Abstraction and refinement in probabilistic systems

Annabelle McIver[1] Carroll Morgan[2]

Abstract

We summarise a verification method for probabilistic systems that is based on abstraction and refinement, and extends traditional assertional styles of verification.

The approach makes extensive use of the *expectation transformers* of *pGCL* [17, 16, 13], a compact probabilistic programming language with an associated logic of real-valued functions. Analysis of large systems is made tractable by abstraction which, together with algebraic and logical reasoning, results in strong and general guarantees about probabilistic-system properties.

Although our examples are specific (to *pGCL*), our overall goal in this note is to advocate the hierarchical development of probabilistic programs via levels of abstraction, connected by refinement, and to illustrate the proof obligations incurred by such an approach.

1 Introduction

The verification of large hybrid systems presents a challange for formal methods, a challenge which is especially acute when those systems operate within a context of uncertainty and randomisation. In such contexts randomisation must be incorporated in the analysis — in particular, system properties are no longer satisfied absolutely, but can only be guaranteed up to some probability that, in turn, impacts on performance and reliability. In addition the analysis is complicated by the interaction of probability with other system features, substantially increasing the verification's overhead.

Standard logics and methods do not accommodate probability; yet decades of experience there has shown that rigorous techniques for controlling system complexity rely on *abstraction* and *refinement*. Simple properties are extracted directly from the details of the system and are then themselves used to prove "second-order properties", such as "with probability $1/3$ eventually the system will terminate".

The crucial point is that the main business of verification can take place without reference to the real system (and all its intricacies) provided a body of simple properties has been extracted beforehand; program *refinement* provides the formal "glue" linking the derived properties to the original system. Indeed these two features together provide what we call the "intellectual advantage" of logical approaches generally.[1]

A wide variety of logics have been developed as possible bases for verifying randomised systems [5, 3]; the *expectation transformer* approach [9, 17, 13] in particular integrates traditional assertional-styles of program verification with probability. It is based on a logic of real-valued functions (called "expectations"), where the real values give access to the explicit probabilities defined in the system. The semantics is designed to work even at the level of program code, and has an in-built notion of program refinement which encourages a prover to move between various levels of abstraction. Moreover the refinement's *compositionality* (see below), by which we mean that the refinement relation is preserved even by contexts, allows the activity of proof to be distributed amongst smaller, more manageable parts of the system, without disturbing the validity of the overall verification. As in any proof-based method, the results provide very strong guarantees of correctness independently of system-specific characteristics such as size; and any assumptions about operating conditions, for example, are naturally documented throughout the proofs.

Typical of refinement based methods —and what distinguishes them from algorithmic methods such as model checking— is the extensive use of *program logics* and *algebras*. Rather than building an explicit model of a particular system and analysing it directly with specialised algorithms, the programs that make up the system are considered to have a common mathematical structure in their own right. These can be described precisely by logical axioms which can be cited at any stage to justify a proof step. The expectation transformers' logic of probabilistic programs combines a weakened form of "expectation logics" found in standard probability theory [18], together with the logic based on Dijkstra's predicate transformers from standard programming theory [2].

Taking expectations as the basis for the logic, rather than a possibly simpler "logic of probabilities" turns out

[1]Computing Department, Macquarie University, Sydney 2109, Australia

[2]Dept. Eng. and Comp. Sci., Univ. New South Wales, Sydney 2052, Australia

[1]The intended analogy is with the "mechanical advantage" of physical systems in which simple well-chosen tools amplify an applied force.

to have theoretical and practical significance. On the theoretical side it is necessary for compositionality of refinement [13], whereas on the practical side having access to general real-valued functions means that performance-like properties such as "average efficiency" can be treated within the proof system, and thus such properties are amenable to abstraction and refinement as well [12].

In Sec. 2 we summarise the main features of the expectation-transformer approach touched on above, and then in Sec. 3 we illustrate them with a simple example which, though small, serves to illustrate how the proof-based techniques can be used for analysing larger, more intricate randomised programs.

