Quantitative refinement *and* model checking for the analysis of probabilistic systems

AK McIver[1]

Dept. Computer Science, Macquarie University, NSW 2109 Australia;
anabel@ics.mq.edu.au

Abstract. For standard (ie non-probabilistic) systems of reasonable size, correctness is analysed by simulation and/or model checking, possibly with standard program-logical arguments beforehand to reduce the problem size by abstraction.
For probabilistic systems there are model checkers and simulators too; but probabilistic program logics are rarer. Thus e.g. model checkers face more severe exposure to state explosion because "front-end" probabilistic abstraction techniques are not so widely known [20].
We formalise probabilistic refinement of action systems [3] in order to provide just such a front end, and we illustrate with the probabilistic model checker PRISM [24] how it can be used to reduce state explosion. The case study is based on a performance analysis of randomised backoff in wireless communication [1].
Keywords: Probabilistic abstraction and refinement, structured specification and analysis of performance, probabilistic model checking.

1 Introduction

The analysis of performance properties of systems requires a quantitative assessment of behaviour, and currently there are two major styles of quantitative verification.

Probabilistic model checking links a "programming" language (for describing the system operationally) with a "logical" language (for describing the desired properties). The latter is usually a form of probabilistic temporal logic expressing e.g. "with probability p the system will eventually establish G". The model checker checks automatically whether the former satisfies the latter.

Proof-based methods, on the other hand, are more general in principle but harder to use in practice. For them, the link between the operational description and the desired properties is made by a formal proof; but the proof is not provided automatically — it must be figured out (by a human).

Although our main focus is on proof, our aim is to combine the two in a way that takes the best from each, and avoids many of their individual limitations. We use the proof-based methods to analyse only *part* of the system, concentrating on exposing a high-level abstraction which preserves the properties of interest. We take this as far as we can, for the more we abstract the smaller the state space will be; but when the details begin to bite, making continued progress with

proof either difficult or impossible, we switch over to model checking — and just push the button.

The key for the abstraction is "preserves the properties of interest". For this we use *action systems* [3, 26, 9] enhanced with probability based on a (sequential) probabilistic program logic [20]. Action systems are sufficiently general to connect the sequential program logic, and its probabilistic programming language pCSP [21], to the modelling language used by probabilistic model checkers such as PRISM [15].

Our contributions are as follows

1. *An extension of probabilistic action systems* of Morgan [21] to include synchronisation (Sec. 3.1), hiding (Sec. 3.2) and property-preserving refinement (Sec. 3.3);
2. *Proof rules for probabilistic refinement* in the context of hiding, extending those used in standard frameworks [7] (Sec. 3);
3. *A mapping* from a subset of action systems to PRISM's modelling language, allowing quantitative specifications to be verified using PRISM (Sec. 4);

The contributions are illustrated by a small case study (Sec. 5), demonstrating how the probabilistic backoff procedure used in wireless communication may be developed in a stepwise fashion, appealing to model checking to investigate detailed performance issues. Our experiments using the PRISM model checker indicate that the technique of probabilistic refinement can substantially reduce the state space overhead, whilst maintaining the integrity of the performance analysis in subsequent refinements, without the need for further model checking.

The notational conventions used are as follows. Function application is represented by a dot, as in $f.x$. We use an abstract state space S, and denote the set of discrete probability distributions over S by $\overline{S}$ (that is the sub-normalised functions from S into the real interval $[0, 1]$, where function f is sub-normalised if $\sum_{s:\, S} f.s \leq 1$). Given predicate *Pred* we write $[\mathit{Pred}]$ for the *characteristic* function mapping states satisfying *Pred* to 1 and to 0 otherwise, punning 1 and 0 with "True" and "False". The $(p, 1-p)$-weighted average of distributions d and d' is denoted $d \,{}_p\!\oplus\, d'$.

2 Probabilistic guarded commands

When programs incorporate probability, their properties can no longer be guaranteed "with certainty", but only "up to some probability". For example the program

$$\mathit{coin} \quad \hat{=} \quad b := \mathsf{T} \;{}_{2/3}\!\oplus\; b := \mathsf{F}\,, \tag{1}$$

sets the Boolean-valued variable b to T only with probability $2/3$ — in practice this means that if the statement were executed a large number of times, and the final values of b tabulated, roughly $2/3$ of them would record b having been set to T (up to well-known statistical confidence [8]).

The language pGCL and its associated *quantitative logic* [20] were developed to express such programs and to derive their probabilistic properties by extending the classical assertional style of programming [22]. Programs in pGCL are modelled (operationally) as functions (or transitions) which map *initial states* in S to (sets of) probability distributions over *final states* — the program at (1) for instance has a single transition which maps any initial state to a (single) final distribution; we represent that distribution as a function d, evaluating to $2/3$ when $b = \mathsf{T}$ and to $1/3$ when $b = \mathsf{F}$.

