Programming-Logic Analysis of Fault Tolerance: Expected Performance of Self-stabilisation

C.C. Morgan[1] and A.K. McIver[2]

[1] Dept. Comp. Sci. and Eng., University of NSW, Sydney 2052 Australia
carrollm@cse.unsw.edu.au
[2] Dept. Computer Science, Macquarie University, Sydney 2109 Australia
anabel@ics.mq.edu.au

Abstract. Formal proofs of functional correctness and rigorous analyses of fault tolerance have, traditionally, been separate processes. In the former a programming logic (proof) or computational model (model checking) is used to establish that all the system's behaviours satisfy some (specification) criteria. In the latter, techniques derived from engineering are used to determine quantitative properties such as probability of failure (given failure of some component) or expected performance (an average measure of execution time, for example).

To combine the formality and the rigour requires a quantitative approach within which functional correctness can be embedded. Programming logics for probability are capable in principle of doing so, and in this article we illustrate the use of the probabilistic guarded-command language (*pGCL*) and its logic for that purpose.

We take self-stabilisation as an example of fault tolerance, and present program-logical techniques for determining, on the one hand, that termination occurs with probability one and, on the other, the the expected time to termination is bounded above by some value. An interesting technical novelty required for this is the recognition of both "angelic" and "demonic" refinement, reflecting our simultaneous interest in both upper- and lower bounds.

1 Introduction

Formal methods establishes correctness of a program (or system) by mathematical methods which have independently been proved sound. Ideally, a formal verification should cover as much of the system's construction as possible: beginning with a specification that is so clear the user can have no doubt of its meaning; and ending with an implementation that is so concrete the manufacturer can have no doubt of how to build it. With the caveat that there always is a gap at either end ("Is this the right specification?" — "Has the implementation been correctly transliterated?"), traditional formal methods concerns itself with so-called "absolute" correctness: a successful formal development ensures (modulo the caveats) that the program will satisfy its user *every time*.

Fault tolerance has a matching traditional form, where the unavoidable failures that reality serves up —in spite of all our efforts— are handled by backup

M. Butler et al. (Eds.): Fault-Tolerant Systems, LNCS 4157, pp. 288–305, 2006.
© Springer-Verlag Berlin Heidelberg 2006

mechanisms, redundancy, *etc.* whose aim is to make that so-called "absolute" correctness in fact as likely as possible. That is, independent of formal methods (and with a much longer history), the techniques of risk- and failure analysis are used to take account of statistical, that is *quantitative* information about possible component-failures and, from it, to derive an estimate about the reliability of the system as a whole.

Our recent work (about ten years [12,10]) has been to address that phrase "independent of formal methods", and the contribution of this article is to illustrate some of the progress that has been made. We choose self-stabilisation as a fault-tolerance paradigm, and show to what extent quantitative behaviour can be *included* in formal reasoning about correctness, rather than being independent of it or an adjunct to it.

Self-stabilisation is a compensating mechanism for systems prone to faults which are either too expensive or impossible to eliminate: when a fault occurs, and is detected, the system automatically takes steps to return itself to a state from which the fault has been removed. The "fault-free" state is considered *stable* in the sense that an absolute-correctness argument has established (or is supposed to have established...) that the system will not itself introduce faults through programming error.

The context for self-stabilisation is usually algorithms which are physically distributed, and "good style" generally dictates that the stabilisation process be symmetric and (hence) to some extent randomised deliberately. Symmetry is to avoid "weak links" whose failure on their own could bring down the whole system; but that symmetry itself introduces a problem because the stable configurations are asymmetric — and only randomisation can take a symmetric system to an asymmetric one.[1]

There are two especially important aspects of randomised algorithms: with what probability are they correct; and how long should we expect them to take. The technical theme of this paper is to show how to deal with those issues in a programming logic, *i.e.* formally. In particular, we investigate the following:

1. The theoretical foundations for reasoning *at the source-code level* about worst-case, *i.e.* upper bounds for expected performance of random algorithms;
2. A sound program-logic rule for estimating those bounds;
3. Practical techniques for using annotations to prove the bounds; and
4. Two case studies illustrating the techniques in action.

One case study deals with expected time to termination (where termination itself is obvious); the other, a more complicated situation, concentrates on showing termination itself.

A key methodological aspect is the prominent role of refinement in our analyses: rather than proving performance properties of "direct" representations of

[1] That is why coins are used in cricket matches: the symmetric state is that the two teams have equal right to bat first; but the outcome —where just one team does so— is asymmetric, and is brought about by the coin flip.

the algorithms, we prove properties of their abstractions. Working with abstractions makes the reasoning more tractable but, most importantly, relies on the properties' being preserved by refinement. That means of course that the refinement rules must be carefully formulated to do that, depending on the properties in question; in our case here, that accounts for our use of angelic nondeterminism when in Sec. 3 we are trying to preserve upper- (rather than the more usual lower) bounds.

We use the these notations. Function f applied to argument x is written $f.x$, where the dot "." is left-associative. This allows for example $f.g.x$ rather than $(f(g))(x)$.

A discrete probability distribution d over a set X is a one-summing function from X into $[0,1]$, thus assigning probability $d.x$ to point x.

For some $x \in X$ the *point probability distribution* "x with probability one" is written $\overline{x}$; for a subset $X' \subseteq X$ the *characteristic function* taking 1 on X' and 0 on the remainder $X{-}X'$ is written $[X']$.

Under *abuses of notation* we collect the following: for the characteristic function of a point we write $[x]$ rather than $[\{x\}]$; for the probability of a set we write $d.X'$ rather than $\sum_{x \in X'} d.x$; for the expected value of a function over a distribution we write $d.f$ rather than $\sum_{x \in X}(d.x \times f.x)$.