The following notational conventions apply. Function application is written $f.x$ rather than $f(x)$, to reduce the clutter of parentheses; thus $F(g)(x)$ becomes just $F.g.x$. We write S for a (fixed) underlying state space, and $\overline{S}$ for the set of *discrete sub-probability distributions* over S, the functions from S to the interval $[0,1]$ which sum to no more than one. Conditional expressions are written *if-part* $\lhd$ *condition* $\rhd$ *else-part*. The set of functions from S to the non-negative reals is called the *expectations*; they are ordered by lifting $\leq$ pointwise; thus we write $A \Rrightarrow B$ (or $B \Lleftarrow A$) if and only if $A.s \leq B.s$ for all s in S. The expression $A \equiv B$ means $A \Rrightarrow B$ and $A \Lleftarrow B$. We write $\underline{c}$ for the expectation returning c for all states. If distribution d is in $\overline{S}$ and A is an expectation then $\int_d A$ denotes the expected value of A over d, effectively $\sum_{s:S} d.s \times A.s$. If *Pred* is a predicate, then $[Pred]$ is the *characteristic function* which takes states satisfying *Pred* to 1, and to 0 otherwise.

2 Expectation transformers: abstraction, refinement and compositionality

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

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

for example, does not guarantee to set variable x to 0 under any (initial) condition — instead the probabilistic choice operator "${}_{1/4}\!\oplus$" describes the behaviour of 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 on any particular occasion.

A formal description of that behaviour —the operational semantics— takes the form of a traditional transition-system-based 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 [7]. 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.) 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 test the final distribution d with respect to the characteristic function $[x = 0]$, since from standard probability theory $\int_d [x = 0]$ is the probability assigned to the event "$x = 0$" by the distribution 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 directly as the basis for deriving properties of any intricacy. This becomes apparent when general program features are included in the programming language such as Boolean choice, nondeterminism, sequential composition and iteration. Better is to use a dual semantics — the expectation transformers— which focusses directly on program properties, rather than on the details of the probabilistic transitions which imply them.

To appreciate the duality we rationalise the above calculation, this time concentrating on properties rather than transitions. First of all, we use expectations rather than predicates to express properties. This immediately allows us to regard programs as *transforming* expectations consistent with their operational semantics. We write $wp.(x{:=}\ 0\ {}_{1/4}\!\oplus\ x{:=}\ 1)$ 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* $\underline{1/4}$; more precisely we say that

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

In general, if Prog is a program, *PostE* a post-expectation, and s an initial state, then $wp.\mathsf{Prog}.[x = 0].s$ 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

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

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

The full definition of *wp*, as a mapping from program texts to expectation transformers, is set out at Fig. 1. We use the small programming language *pGCL* [16], an extension of *GCL* [2] with probabilistic choice. The definitions are almost identical to Dijkstra's original predicate

assignment	$wp.(x{:=}\ E).A \ \widehat{=}\ A[E/x]$,
sequence	$wp.(\mathsf{P};\mathsf{Q}).A \ \widehat{=}\ wp.\mathsf{P}.(wp.\mathsf{Q}.A)$,
probability	$wp.(\mathsf{P}\ {}_p\!\oplus\ \mathsf{Q}).A \ \widehat{=}$ $p * wp.\mathsf{P}.A + (1{-}p) * wp.\mathsf{Q}.A$,
nondeterminism	$wp.(\mathsf{P} \sqcap \mathsf{Q}).A \ \widehat{=}\ wp.\mathsf{P}.A \ \mathbf{min}\ wp.\mathsf{Q}.A$,
Boolean choice	$wp.(\mathbf{if}\ B\ \mathbf{then}\ \mathsf{P}\ \mathbf{else}\ \mathsf{Q}\ \mathbf{fi}\ ).A \ \widehat{=}$ $[B] * wp.\mathsf{P}.A + [\neg B] * wp.\mathsf{Q}.A$,
iteration	$wp.(\mathbf{do}\ B \to r\ \mathbf{od}).A \ \widehat{=}$ $(\mu X \cdot [B] * wp.r.X + [\neg B] * A)$.

A is an expectation, E is an expression over the state variables, $[E/x]$ is syntactic replacement of x by E, and a term $(\mu X \cdot F.X)$ refers to the $\Rrightarrow$-least fixed-point of expectation-to-expectation function F; it is guaranteed to exist, provided that F is monotone, since the expectations form a complete partial order [8].
These definitions are dual to an operational model based on the state-to-distribution semantics [17].