Since properties are now quantitative we express them via a logic of *real-valued functions*, or *expectations*. For example the property "the final value of b is T with probability $2/3$" can be expressed as the *expected value* of the function $[b = \mathsf{T}]$ with respect to d [8], which evaluates to $2/3 \times 1 + 1/3 \times 0 = 2/3$.

Direct appeal to the operational semantics quickly becomes impractical for all but the simplest programs — better is the equivalent transformer-style semantics which is obtained by rationalising the above calculation in terms of expected values rather than transitions, and the explanation runs as follows. Writing $\mathcal{E}S$ for the set of all (non-normalised) functions from S to $[0, 1]$, which we call the set of *expectations*, we say that the expectation $[b = \mathsf{T}]$ has been transformed to the expectation $2/3$ by the program *coin* (1) above so that they are in the relation "$2/3$ is the expected value of $[b = \mathsf{T}]$ with respect to the *coin*'s result distribution". More generally given a program *Prog*, an expectation E in $\mathcal{E}S$ and a state $s \in S$, we define $\mathsf{wp}.\mathit{Prog}.E.s$ to be the expected value of E with respect to the result distribution of program *Prog* if executed initially from state s [20]. We say that $\mathsf{wp}.\mathit{Prog}$ is the *expectation transformer* relative to *Prog*. In our example that allows us to write

$$2/3 \quad \equiv \quad \mathsf{wp}.(b := \mathsf{T} \,{}_{2/3}\!\oplus\, b := \mathsf{F}).[b = \mathsf{T}] \ .$$

In the case that *nondeterminism* is present, execution of *Prog* results in a *set* of possible distributions and we modify the definition of wp to take account of this — indeed we define $\mathsf{wp}.\mathit{Prog}.E.s$ so that it delivers the *least*-expected value with respect to all distributions in the result set. The transformers [20] give rise to a complete characterisation of probabilistic programs with nondeterminism, and they are sufficient to express many performance-style properties, including the probability that an event occurs [20], the expected time that it occurs [20], and long-run average of the number of times it occurs over many repeated executions of the system [18].

In Fig. 1 we set out the semantics for pGCL, a variation of Dijkstra's GCL with several extensions and modifications, all of which are labelled by (†). They are miracles, probability and "unguarded iteration" [4], the last representing a looping program which can iterate for an indeterminate length of time, a behaviour typifying reactive systems. All the programming features have been defined previously elsewhere, and (apart from probabilistic choice) have interpretations which are merely adapted to the real-valued context. For example nondeterminism, as explained above, is interpreted *demonically* and can be thought of as being resolved by a "minimal-seeking demon", providing guarantees on all

program behaviour, such as is expected for total correctness. *Probabilistic choice*, on the other hand, selects the operands at random with weightings determined by the probability parameter p.

In addition to the definitions in Fig. 1 we will use the function *gd* which, applied to a command, defined the states from which the command is enabled.

$$gd.C \quad \hat{=} \quad (1 - \mathsf{wp}.C.0) \ , \tag{2}$$

so that when applied to a command of the form $G \to Prog$ returns $[G]$; we also refer to *Prog* in $G \to Prog$ as its *body*. Given a family $\mathcal{I}$ of commands we write $[\!]_{i:\mathcal{I}} C_i$ for the generalised (nondeterministic) choice over the family, and $\sum_{i\in\mathcal{I}} C_i @ p_i$ for the generalised probabilistic choice (where $\sum_{i\in\mathcal{I}} p_i = 1$). Similarly for variable v we write $n :\in Pred$ for the generalised nondeterministic choice over any value from its type that satisfies the predicate *Pred*.

We say that a command is *normal* if it is of the form of a generalised probabilistic choice over standard (non-probabilistic) commands F_i, i.e. of the form $\sum_{i\in\mathcal{I}} F_i \ @p_i$, where $\sum_{i\in\mathcal{I}} p_i \leq 1$. We shall need to be able to compose the effect of "running" commands simultaneosly, and the next definition sets out how to do it.

Definition 1. *Given normal guarded commands $C \mathrel{\hat{=}} G \to Prog$ and $C' \mathrel{\hat{=}} G' \to Prog'$, we define their* composition *as follows.*

$$C \otimes C' \quad \hat{=} \quad (G \wedge G') \to \sum_{(i,j)\in I\times J} (F_i \otimes F'_j) \ @(p_i \times p'_j) \ ,$$

where $Prog = \sum_{i\in I} F_i \ @p_i$ and $Prog' = \sum_{j\in J} F'_j \ @p'_j$, and $\mathsf{wp}.F \otimes F'$ is given by the fusion *operator of Back and Butler [4]. In the case that F and F' operate over distinct state spaces (as in our case study), $F \otimes F'$ is equivalent to $F \ ; \ F'$.*

We end this section with a list of some nice features and idioms of pGCL.