Where context supplies unambiguously a predicate language for describing subsets, we write predicates directly for the subsets they denote. Thus for example if X is a state space and d a distribution over it, and (say) for some variables a, b the predicate $a > b$ denotes a subset of X' of X, then we write freely $a > b$ where X' might be expected — whence $d.[a{>}b]$ is the probability that $a > b$ holds in distribution d over X.

2 Performance-Style Properties in *pGCL*

When systems operate within random contexts their properties can no longer be guaranteed absolutely, but only up to some probability. The program fragment

$$x := 0 \;\; {}_{1/4}\oplus \;\; x := 1 \;, \tag{1}$$

for example, does not guarantee to set variable x to 0 under any (initial) condition — the probabilistic choice operator "${}_{1/4}\oplus$" describes the flip of a $(1/4, 3/4)$-biased coin, so that operationally *either* 0 or 1 will be observed, but it is impossible to predict which. The only guarantee is probabilistic, in this case that "with *probability* 1/4, x will be set to 0 if the program fragment is executed". What this means in practice is that over a large number of experiments, the ratio of recorded 1's and 0's will be approximately 3, up to statistical confidence measures [16].

A formal description of that behaviour —the operational semantics— takes the form of a transition-system model of programs combined with probability. The model characterises program execution as causing the state to change, though for probabilistic programs the precise state change can be decided by a coin flip. Thus an operational model for a probabilistic program is a function

which maps an (initial) state to a (set of) *probability distributions* over final states. For example the program at (1) above maps any initial state s to a single result distribution d where $d.s_0 = 1/4$ and $d.s_1 = 3/4$. (Here s_0 and s_1 are states in which "$x = 0$" and "$x = 1$" respectively, but otherwise agree with s.) Given the details of the model we can, for example, determine the probability with which the above property "x is set to 0 finally" is established when the program executes: all we need do is evaluate $d.[x = 0]$, where d is the distribution of final states of the program, since from standard probability theory it is the probability that the predicate "$x = 0$" holds with respect to d. In this case the answer is 1/4.

Although the operational semantics is indeed a faithful model of program behaviour, in practice —from a prover's perspective— it is too complicated to use as the basis for deriving properties of any intricacy. This becomes apparent when general program features are included in the the programming language, such as Boolean choice, nondeterminism, sequential composition and iteration. Better is to use the dual semantics —the so-called expectation transformers— which focusses directly on program properties, rather than on the details of the probabilistic transitions which imply them.

We use the *expectations* as a generalisation of predicates; they are defined to be the set of real-valued functions $\mathcal{E}S$ from the state space S to the reals $\mathbb{R}$, and they are ordered by lifting $\leq$ so that we say $A \Rrightarrow A'$ if, for all $s \in S$, we have $A.s \leq A'.s$. They generalise Boolean predicates if the latter are considered as characteristic functions $S \rightarrow \{0, 1\}$ with false being zero and true one, in which case $\Rrightarrow$ generalises $\Rightarrow$ as well.

To appreciate the duality we rationalise the above calculation, this time concentrating on properties rather than transitions. First of all, we use expectations to express properties rather than predicates. This immediately allows us to regard programs as *transforming* expectations consistent with their operational semantics. We write $\underline{wp}.(x := 0 \;{}_{1/4}{\oplus}\; x := 1)$ for the expectation transformer associated with (1), which must now be defined in such a way that it *transforms* the *post-expectation* $[x = 0]$ to the *pre-expectation* 1/4; more precisely we say that

$$1/4 \quad \equiv \quad \underline{wp}.(x := 0 \;{}_{1/4}{\oplus}\; x := 1).[x = 0] \;.^2$$

In general, if Prog is a program, *PostE* a post-expectation, and s an initial state, then $\underline{wp}.\mathsf{Prog}.\mathit{PostE}.s$ is defined to be the "greatest guaranteed expected value of *PostE* with respect to the result distributions of program Prog if executed from initial state s". We often make use of the familiar Hoare-triple format to say the same thing for all initial states at once; thus we would write equivalently

$$\{\mathit{PreE}\}\ \mathsf{Prog}\ \{\mathit{PostE}\} \;. \tag{2}$$

We say that Prog has been *correctly annotated* with a *pre-expectation PreE* and *post-expectation PostE* just when $\mathit{PreE} \Rrightarrow \underline{wp}.\mathsf{Prog}.\mathit{PostE}$.

The full definition of $\underline{wp}$, as a mapping from program texts to to expectation transformers, is set out at Fig. 1. We use the small programming language

[2] The underline is an indication that choice is interpreted demonically.

pGCL [11] an extension of *GCL* [3] with probabilistic choice. The definitions are almost identical to the Dijkstra's original predicate transformers, the difference being that we use a domain of expectations based on the $\Rrightarrow$ order, rather than predicates and implication. This means, conveniently, that the only apparent differences are that the definitions use arithmetical- rather than Boolean operators. Nondeterministic choice, for example, takes the minimum of its two arguments. The new operator *probabilistic choice* is parametrised by a real $0 \leq p \leq 1$ and takes the p-weighted average of its arguments.

skip	$\underline{wp}.\mathbf{skip}.A \mathrel{\widehat{=}} A$,
abort	$\underline{wp}.\mathbf{abort}.A \mathrel{\widehat{=}} 0$,
assignment	$\underline{wp}.(x := E).A \mathrel{\widehat{=}} A[E/x]$,
sequence	$\underline{wp}.(r; r').A \mathrel{\widehat{=}} \underline{wp}.r.(\underline{wp}.r'.A)$,
probability	$\underline{wp}.(r\ {}_p\!\oplus r').A \mathrel{\widehat{=}}$ $p * \underline{wp}.r.A + (1{-}p) * \underline{wp}.r'.A$,
nondeterminism	$\underline{wp}.(r \mathbin{[\!]} r').A \mathrel{\widehat{=}} \underline{wp}.r.A \sqcap \underline{wp}.r'.A$,
Boolean choice	$\underline{wp}.(\mathbf{if}\ B\ \ \mathbf{then}\ r\ \ \mathbf{else}\ r'\ \mathbf{fi}).A \mathrel{\widehat{=}}$ $[B] * \underline{wp}.r.A + [\neg B] * \underline{wp}.r'.A$,
iteration	$\underline{wp}.(\mathbf{do}\ B \rightarrow r\ \mathbf{od}).A \mathrel{\widehat{=}}$ $(\mu X \cdot [B] * \underline{wp}.r.X + [\neg B] * A)$.