Figure 1: Structural definitions of *wp* for *pGCL*.

transformers; the difference is that we use a domain of expectations based on the $\Rrightarrow$ order, rather than of predicates based on the implication order $\Rightarrow$. This means, conveniently, that the only apparent differences syntactically are that the definitions use arithmetical- rather than Boolean operators. Nondeterministic choice, for example, takes the minimum of its two arguments; and the new operator *probabilistic choice* is parametrised by a real $0 \leq p \leq 1$ and takes the p-weighted average of its arguments.

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 rules of the transformer logic; they play the part of "healthiness conditions" coined by Dijkstra in his original presentation of the predicate transformers. Intuitively they characterise "legal computations" — mathematically they define the common rules satisfied exactly by *wp*−images of programs [17]. 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.

Program refinement is defined by property preservation: a program Prog is refined by another Prog', or $\mathsf{Prog} \sqsubseteq \mathsf{Prog}'$, precisely when all properties of interest of Prog are inherited by Prog'. In our case it means that we have

$$\mathsf{Prog} \sqsubseteq \mathsf{Prog}' \quad \textit{iff} \quad wp.\mathsf{Prog}.A \Rrightarrow wp.\mathsf{Prog}'.A \ , \qquad (3)$$
$$\text{for any expectation } A.$$

Thus any correct annotation of Prog is also a correct annotation of Prog'.

sub-additivity	$wp.\mathsf{Prog}.(A+B) \ \Lleftarrow\ wp.\mathsf{Prog}.A + wp.\mathsf{Prog}.B$
scaling	$wp.\mathsf{Prog}.(k*A) \ \equiv\ k * wp.\mathsf{Prog}.A$
constants	$wp.\mathsf{Prog}.(A \ominus \underline{k}) \ \Lleftarrow\ wp.\mathsf{Prog}.A \ominus \underline{k}$

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

$$A \ominus \underline{k} \ \ \widehat{=}\ \ (A - \underline{k})\ \mathbf{max}\ \underline{0} \ .$$

Figure 2: Axioms of the expectation transformer logic [17].

For us, compositionality is critical in the expectation-transformer approach, where we use it to replace more detailed implementations with "abstractions" in the same program contexts. A program Prog as in (3) can be thought of as a *specification*, and Prog' its *implementation*. In operational terms Prog' exhibits less nondeterminism than Prog; on the other hand Prog is more abstract, less detailed, and has fewer properties, and consequently is much easier to reason about when used to replace Prog' in a context — and the important point here is that the reasoning can be carried out *without explicit reference to* Prog' at all. Thus if $\mathsf{Prog} \sqsubseteq \mathsf{Prog}'$ then compositionality implies, for example, that

$$\mathbf{do}\ B \to \mathsf{Prog}\ \mathbf{od} \quad \sqsubseteq \quad \mathbf{do}\ B \to \mathsf{Prog}'\ \mathbf{od}$$

as well. Compositionality is the contract that binds the results of that simpler analysis of the left-hand side to the right-hand side, even though the right-hand loop did not participate actively in the verification at all.

3 Generating a uniform choice

In this section we give a small example of a probabilistic program developed in two stages linked by abstraction and refinement.

For practical purposes a source of randomness is generally only available as a stream of unbiased random bits; however many applications' correctness relies on more elaborate distributions. Those distributions can be generated by using various sampling methods, such as the following in which a small program uses unbiased bits to generate a uniform choice over a positive number N of alternatives.

3.1 Overall strategy

The complete program is given in Fig. 3, together with its pre- and post-expectations that, in the style of (2), assert it to select variable k uniformly from the range $0 \leq k < N$. The goal of the verification is to show that the given expectations form a correct annotation.

