, Wiley, New York, 1971.

[9] H. Hansson, B. Jonsson, A logic for reasoning about time and probability, Formal Aspects Comput. 6 (5) (1994) 512–535.

[10] J. He, K. Seidel, A.K. McIver, Probabilistic models for the guarded command language, Sci. Comput. Programming 28 (2,3) (1997) 171–192.

[11] D. Kozen, A probabilistic PDL, in: Proc. 15th ACM Symp. on Theory of Computing, ACM, New York, 1983.

[12] D. Kozen, Results on the propositional μ-calculus, Theoret. Comput. Sci. 27 (1983) 333–354.

[13] D. Lehmann, M.O. Rabin, On the advantages of free choice: a symmetric and fully-distributed solution to the Dining Philosophers Problem, in: Proc. 8th Ann. ACM Symp. on Principles of Programming Languages, New York, 1981, ACM, pp. 133–138.

[14] A.K. McIver, Reasoning about efficiency within a probabilistic mu-calculus, Electron. Notes in Theoret. Comput. Sci. 21 (1999). http://www.elsevier.nl/locate/encts/volume22.html.

[15] C. Morgan, A.K. McIver, A probabilistic temporal calculus based on expectations, in: L. Groves, S. Reeves (Eds.), Proc. Formal Methods Pacific '97, Springer, Berlin, 1997. See also PTL96 at http [19].

[16] C.C. Morgan, A.K. McIver, K. Seidel, Probabilistic predicate transformers, ACM Trans. Programming Languages Systems 18 (3) (1996) 325–353.

[17] R. Segala, N. Lynch, I. Saias, Proving time bounds for randomized distributed algorithms, Proc. 13th Ann. Symp. on Principles of Distributed Algorithms, 1994, pp. 314–323.

[18] A. Pnueli, L. Zuck, Verification of multiprocess probabilistic protocols, Distributed Comput. (1) (1986) 53–72.

[19] PSG, Probabilistic Systems Group: Collected Reports. http://www.comlab.ox.ac.uk/oucl/groups/probs/bibliography.html.

[20] J.R. Rao, Building on the UNITY experience: compositionality, fairness and probability in parallelism, Ph.D. Thesis, Univ. of Texas at Austin, Austin, TX, 1992.

[21] A. Israeli, S. Dolev, S. Moran, Analyzing expected time by scheduler-luck games, IEEE Trans. Software Eng. 22 (5) (1995) 429–439.

[22] M. Sharir, A. Pnueli, S. Hart, Verification of probabilistic programs, SIAM J. Comput. 13 (2) (1984) 292–314.

[23] C. Stirling, Local model checking games, in: CONCUR 95, Lecture Notes in Computer Science, Vol. 962, Springer, Berlin, pp. 1–11, 1995, Extended abstract.

[24] M. Vardi, Automatic varification of probabilistic concurrent finite state programs, Proc. 25th Symp. on the Foundations of Computer Science, Portland, Oregon, 1985, pp. 327–338.