A is an expectation, E is an expression in the program variables, and a term $(\mu X \cdot f.X)$ refers to the least fixed point of expectation-to-expectation function f with respect to $\Rrightarrow$. These definitions are dual to an operational model based on the state-to-distribution semantics [12]. We define (demonic) program refinement so that $\underline{wp}$-properties are preserved.

$$r \sqsubseteq r' \quad \mathit{iff} \quad (\forall A : \mathcal{E}S \mid \underline{wp}.r.A \Rrightarrow \underline{wp}.r'.A) \ .$$

Fig. 1. Structural definitions of $\underline{wp}$ for *pGCL*

Nondeterminism is distinguished from probabilility in the program model — unlike probability it represents truly *unquantifiable* uncertainty present in the system. This distinction leads to a logic of programs based on arithmetical properties of transformers, in which the presence of nondeterminism can be characterised by the failure to distribute addition. In Fig. 2 we set out the full transformer logic; the rules play the part of the "healthiness conditions" used by Dijkstra in his original presentation of the predicate transformers. Intuitively they characterise "legal computations" — mathematically they define the common rules satisfied exactly by $\underline{wp}$−images of programs [12]. For practical purposes this kind of "completeness" means that the prover is at liberty to appeal to any rule in Fig. 2 without disturbing the integrity of his proof.

$$\begin{array}{ll} \textit{subadditivity} & \underline{wp}.\mathsf{Prog}.(A+B) \Leftarrow \underline{wp}.\mathsf{Prog}.A + \underline{wp}.\mathsf{Prog}.B \ , \\ \textit{scaling} & \underline{wp}.\mathsf{Prog}.(k*A) \equiv k * \underline{wp}.\mathsf{Prog}.A \ , \\ \textit{constants} & \underline{wp}.\mathsf{Prog}.(A \ominus k) \Leftarrow \underline{wp}.\mathsf{Prog}.A \ominus k \ . \end{array}$$

A, B are expectations, k is a non-negative real, and Prog is a program. The function $\ominus$ is defined by

$$A \ominus k \quad \widehat{=} \quad (A-k) \sqcup 0 \ .$$

Fig. 2. Axioms of the expectation transformer logic [12]

The decision to interpret nondeterministic choice as the minimum applies when lower bounds on guarantees are sought: one typically proves that a program establishes a postcondition with *at least* some probability. In standard logic this is reduces to the usual *total* correctness, where the postcondition is to be established with probability (at least) one.

For example the program

$$\textit{faultyFlip} \quad \widehat{=} \quad (x:=0 \ {}_{1/3}\oplus \ x:=1) \ \ [\!] \ \ (x:=0 \ {}_{2/3}\oplus \ x:=1) \ , \tag{3}$$

represents the program that flips for the value of x with a probability that varies between the specified bounds, so that x is set to 0 with probability anywhere in the range $[1/3, 2/3]$. Thus we can regard *faultyFlip* as modelling a coin which does not behave like one which can exhibit an exact distribution of 0's and 1's (a feat which in any case is impossible to achieve in practice), but rather more realistically one which can approximate a probability distribution within error bounds. As suggested above, and from application of the definitions at Fig. 2, we have that $\underline{wp}.\textit{faultyFlip}.[x=0]$ is $1/3$, since all probabilistic transitions give a probability that x is set to zero of *at least* $1/3$ (even the right-most transition at (3)).

In some cases however we are interested in bounding the probabilistic properties from above, and for that we need to interpret the nondeterminism as maximum. Once we do that, refinement —corresponding to a reduction in the range of nondeterminism— means that upper bounds decrease.[3] The next definition supplies the details.

Definition 1. *The* greatest possible *expected value of A on execution of Prog is given by $\overline{wp}.\mathsf{Prog}.A$, where $\overline{wp}$ interprets all nondeterminism angelically: definitions in Fig. 2 remain the same except for nondeterminism which becomes*

$$\overline{wp}.(r [\!] r').A \quad \widehat{=} \quad \overline{wp}.r.A \sqcup \overline{wp}.r'.A \ .$$

[3] This raises the question of whether "flipping" of nondeterminism from minimum to maximum should make us flip our fixed points from least- to greatest as well; we can do either, depending on how we want to interpret the performance metric in the case of non-termination. However when termination occurs with probability one (actually a slightly stronger condition [10, Sec. 2.11.1]) the fixed-points are the same, and that is the case here.

Angelic refinement decreases $\overline{wp}$*-properties.*

$$r \sqsubseteq r' \quad \textit{iff} \quad (\forall A : \mathcal{E}S \cdot \overline{wp}.r.A \Lleftarrow \overline{wp}.r'.A) \ .$$

To see Def. 1 in action we can consider the upper bound on the probability that *faultyFlip* can establish $[x = 0]$.

$$\begin{array}{lll} & \overline{wp}.\textit{faultyFlip}.[x = 0] & \\ = & (x\!:=\!0 \;{}_{1/3}\!\oplus\; x\!:=\!1) \;[\!]\; (x\!:=\!0 \;{}_{2/3}\!\oplus\; x\!:=\!1).[x = 0] & (3) \\ = & \overline{wp}.(x\!:=\!0 \;{}_{1/3}\!\oplus\; x\!:=\!1).[x = 0] \sqcup \overline{wp}(x\!:=\!0 \;{}_{2/3}\!\oplus\; x\!:=\!1).[x = 0] & \text{Def. 1} \\ = & 1/3 \sqcup 2/3 & \\ = & 2/3 \ . & \end{array}$$

We write

$$\{\!\mid \textit{PreE} \mid\!\} \ \mathsf{Prog} \ \{\!\mid \textit{PostE} \mid\!\} \ , \tag{4}$$

to mean that $\textit{PreE} \Lleftarrow \overline{wp}.\mathsf{Prog}.\textit{PostE}$, or "*PreE* is an upper bound on the greatest possible expected value of *PostE* after executing Prog".