Rather than proving the program's properties all at once, in a hierarchical approach instead we first establish relevant properties of the inner loop (Fig. 4) on its own —

```
{ K,N∈ℕ ∧ N>0 }
var k,n: ℕ;

{ [0≤K<N]/N }
k:= N;
do k ≥ N →
  k,n:= 0,N−1;
  do n ≠ 0 →
    k:= 2k 1/2⊕ k:= 2k+1;
    n:= n ÷ 2
  od
od
{ [k=K] }
```

The inner loop selects k uniformly such that $0 \leq k < \$N$, where $\$N$ is the least power-of-two no less than N. The outer loop accepts that choice only if $k < N$; otherwise the inner loop is repeated. The effect overall is to select k uniformly so that $0 \leq k < N$.
The pre- and post-expectation annotations express that for any K the probability of achieving $k = K$ on termination is at least $1/N$ if $0 \leq K < N$ (and at least zero otherwise).[2]

Figure 3: Uniform-selection algorithm: entire

and then we use those properties, as an abstraction of the inner loop, to reason about the outer loop without (any longer) having to refer to the inner loop's code.

The strategy is sound because refinement (used here to link the abstract properties of the inner loop with the actual loop itself) is compositional. We will use the general progamming logic, set out at Fig. 2, on the abstraction in order to complete the full proof.

3.2 Proofs for inner loop

The complete annotations for the inner loop are given in Fig. 5. They are validated using the definitions set out in Fig. 1, together with special-purpose "invariant-style" rules for probabilistic loops, derived from Figs. 1,2 [15].

The rules are set out at Fig. 6 and, as for the classical loop rules from assertional programming, they rely on finding a "loop invariant" summarising a formal relation between program variables and probabilities which remains invariant under execution of the loop body.

[2] We write "at least" because the programming logic is designed to give lower-bound estimates. In this case however, since there are only N possible values for k, the probability $1/N$ is exact. That reasoning is easily formalised using the algebraic techniques of Sec. 4; we omit it to save space.

```
{ [0≤K<$N]/$N }
k,n:= 0,N−1;
do n ≠ 0 →
  k:= 2k 1/2⊕ k:= 2k+1;
  n:= n ÷ 2
od
{ [k=K] }
```

Figure 4: Uniform-selection algorithm: inner loop only

```
{ [0≤K<$N]/$N }
k,n:= 0,N−1;
{ [k($̄n)≤K<(k+1)($̄n)]/$̄n }
do n ≠ 0 →
  { [k($̄n)≤K<(k+1)($̄n)]/$̄n }
  k:= 2k 1/2⊕ 2k+1;
  { [k$̄(n÷2)≤K<(k+1)$̄(n÷2)]/$̄(n÷2) }
  n:= n ÷ 2
  { [k($̄n)≤K<(k+1)($̄n)]/$̄n }
od
{ [k=K] }
```

We write $\overline{\$}n$ for $\$(n{+}1)$.

Figure 5: Inner loop with annotations

Let Loop be an initialised loop Init; **do** $B \to$ Body **od**. If the three conditions

(Sec. 3.3)	*on initialisation:*	$PreE \;\Rrightarrow\; wp.\mathsf{Init}.I$
(Secs. 3.4,5.1)	*while iterating:*	$[B] * I \;\Rrightarrow\; wp.\mathsf{Body}.I$
(Sec. 3.5)	*on termination:*	$[\neg B] * I \;\Rrightarrow\; PostE$

all hold, then $\{I\}$ Loop $\{PostE\}$ is a correct annotation provided the loop terminates.[3]

Figure 6: The probabilistic loop rule

In the proofs below we right-justify the justification for each step. In the special case that we are reasoning from a post-expectation "backwards" towards a pre-expectation, and are asserting $PreE \;\Rrightarrow\; wp.\mathsf{Prog}.PostE$, we use the compact notation

$$\begin{array}{lll} & PostE & \\ \cdot \;\Lleftarrow & PreE & \text{applying } wp.\mathsf{Prog} \end{array}$$

so that the linear "step-by-step" layout of the proof can be continued. (The "$\cdot$" at left warns that we are not asserting $PostE \Lleftarrow PreE$ itself.)

The invariant I is $[k(\overline{\$}n) \leq K < (k{+}1)(\overline{\$}n)]/\overline{\$}n$, where $\overline{\$}n$ is $\$(n{+}1)$, the least power-of-two *strictly greater than* n; its nicer algebraic properties make the proof slightly more compact.