• Unguarded iteration satisfies the equation $\mathsf{it}\ C\ \mathsf{ti} = \mathsf{skip} \ [\!] \ C; \ (\mathsf{it}\ C\ \mathsf{ti})$, expressing the fact that termination may occur at any time.

• Following from above, $\mathsf{it}\ \mathsf{magic}\ \ \mathsf{ti} = \mathsf{skip} \ [\!] \ \mathsf{magic} \ ; \ (\mathsf{it}\ \mathsf{magic}\ \ \mathsf{ti}) = \mathsf{skip}$.

• When C is a guarded command $G \to Prog$, the expression

$$\mathsf{wp}.(\mathsf{it}\ C\ \mathsf{ti}).1 \ , \tag{3}$$

is the greatest guaranteed probability that the command (if executed) must establish $\neg G$ eventually. If this is 1 at any state, it means that the demon is obliged to terminate after a finite number of executions of C (with probability 1) to minimise the risk of a miracle; if it is 0 however it means that C may be executed forever, without ever establishing $\neg G$.

For example when G is "$(n > 0)$", and *Prog* is "$n := n + 1$", then (3) is the (standard) $[n \leq 0]$, since when n is greater than 0, the command can increase n indefinitely, and when n is less than 0 the interior command cannot execute even once. On the other hand if *Prog* is "$n := n + 1 \ {}_{2/3}\!\oplus n := n - 1$" (and G

skip	$\mathsf{wp.skip}.E \mathrel{\hat{=}} E$,
abort	$\mathsf{wp.abort}.E \mathrel{\hat{=}} 0$,
magic(†)	$\mathsf{wp.magic}\ .E \mathrel{\hat{=}} 1$,
assignment	$\mathsf{wp}.(x := f).E \mathrel{\hat{=}} E[x := f]$,
sequential composition	$\mathsf{wp}.(r; r').E \mathrel{\hat{=}} \mathsf{wp}.r.(\mathsf{wp}.r'.E)$,
probabilistic choice(†)	$\mathsf{wp}.(r\ {}_p\!\oplus r').A \mathrel{\hat{=}} p \times \mathsf{wp}.r.E + (1-p) \times \mathsf{wp}.r'.E$,
nondeterministic choice	$\mathsf{wp}.(r \parallel r').A \mathrel{\hat{=}} \mathsf{wp}.r.E \sqcap \mathsf{wp}.r'.E$, where $\sqcap$ is pointwise minimum,
Boolean choice	$\mathsf{wp}.(\mathsf{if}\ G\ \mathsf{then}\ r\ \mathsf{else}\ r').E \mathrel{\hat{=}} [G] \times \mathsf{wp}.r.E + [\neg G] \times \mathsf{wp}.r'.E$,
Guarded command	$\mathsf{wp}.(G \to r).E \mathrel{\hat{=}} [G] \times \mathsf{wp}.r.E + [\neg G]$,
Unguarded iteration(†)	$\mathsf{wp}.(\mathsf{it}\ r\ \mathsf{ti}).E \mathrel{\hat{=}} (\mu X \bullet E \sqcap \mathsf{wp}.r.X)$.

E is an expectation in $\mathcal{E}S$, and f is a function of the state. The real p is restricted to lie between 0 and 1, and the term $(\mu X \dots)$ refers to the least fixed point with respect to $\leq$, which we lift to real-valued functions; it is guaranteed to exist since the expectations form a complete partial order. Commands labelled † are extensions of the standard pGCL. Commands are ordered using *refinement*, so that more refined programs improve probabilistic results, thus

$$P \sqsubseteq Q \quad \textit{iff} \quad (\forall E \in \mathcal{E}S \cdot \mathsf{wp}.P.E \leq \mathsf{wp}.Q.E) \ .$$

Fig. 1. Structural definitions of wp for pGCL with miracles and unguarded iteration.

remains "$(n > 0)$"), then solving the fixed point equation (for the least solution) tells us that (3) is $1/2$ evaluated at "$n = 1$", implying that there is only $1/2$ chance that the command must establish $(n \leq 0)$ if iterated indefinitely. (And indeed the larger the initial value of n, the more likely is it that the command may be iterated forever, without incurring a miracle at all.[1])

• $gd.C \times \mathsf{wp}.C.post$ selects the greatest guaranteed expected value of *post* from initially enabled states.