As we shall see in the next section, for performance-style properties we are more interested in upper bounds.

3 Estimating Performance-Style Properties

The use of probability in many distributed algorithms and protocols is only to guarantee termination [5,14] — in these cases a proof of termination can often boil down to the behaviour of a finite-state probabilistic process, and techniques for proving termination *with probability 1* are explored in detail elsewhere [10]. The idea is to combine the notion of a standard program variant with probability theory, so that now a *termination variant* may either increase or decrease within some finite range of values provided that there is always some fixed probability with which it is guaranteed to decrease.

We summarise the main steps in a probabilistic proof rule [10, p.191]. Let V be an integer-valued expression in the program variables. Suppose further that

1. there are some fixed integer constants L (low) and H (high) such that $L \leq V \leq H$ is an invariant of the loop, and
2. for some fixed probability $\epsilon > 0$, and for all integers N we have

$$\epsilon[G \wedge (V = N)] \quad \Rrightarrow \quad \underline{wp}.\textit{body}.[V < N] \ .$$

Then termination is certain everywhere.

Next we study the *expected time to termination*, and how to reason about it in a Hoare-style framework.

We begin with the simple case of *faultyFlip* inside a loop

$$\begin{array}{l} \textit{faultyLoop} \quad \widehat{=} \\ \quad \textbf{do} \;\; x = 1 \rightarrow \; (x\!:=\!0 \;{}_{1/3}\!\oplus\; x\!:=\!1) \;\; [\!] \;\; (x\!:=\!0 \;{}_{2/3}\!\oplus\; x\!:=\!1) \;\textbf{od} \ , \end{array}$$

and consider how to compute the expected number of times the loop body must iterate until x is set to 0. Using our definitions we see that, if we add a fresh variable n which is updated at the end of every iteration, so that

$$\begin{array}{lll} \mathit{faultyLoop}_n & \widehat{=} & \textbf{do}\ \ x=1\ \rightarrow \\ & & \quad (x:=0\ {}_{1/3}\oplus\ x:=1) [\!] (x:=0\ {}_{2/3}\oplus\ x:=1); \\ & & \quad n:=n+1 \\ & & \ \textbf{od}\ , \end{array}$$

we may compute the least expected number of iterations by evaluating

$$\underline{wp}.(n:=0; \mathit{faultyLoop}_n).n\ .$$

Here n, as a postcondition, is simply the expectation which returns the value of n in its current state. However, if we now imagine that *faultyLoop* is used to guarantee termination in a distributed protocol, we would be more interested in the greatest expected number of iterations.

Definition 2. *The* greatest expected time *to termination of a loop with terminating body*

$$\mathit{loop} \quad \widehat{=} \quad \textbf{\textit{do}}\ \ B\ \rightarrow \mathsf{Prog}\ \textbf{\textit{od}}\ ,$$

is given by

$$\mathcal{T}(\mathit{loop}) \quad \widehat{=} \quad \lim_{N \geq 1} (\overline{wp}.\mathit{loop}_N.n)\ ,$$

where

$$\mathit{loop}_N \quad \widehat{=} \quad \textbf{\textit{do}}\ \ (B \wedge n < N)\ \rightarrow \mathsf{Prog}; n:=n+1\ \textbf{\textit{od}}\ .$$

In fact Def. 2 computes the longest expected execution path until termination.[4]

Combining the above results reveals a rule for proving upper bounds on worst-case expected performance of programs.

Lemma 1. *Let loop be defined by*

$$\mathit{loop} \quad \widehat{=} \quad \textbf{\textit{do}}\ \ B\ \rightarrow \mathsf{Prog}\ \textbf{\textit{od}}\ .$$

If E is an expectation such that

$$\{\!|\ [B] \times (E{-}1)\ |\!\}\ \ \mathsf{Prog}\ \ \{\!|\ E\ |\!\},^{5} \tag{5}$$

then $\mathcal{T}(\mathit{loop})$ is bounded above by E. We call such an expectation a bounding variant*.*

[4] The reason we take an explicit limit is to avoid arithmetic with ∞ in Fig. 1's definition of loop semantics.

[5] Often in proving properties of a loop body it's convenient to assume truth of some predicate whose invariance is proved separately via standard (*i.e.* non-probabilistic) $\underline{wp}$ [10, Lem. 1.7.1]. That technique applies here also.