[3] We have specialised the rules to the terminating-loop case to save space. In general, termination is proved using "probabilistic variants" [6, 15, 13].

3.3 On initialisation: inner loop

These steps in fact could assert equalities ($\equiv$) in this case, but we use the weaker $\Leftarrow$ or $\Rrightarrow$, as appropriate, to make the method clear.

$$
\begin{array}{ll}
 & [k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)]/\overline{\$}n \qquad \text{invariant} \\
\cdot \Leftarrow & \qquad \text{applying } wp.(k, n{:=}\ 0, N{-}1) \\
 & [0(\overline{\$}(N{-}1)) \le K < (0{+}1)(\overline{\$}(N{-}1))]/\overline{\$}(N{-}1) \\
\Leftarrow & \qquad \text{arithmetic gives pre-expectation} \\
 & [0 \le K < \$N]/\$N \ .
\end{array}
$$

3.4 While iterating: inner loop

To show the invariant is maintained, we reason "backwards" through the loop body.

$$
\begin{array}{ll}
 & [k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)]/\overline{\$}n \qquad \text{invariant} \\
\cdot \Leftarrow & \qquad \text{applying } wp.(n{:=}\ n{\div}2) \\
 & [k(\overline{\$}(n{\div}2)) \le K < (k{+}1)(\overline{\$}(n{\div}2))]/\overline{\$}(n{\div}2) \\
 & \\
\cdot \Leftarrow & \qquad \text{applying } wp.(k{:=}\ 2k \ {}_{1/2}\!\oplus\ 2k{+}1) \\
 & \quad 1/2 * \left[\begin{array}{lll} & (2k)(\overline{\$}(n{\div}2)) \\ \le & K \\ < & ((2k){+}1)(\overline{\$}(n{\div}2)) \end{array}\right] /\overline{\$}(n{\div}2) \\
 & \\
 & \quad + 1/2 * \left[\begin{array}{lll} & (2k{+}1)(\overline{\$}(n{\div}2)) \\ \le & K \\ < & ((2k{+}1){+}1)(\overline{\$}(n{\div}2)) \end{array}\right] /\overline{\$}(n{\div}2) \\
 & \\
\Leftarrow & \qquad \text{arithmetic} \\
 & \quad \left[\begin{array}{lll} & (2k)(\overline{\$}(n{\div}2)) \\ \le & K \\ < & (2k{+}1)(\overline{\$}(n{\div}2)) \end{array}\right] /(2\overline{\$}(n{\div}2)) \\
 & \\
 & \quad + \quad \left[\begin{array}{lll} & (2k{+}1)(\overline{\$}(n{\div}2)) \\ \le & K \\ < & (2(k{+}1))(\overline{\$}(n{\div}2)) \end{array}\right] /(2\overline{\$}(n{\div}2)) \\
 & \\
\Leftarrow & \qquad \text{for natural number } n \ne 0 \text{ have } 2\overline{\$}(n{\div}2) = \overline{\$}n \\
 & [n \ne 0] * \\
 & \quad (\ [k(\overline{\$}n) \le K < (2k{+}1)(\overline{\$}(n{\div}2))]/\overline{\$}n \\
 & + \quad [(2k{+}1)(\overline{\$}(n{\div}2)) \le K < (k{+}1)(\overline{\$}n)]/\overline{\$}n \ ) \\
 & \\
\Leftarrow & \qquad \text{merging inequalities gives guard and invariant} \\
 & [n \ne 0] * [k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)]/\overline{\$}n \ .
\end{array}
$$

3.5 On termination: inner loop

In this case we reason forwards, towards the overall post-expectation.

$$
\begin{array}{ll}
 & \qquad \text{negated guard and invariant} \\
 & [n = 0] * [k(\overline{\$}n) \le K < (k{+}1)(\overline{\$}n)]/\overline{\$}n \\
\Rrightarrow & [k(\overline{\$}0) \le K < (k{+}1)(\overline{\$}0)]/\overline{\$}0 \qquad \text{arithmetic} \\
\Rrightarrow & [k \le K < (k{+}1)] \qquad \overline{\$}0 = 1 \\
\Rrightarrow & [k = K] \ . \qquad k, K \in \mathbb{N} \text{ gives post-expectation}
\end{array}
$$

4 The algebra of abstractions

We have shown that for any K the inner loop Inner satisfies

$$[0 \le K < \$N]/\$N \quad \Rrightarrow \quad wp.\mathsf{Inner}.[k = K] \ . \qquad (4)$$

Intuitively this clearly is a uniform choice from $[0, \$N)$, since it states that k is set to any chosen K in the stated range with (at least [4]) probability $1/\$N$. Although we have *proved* this for our given inner loop, in abstracting from the details we now take it as the *definition* of the property that any inner loop should have if it is to be used within our chosen outer loop.