Proof. We show that $\overline{wp}.loop_N \Rrightarrow E + n$, *and the result then follows from Def. 2 (where as before* n *is a fresh variable, so that* E *is independent of it). First we show that* $[\neg B \vee n = N] \times n + [B \wedge n < N] \times \overline{wp}.\mathsf{Prog}.(E + n) \Rrightarrow E + n$, *as follows:*

$$
\begin{array}{ll}
 & [\neg B \vee n = N] \times n + [B \wedge n < N] \times \overline{wp}.\mathsf{Prog}; (n := n + 1).(E + n) \\
\equiv & [\neg B \vee n = N] \times n + [B \wedge n < N] \times \overline{wp}.\mathsf{Prog}.(E + n + 1) \\
\Rrightarrow & [\neg B \vee n = N] \times n + [B \wedge n < N] \times (E + n) \qquad \textit{(5), (4), arithmetic} \\
\Rrightarrow & E + n\ . \qquad 0 \Rrightarrow E
\end{array}
$$

Appealing now to the least fixed point property, we see that $\overline{wp}.loop_N \Rrightarrow E + n$, *as required.*

To see Lem. 1 in action, we consider the expected number of iterations of *faultyLoop* above. We note that

$$\{\!|\ 2[x = 1]\ |\!\} \quad \mathit{faultyFlip} \quad \{\!|\ 3[x = 1]\ |\!\}\ , \tag{6}$$

since

$$
\begin{array}{ll}
 & \overline{wp}.\mathit{faultyFlip}.(3 * [x = 1]) \\
\equiv & 3 * \overline{wp}.\mathit{faultyFlip}.[x = 1] \qquad \overline{wp}\ \text{distributes scalars} \\
\equiv & 3 * 2/3 \\
\equiv & 2\ .
\end{array}
$$

Thus we are able to deduce that, for any execution of the nondeterminism in *faultyLoop*, it must terminate after performing on average no more than 3 iterations.

In this section we have illustrated some general results for deducing the expected-performance-style properties of programs. Our approach is to analyse an *abstraction* of the program, then using program refinement to associate the results with a refinement.

Whether we use the program logic for demonic $\sqsubseteq$ or angelic $\sqsubset$ refinement depends on whether we are concerned with correctness (demonic: postcondition established with probability at least some p) or performance (angelic: expected iterations is at most some N). In either case, since refinement preserves program properties we see that if we prove termination with probability one of the abstraction, then any demonic refinement will also terminate with probability one. Similarly any upper bound or worst case behaviour of the abstraction is also an upper bound or worst case behaviour of any angelic refinement.

Note that "removing $[\!]$" achieves both forms of refinement simultaneously, as one would expect: our separation of the two is so that we do not have to calculate both if in fact we're interested in only one of them.

4 Case Study: Self-stabilisation Algorithms

We illustrate the above techniques on two case studies. The first is a leadership-election protocol, in which the stable states are those where exactly one of N

participants is the leader, allowed by convention then to take certain actions on behalf of the group; an unstable state is one where there is no leader or several (aspiring) leaders, perhaps due to hardware failure; and the stabilisation algorithm is to bring about the exactly-one-leader situation again. We analyse the expected time for the election to complete.

The second case study is a general network in which tokens circulate (an abstraction of many distributed algorithms); unstable states are those in which there are several tokens; stable states are those in which there is exactly one.

The difference between the two studies is that in the first, the communication pattern is regular (all-to-all) and the unstable state is presumed to be detected somehow, leading to the initiation of the stabilisation protocol. In the second, the network and communication patterns are so general that we can only hope to establish termination (and not expected time to it), and the stabilisation algorithm is running continuously, without any need to detect unstable states.

4.1 A Leadership-Election Protocol

Our first example is a leadership-election protocol [1, Sec. 8.5.4] for a totally connected network of processes; we show that its expected number of rounds to stabilisation is constant.[6]

We first give an informal description of the protocol.

Informal description and formalisation. A number N of processes are to elect a single leader. On each round, each process chooses a number k for itself, uniformly from $1..N$, and sends its choice to all other processes. Each process then separately acts as follows:

- If no process chose 1, then it enters a new round.
- If some processes chose 1, but it did not, then it drops out.
- If it and possibly other processes chose 1, it enters a new round.

The election is finished when only one process remains. We formalise the protocol in Fig. 3; more detailed descriptions would be angelic refinements of this one.

Rapid termination. We note first that the protocol of Fig. 3 terminates exponentially fast, that is the chance of its taking more than some number of steps M is exponentially small in M.

[6] The earlier example of cricket can illustrate rounds, and expected time to termination. The normal protocol is *not* symmetric, because one team flips and the other calls. But the time to termination is exactly one flip.

A truly symmetric protocol would employ three coins, and both teams and the referee would flip all at once: the winning team would be the first to flip a face different from the other two. This protocol has constant *expected* time to termination of two rounds of three simultaneous flips each, assuming the coins are fair. (It still works if the players' coins are unfair —one never knows— but then it could take longer.)

```
1    n := N
2    do n > 1 →
3        n' :∈ {k: 0..n @ (n choose k) × (n−1)^(n−k)/n^n}
4        if n' ≠ 0 then n := n' fi
     od
```

$$
\begin{array}{ll}
1 & n := N \\
2 & \mathbf{do}\ n > 1 \ \rightarrow \\
3 & \quad n' :\in \{k\colon 0..n \ @ \ \binom{n}{k} \times (n{-}1)^{n-k}/n^n\} \\
4 & \quad \mathbf{if}\ n' \neq 0 \ \ \mathbf{then}\ n := n' \ \ \mathbf{fi} \\
 & \mathbf{od}
\end{array}
$$

1— Initially all N processes participate; subsequently n is the number (still) participating at any point, and n decreases over time as processes drop out.

2— Termination occurs when only one process remains, and it becomes the leader. (Note that $1 \leq n \leq N$ is an invariant.)

3— Here with operator $:\in$ we choose n' from a distribution, indicated by a set-like comprehension (bound variable k) but containing an @ for "with probability" (instead of a | for "such that"), in which the probability of there having been being k processes that chose 1 (out of $1..n$) is explicitly given.

4— If no processes chose 1, then they all go on to the next round (and n is not changed); if at least one processor chose 1, then it and any others similarly go on to the next round. (Note therefore that they all go 'round again in *two* cases: all chose 1, or none did.)

Fig. 3. Leadership election protocol

Sufficient for that is a bounded-away-from-zero probability of termination on any single iteration. That is trivial, by inspection, as the probability of setting n' to 1 is just

$$\binom{n}{1} \times (n{-}1)^{n-1}/n^n \quad = \quad (n{-}1/n)^{n-1}$$

which, being anti-monotonic in n, is bounded below by $(N{-}1/N)^{(N-1)}$ no matter what value n has as the loop continues to execute.