From that definition we use the algebra of Fig. 2 to show that the pre-expectation for Inner is at least the *average* of any chosen post-expectation it is applied to, again over the range $[0, \$N)$. We no longer refer to the code of Fig. 4 at all; and thus any alternative code that also satisfied (4) would be sufficient as well.

We take arbitrary post-expectation *PostE*, and define constants $PostE_K$ = "the value of *PostE* when k is K". Then we reason

$$
\left.\begin{array}{ll}
 & wp.\mathsf{Inner}.PostE \\
\Leftarrow & \qquad \text{arithmetic, } wp.\mathsf{Inner} \text{ monotonic} \\
 & wp.\mathsf{Inner}.(\textstyle\sum_{0 \le K < \$N} \ PostE_K * [k = K]) \\
\Leftarrow & \qquad \text{Fig. 2 used } \$N{-}1 \text{ times} \\
 & \textstyle\sum_{0 \le K < \$N} \ PostE_K * wp.\mathsf{Inner}.[k = K] \\
\Leftarrow & \textstyle\sum_{0 \le K < \$N} \ PostE_K * [0 \le K < \$N]/\$N \quad (4) \\
\Leftarrow & \qquad \text{within summation } [0 \le K < \$N] = 1 \\
 & (\textstyle\sum_{0 \le K < \$N} \ PostE_K) \ /\$N \\
\Leftarrow & \qquad \text{change bound variable } K \text{ to } k \\
 & (\textstyle\sum_{0 \le k < \$N} \ PostE) \ /\$N \ ,
\end{array}\right\} \qquad (5)
$$

which is the averaging property we sought.[5]

5 Proofs for outer loop

The outer loop with the inner loop abstracted is shown in Fig. 7; the complete annotations for it in that form are given in Fig. 8. Since it has nonzero probability $N/\$N$ of termination on each iteration, its overall termination probability is one.

The invariant is $[k = K] \ \lhd\, k \ge N \,\rhd\ [0 \le K < N]/N$; the termination and post-expectation proofs are trivial.

[4] Recall Footnote 2.

[5] The reasoning above is very similar to the reasoning mentioned in Footnote 2 earlier.

$\{\ [0{\le}K{<}N]/N\ \}$
$k{:=}\ N;$
do $k \geq N\ \rightarrow$
  Inner
od
$\{\ [k{=}K]\ \}$

Figure 7: Outer loop, with inner loop abstracted

$\{\ [0{\le}K{<}N]/N\ \}$
$k{:=}\ N;$
$\{\ [k{=}K]\ \lhd k{\geq}N \rhd\ [0{\le}K{<}N]/N\ \}$
do $k \geq N\ \rightarrow$
  $\{\ [0{\le}K{<}N]/N\ \}$
  Inner
  $\{\ [k{=}K]\ \lhd k{\geq}N \rhd\ [0{\le}K{<}N]/N\ \}$
od
$\{\ [k{=}K]\ \}$

Figure 8: Outer loop, with annotations

5.1 While iterating: outer loop

We reason

$$
\begin{array}{lll}
 & [k=K]\ \lhd k \geq N \rhd\ [0 \leq K < N]/N & \text{invariant} \\
\cdot \Leftarrow & & \text{applying } wp.\mathsf{Inner} \text{ from (5), and } \$N \geq N \\
 & (\quad \sum_{0\leq k<N}\ [k=K] & \\
 & \quad + \ \sum_{N\leq k<\$N}\ [0 \leq K < N]/N & \\
 &)\ /\$N & \\
\Leftarrow & (\quad [0 \leq K < N] & \text{arithmetic} \\
 & \quad + \ (\$N - N)[0 \leq K < N]/N & \\
 &)\ /\$N & \\
\Leftarrow & [0 \leq K < N]/N & \text{arithmetic} \\
\Leftarrow & & \text{assuming guard } k \geq N \\
 & [k=K]\ \lhd k \geq N \rhd\ [0 \leq K < N]/N\ . &
\end{array}
$$