Expected iterations. We now show that the expected number of iterations is constant. We assume that constant to be some E, and by a schematic proof find suitable conditions for it. Since termination occurs in zero steps when $n = 1$, we choose our bounding variant to be

$$E \times [n > 1] \ ,$$

and from Lem. 1 we must show that $n > 1$ (the guard) implies

$$E \times [n > 1] \ - \ 1 \quad \Lleftarrow \quad \overline{wp}\,.\,\text{“3;4 in Fig. 3”}\,.\,(E \times [n > 1]) \ .$$

Here is the calculation:

$$
\begin{array}{lll}
 & E \times [n-1] & \\
\cdot \equiv & \textbf{if } n' \neq 0 \textbf{ then } E \times [n' > 1] \textbf{ else } E \times [n > 1] \textbf{ fi} & \text{applying } \overline{wp}.(4) \\
\cdot \equiv & \quad (\sum k\colon 1..n \cdot E \times [k > 1] \times \binom{n}{k} \times (n{-}1)^{n-k}/n^n) & \text{applying } \overline{wp}.(3) \\
 & + E \times [n > 1] \times \binom{n}{0} \times (n{-}1)^n/n^n & \\
\equiv & \quad E \times (\sum k\colon 2..n \cdot \binom{n}{k} \times (n{-}1)^{n-k}/n^n) & \text{arithmetic; assumption } n > 1 \\
 & + E \times \binom{n}{0} \times (n{-}1)^n/n^n & \\
\equiv & \quad E \times (1 - ((n{-}1)/n)^n - ((n{-}1)/n)^{n-1} & \text{arithmetic} \\
 & + E \times ((n{-}1)/n)^n & \\
\equiv & E \times (1 - ((n{-}1)/n)^{n-1}) & \text{arithmetic} \\
\Rrightarrow & E \times [n > 1] \ - \ 1 & \text{assume } n > 1 \text{ and } 1 \leq E \times ((n{-}1)/n)^{n-1}
\end{array}
$$

Our assumption, rearranged, is that for $n \geq 2$ we have

$$(n{-}1/n)^{n-1} \quad \geq \quad 1/E \ ,$$

a property that holds for the "real" $e = 2.718 \cdots$

Thus we have shown that the protocol terminates in expected constant time no more than e, that is just under 3, rounds. If the rounds themselves cost time N each (for the exchange of N messages), then the expected time complexity of stabilisation is no more than $3N$.

A more severe abstraction. There is however an alternative approach, in which our initial description of the algorithm is "more severely abstracted" — we note merely whether $n = 1$ or not. Letting Boolean b record that abstraction, our algorithm is transformed into the one shown in Fig. 4.

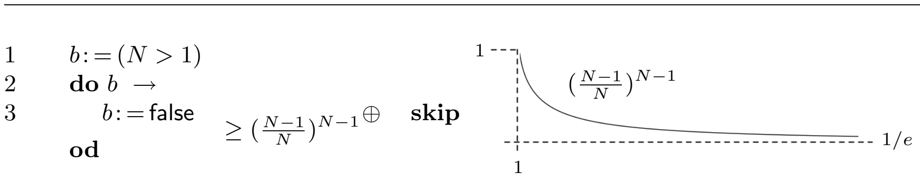


Fig. 4. Leadership election protocol, more severely abstracted

We justify the abstraction by noting that the only command that sets n to 1 in the original $(3; 4)$ does so with probability

$$\binom{n}{k} \times (n{-}1)^{n-k}/n^n \quad = \quad \left(\frac{n{-}1}{n}\right)^{n-1} \quad \text{when } k = 1,$$

that we know $n \leq N$, and that the expression shown is anti-monotonic in n (tending to $1/e$ from above), so that the $n = N$ case —as appears in the abstracted algorithm— is indeed the most pessimistic value.

The expected number of rounds here is then no more than the inverse of that probability, which tends to e from below as N increases without bound.

But is this easier, really? The work to prove the soundness of the abstraction Fig. 4 is probably the same as required to do the earlier calculations anyway.

4.2 A Token-Graph Stabilisation Algorithm

As a second example we treat a more general situation whose exact behaviour is quite complex but for which, nevertheless, proof of termination is still possible using the techniques we have explained.

Informal description. There is a strongly connected directed graph with N nodes; each node is either full (contains a token) or empty (doesn't). An adversarial scheduler (but fair — see below) repeatedly selects some single node to take a step:

- If the node is empty, nothing happens.
- If the node is full then it chooses between keeping its token or passing it one step along an outgoing edge. The choice is made probabilistically, with a fixed non-zero lower bound applied to each alternative (including keeping). (Note that if the lower bounds sum to less than one, the node can itself act demonically — thus we have demonic choice potentially in both the scheduling and in the nodes' actions.)

Any node receiving a token becomes full (but never "over-full" — multiple tokens reduce to one).

The adversarial scheduler. We allow the scheduler to choose nodes demonically, except for the following fairness constraint. Say that a node's *priority* is the number of steps since it was last scheduled: we require that for some fixed constant (trigger) T the scheduler must schedule nodes of priority at least T before any of priority lower than T.

This is a realistic policy (could easily be implemented), and if T is large it allows the scheduler a great deal of choice.

With suitable $T \geq N > 1$ the policy maintains the invariant (*I1*) that all priorities are no more than (a maximum) $M = T{+}N$, which in turn gives an easy variant to show that no node is forever overlooked. To prove the invariant we need a subsidiary invariant (*I2*), that

> if there is a node with priority p satisfying $T \leq p \leq M$, then there are at least $p{-}T$ nodes with priority below $p{-}N$.