Now using compositionality — that the properties of Inner imply the properties of the actual inner loop code, we can transcribe the annotations of Fig. 7 directly to Fig. 4, and we are done: Fig. 3 has been validated, in a two-stage process where the stages are linked by abstraction and refinement.

6 Conclusion

We have shown how the expectation transformers provide a framework for refinement and abstraction for probabilistic programs. As well as "pen-and-paper" proofs the logic has been implemented in the HOL4 theorem-proving environment [8, 1] and proof tools have been developed to aid with the many small proof steps that are required to complete a full verification.

Although our example was deterministic in the sense that it contains only exact probabilities [13], abstraction applies even in situations where the probabilities are known only within some tolerance; indeed, it is even more important in that case (which is the rule, in practice), and is an alternative to probabilistic metrics [4] for approximate equality (bisimilarity). Sec. **??** shows how our uniform selection behaves when the underlying random bit-source is only approximately unbiased.

Future research in this area will work towards on augmenting the abstraction techniques with results from model-checking [10, 11], enhancing the capabilities of both methods.

References

[1] O. Celiku and A. McIver. Cost-based analysis of probabilistic programs mechanised in HOL. *Nordic Journal of Computing*, 11(2):102–128, 2004.

[2] E. Dijkstra. *A Discipline of Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1976.

[3] Y. A. Feldman and D. Harel. A probabilistic dynamic logic. *J. Computing and System Sciences*, 28:193–215, 1984.

[4] A. Giacalone, C.-C. Jou, and S. Smolka. Algebraic reasoning for probabilistic concurrent systems. In *Proc. IFIP WG 2.2/2.3 Working Conf.*, pages 443–58, Apr. 1990.

[5] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–35, 1994.

[6] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Trans. Prog. Lang. Syst.*, 5:356–80, 1983.

[7] J. He, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Sci. Comput. Programming*, 28:171–92, 1997. Available at [14, key HSM95].

[8] J. Hurd, A. McIver, and C. Morgan. Probabilistic guarded commands mechanised in HOL. To appear in *Proc. QAPL '04 (ETAPS)*, 2005.

[9] D. Kozen. A probabilistic PDL. *Jnl. Comp. Sys. Sciences*, 30(2):162–78, 1985.

[10] M. Kwiatkowska. Model checking for probability and time: from theory to practice. In P. G. Kolaitis, editor, *Proc. Eighteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2003*, pages 351–360. IEEE Computer Society Press, June 2003.

[11] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.

[12] A. McIver. Quantitative program logic and expected time bounds in probabilistic distributed algorithms. *Theoretical Comput. Sci.*, 282(1):191–219, 2002.

[13] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Technical Monographs in Computer Science. Springer Verlag, New York, 2004.

[14] A. McIver, C. Morgan, J. Sanders, and K. Seidel. Probabilistic Systems Group: Collected reports. `web.comlab.ox.ac.uk/oucl/research/areas/probs`.

[15] C. Morgan. Proof rules for probabilistic loops. In H. Jifeng, J. Cooke, and P. Wallis, editors, *Proceedings of the BCS-FACS 7th Refinement Workshop*, Workshops in Computing. Springer Verlag, July 1996.
`ewic.bcs.org/conferences/1996/...`
`...refinement/papers/paper10.htm`.

[16] C. Morgan and A. McIver. *pGCL*: Formal reasoning for random algorithms. *South African Computer Journal*, 22, Mar. 1999. Available at [14, key pGCL].

[17] C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Trans. Prog. Lang. Syst.*, 18(3):325–53, May 1996.
`doi.acm.org/10.1145/229542.229547`.

[18] P. Whittle. *Probability via Expectation*. Springer Texts in Statistics. Springer Verlag, third edition, 1992.