Truth of *I1* is immediate from *I2* and the fact that there are only N nodes: assume for a contradiction that some "high" node has priority p more than M;

then from *I2* there would be more than $M - T = (T + N) - T = N$ nodes with priority below $p{-}N$. Since there are only N nodes overall, that is a contradiction.

Preservation of *I2* is argued as follows. Suppose a step has just been taken, and consider all nodes in turn, just after a step has been taken: all nodes will have "new" priority one more than their "old" priority, except for the one scheduled, whose new priority will be 0.

If a node's new priority p satisfies $p < T$, then *I2* is true trivially (false antecedent); if it satisfies $p = T$ then *I2* is again trivial (there are at least zero nodes satisfying anything).

In the remaining case where the new priority p satisfies $T < p \leq M$ then —because $p{-}T$ has increased by one— we must show there to be one more node prioritised below $p{-}N$ after the step than there were below $(p{-}1) - N$ before the step. Since all not-scheduled nodes below $(p{-}1) - N$ before are (still) below $p{-}N$ now, and also the just-scheduled node is below $p{-}N$ now with its new priority 0 satisfying $0 < p{-}N$ (because $p > T \geq N$), we need only show that the just-scheduled node was *not* below $(p{-}1) - N$ before.

Suppose the just-scheduled node had priority p' before. Since $T < p$ now, we know that $T \leq p'$, since otherwise by the policy p' would not have been scheduled instead of p. We reason

	$p' \not< (p{-}1) - N$	
iff	$p' \geq (p{-}1) - N$	
if	$T \geq (p{-}1) - N$	$p' \geq T$
if	$T \geq (M{-}1) - N$	$p \leq M$
iff	$T \geq (T{+}N{-}1) - N$	$M = T{+}N$
iff	$T \geq T{-}1$,	

which concludes the argument for maintaining *I2*.

Formalisation of the protocol. Say that a *full cover* of the nodes is a directed path in which all full nodes appear; its *size* is the number of nodes in it (including of course any empty ones along the way). A *minimal* full cover (*MFC*) is a full cover of minimum size; and the *minimum cover size* (*MCS*) is the size of a minimal full cover.

We say that a node is a *straggler* if it is the trailing node of some *MFC*. The importance of stragglers is that, if scheduled, they will with non-zero probability decrease the *MCS* by choosing to move one edge along the *MFC* they are at the end of.

Let *NId* be a set of (unique) node identifiers, so that $\#NId = N$. Function $pr\colon NId \to \mathbb{N}$ gives the priority of each node, zeroed whenever it is scheduled and incremented otherwise.

The set $sgs\colon \mathbb{P}NId$ contains the *NId*'s of the stragglers. Natural number $mcs\colon \mathbb{N}$ is the minimum cover size; note that if $mcs = 1$ then there is only one full node, and the algorithm should terminate.

In Fig. 5 is a program giving the behaviour of these variables; more detailed descriptions would be demonic refinements of this one. It turns out, surprisingly, that we do not have to keep track of which nodes are full: information about *sgs* is enough.

```
     pr := 0;                          // All nodes' priorities initially zero.
     do mcs > 1 →
        // — Book-keeping of priorities; selection of token; fairness constraint. —
1       pr := pr + 1;                  // Increment all priorities...
        n :∈ NId;                      // ...but then choose one node...
        pr.n := 0;                     // ...and schedule it.
        [pr ≤ M];                      // Require for fairness that no priority is too large.

        // — Movement of selected token: straggler, or not? —
        if n ∈ sgs →                   // If the scheduled node is a straggler...
2              ( mcs :<₁ mcs           // ...then it might decrease mcs...
            ≥p⊕                        // ...but if it moves the wrong way...
3                mcs :≤₁ N );          // ...then anything goes.
4           sgs :⊆₁ NId                // Either way, the stragglers can change.

        [] n ∉ sgs →                   // If it's not a straggler...
5              ( skip                  // ...then it might stay where it is...
            ≥p⊕                        // ...but, if not, again...
                 mcs :≤₁ N;            // ...anything goes...
                 sgs :⊆₁ NId )         // ...and stragglers can change if it moved.
         fi
     od
```

Assignments and tests to *pr* as a whole operate pointwise: thus $(pr + 1)$ increments all priorities, and $(pr \leq)$ bounds all priorities.

The "coercion" $[pr \leq M]$ acts as a miracle (in theory causing backtracking) if its predicate is false, having the effect thus of forcing earlier nondeterminism —if possible— never to *make* it false. The nondeterminism in this case is the selection $n{:}{\in}NId$ of the node to schedule, and our earlier argument establishing *I1* simply shows that there are non-backtracking implementations of the nondeterminism which make the scheduling feasible.

The assignments $:<_1$, $:\leq_1$and $:\subseteq_1$ are nondeterministic choices according to the relation given, but requiring that the result be at least 1, or non-empty, as appropriate.

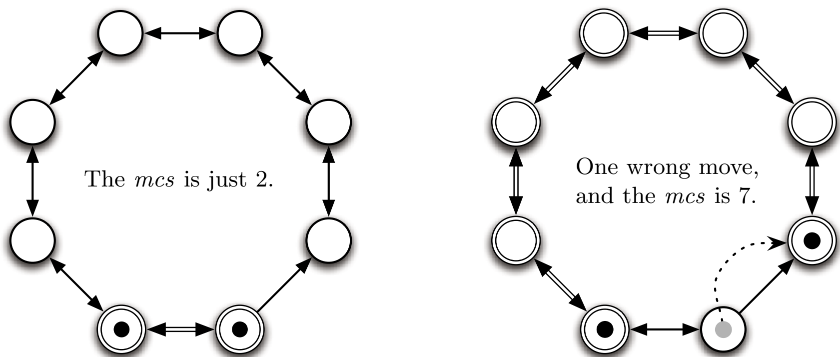
Note that in the $n \in sgs$ alternative the assignment to *sgs* occurs unconditionally; in the $n \notin sgs$ alternative, it occurs only with probability $< p$.

Fig. 5. Stabilisation of token network: abstraction

Termination of the algorithm. Define *V0* to be the maximum over all $n \in sgs$ of $pr.n$; this cannot exceed M. Define *V1* to be *mcs*. Then the termination variant overall is lexicographic, with *V0* ascending and *V1* descending:

$$\begin{array}{lcl} V0 & \widehat{=} & (\sqcap n\colon sgs \cdot pr.n) \\ V1 & \widehat{=} & mcs \end{array}$$

$$V \;\widehat{=}\; V1 \times (M{+}1) - V0\,.$$



The directed network is connected as shown; note the uni-directional arc at bottom right. Nodes in a minimal cover are shown double-bordered, and the cover's arcs are double-arrowed.

On the left, the *mcs* is just two, and the algorithm is "near" termination: both full nodes are min-stragglers. We suppose the right-hand min-straggler is selected but —unfortunately— the probabilistic choice ${}_{\geq p}\oplus$ goes against us, and Statement 3 is executed for that node.

As a result the small minimal full cover is replaced by a very large one, and the variant V has *increased* substantially, by approximately $5M$ (where M, recall, is the fairness parameter for scheduling).

The virtue of the probabilistic variant is that these complex situations *do not matter* for termination —they can be ignored— as long as their probability of occurrence is bounded away from one.

Fig. 6. A straggler moves "the wrong way"

This variant is bounded below by zero because *V1* is at least 1 (loop guard) and *V0* never exceeds M (invariant *I1*, enforced by the coercion).

Thus for termination with probability one it is sufficient to show that on each iteration V strictly decreases with non-zero probability. Informally we argue that there are two cases:

- A straggler is scheduled, in which case with probability at least p the sub-variant *V1* decreases by at least one (Statement 2). Sub-variant $-$*V0* can increase (Statement 4), but not by more than M. Hence overall V decreases by at least 1.
- or a non-straggler is scheduled, in which case with probability at least p sub-variant *V1* is unchanged (Statement 5), but sub-variant $-$*V0* has (already) decreased by 1 (Statement 1). Again, V must decrease with non-zero probability.

Those two cases are sufficient to establish termination with probability one.

Illustration of the complexity avoided. Once the termination variant is found, the termination argument (as usual) is very straightforward. Recall however that we are illustrating novel *probabilistic* variant techniques, and that the control of complexity they provide was "designed in" by analogy with their standard versions, and we are taking advantgage of it.

Consider for example the case where a straggler is scheduled but (with probability $< p$) it moves "the wrong way" (Statement 3) and does not act to reduce the minimum cover size: this situation is illustrated in Fig. 6 for a simple directed ring topology. Although the variant can increase enormously (by approximately $5M$ in the figure), the probabilistic-variant technique ensures that those situations need not be analysed if probability-one termination is all that is required.

5 Conclusions

We have illustrated how the expectation-transformer approach to verification can be used to calculate both correctness and performance-style properties of probabilistic programs by reasoning at the source-code level. The fact that refinement is an integral part of the expectation transformers means that we may transfer proved properties of the abstraction to any refinement, a feature which separates us from other approaches to program verification, such as model checking [13,6,8]. This effectively allows us to use "lightweight" methods, leaving the bulk of the formality to a proof of refinement, and techniques for expediting that are addressed elsewhere [10], some of which have mechanised support [7].

In standard program semantics the use of a program variant is sufficient to supply both an upper bound on performance (number of iteration of a loop) as well as termination. In the probabilistic systems, it appears at first that the two must be separated — but in fact the bounding variant is the more general [2], although the termination variant is rather easier to use.

Other systems using refinement for performance include Hallerstede *et al.* [4] and Sere *et al.* [15].

Acknowledgements

We thank the reviewers for their helpful comments.

References

1. G. Brassard and P. Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996.
2. O. Celiku and A. McIver. Compositional specification and analysis of cost-based properties in probabilistic programs. In *Proceedings of Formal Methods 2005*, number 3582 in LNCS, 2005.
3. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1976.
4. S. Hallerstede and M. Butler. Performance analysis of probabilistic action systems. *Formal Aspects of Computing*, 16(4):313–331, 2004.

5. T. Herman. Probabilistic self-stabilization. *Inf. Proc. Lett.*, 35(2):63–7, 1990.
6. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
7. TS Hoang, Z Jin, K Robinson, A McIver, and C Morgan. Development via refinement in probabilistic B. In *Proc. of ZB 2005*, number 3455 in LNCS, 2005.
8. J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov-chain model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 347–362. Springer Verlag, 2000.
9. A.K. McIver, C.C. Morgan, J.W. Sanders, and K. Seidel. Probabilistic Systems Group: Collected reports. `web.comlab.ox.ac.uk/oucl/research/areas/probs`.
10. Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Technical Monographs in Computer Science. Springer Verlag, New York, 2004.
11. C.C. Morgan and A.K. McIver. *pGCL*: Formal reasoning for random algorithms. *South African Computer Journal*, 22, March 1999. Available at [9, key pGCL].
12. C.C. Morgan, A.K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–53, May 1996. `doi.acm.org/10.1145/229542.229547`.
13. PRISM. Probabilistic symbolic model checker. `www.cs.bham.ac.uk/~dxp/prism`.
14. M.O. Rabin. The choice-coordination problem. *Acta Informatica*, 17(2):121–34, June 1982.
15. Kaisa Sere and Elena Troubitsyna. Probabilities in action systems. In *Proc. of the 8th Nordic Workshop on Programming Theory*, 1996.
16. David Stirzaker. *Elementary Probability*. Cambridge University Press, 1994.