• We write $\langle G \rangle$ for the "coercion" $G \to \mathsf{skip}$, the command which behaves like magic if G does not hold. Similarly we use $\{G\}$ for its dual, if G then skip else abort, the "assertion", which behaves like skip if G holds, and aborts if it doesn't. We say that a command C terminates, or is *total* if $\mathsf{wp}.C.1 = 1$.

3 Probabilistic action systems

Action systems [3] are a "state-based" formalism for describing so-called reactive systems, *viz.* systems that may execute indefinitely. Although others [9, 26] have added probability to action systems, our work is most closely related to Morgan's version of labelled probabilistic action systems [21], which we extend in various ways described below.

[1] See [20], page 287 for a wp-proof of this fact: it is an example of the asymmetric random walk.

A (probabilistic) action system consists of a (finite) set of labelled guarded commands, together with a distinguished command called an initialisation. An action system is said to *operate* over a state space S, meaning that the variables used in the system define its state space. Operationally an action system first executes its initialisation, after which any labelled action may "fire" if its guard is true by executing its body. Actions may continue to fire indefinitely until all the guards are false. If more than one guard is true then any one of those actions may fire, nondeterministically. We reserve the label τ for "hidden" actions, discussed later in Sec. 3.2.

In Fig. 2 we set out a small example of a probabilistic action system *Walker* which operates over the state defined by its variable n. First n is set nondeterministically either to -1 or 1, and then action a or b fires depending on whether n is greater than or less than 0, terminating if n is ever set to 0 (which, incidently, occurs with probability 1). In terms of actions, *Walker* executes alternately a string of a's or b's, whose frequency is determined by the probabilistic transitions.

$$\mathit{Walker} \mathrel{\hat{=}} \begin{pmatrix} \textbf{var } \mathsf{n} : \mathbb{Z} \\ \textbf{initially } n :\in \{-1, 1\} \\ a: \ (n > 0) \to \ n := n+1 \ {}_{1/3}\!\oplus \ n := n-2 \\ b: \ (n < 0) \to \ n := n+2 \ {}_{2/3}\!\oplus \ n := n-2 \end{pmatrix}$$

Fig. 2. A random walker with actions.

For action system P and label a we write P_a for the generalised choice of all actions labelled with a, and P_i for its initialisation. In the case that P has no a action we define P_a to be magic . The set of non-τ labels (labelling actions in P) is denoted $\alpha.P$, and called P's *alphabet*. The semantics of an action system is given by pGCL set out at Fig. 1, so that, for example, $gd.(P_a) \times \mathsf{wp}.P_a.E$ is the greatest guaranteed expected value of E from execution of P_a.

We use the action labels in two important ways — the first is to define *synchronisation* and the second is to define *refinement* of action systems. We deal with both below.

3.1 Synchronising actions

We define synchronisation so that all action systems participating in a parallel composition simultaneously fire their choice-shared actions together — in this mode the nondeterminism (arising from possibly overlapping guards) is resolved first, followed by any probability in the bodies. All other actions fire independently, interleaving with any others.

Definition 2. *Given normal action systems P and Q and subset A of actions (not containing τ), we define their* parallel composition $P||_A Q$ *as follows.*

1. $P||_A Q$ *operates over the union of the two state spaces, and* $\alpha.(P||_A Q) = \alpha.P \cup \alpha.Q$;
2. $(P||_A Q)_i \quad \hat{=} \quad P_i \otimes Q_i$;
3. $(P||_A Q)_b \quad \hat{=} \quad P_b \parallel Q_b$, *for* $b \notin A$;
4. $(P||_A Q)_a \quad \hat{=} \quad \parallel_{\{P^a \in P, Q^a \in Q\}} P^a \otimes Q^a$ *for* $a \in A$, *where* P^a *and* Q^a *are the individual* a*-labelled actions belonging to* P *and* Q *respectively.*

Note that the normality is preserved by Def. 2.

3.2 Hiding actions

Our notion of abstraction is founded on the use of τ to indicate actions which are hidden *viz.* those actions that execute so-to-speak "behind the scenes". The idea is to imagine an observer of the system judging its behaviour only on the passage of non-τ-labelled actions. Thus he does not notice the actual firing of the τ actions, according any state change they may induce instead to the actions he has witnessed. The next definition sets out the details.

Definition 3. *Given a labelled action system* P*, and a set of labels* H*, we define the action system* $P\backslash H$ *as follows:* [2]

1. $P\backslash H$ *operates over the same state space as* P*, and* $\alpha.(P\backslash H) \hat{=} (\alpha.P) - H$;
2. $(P\backslash H)_a \hat{=} P_a$, *if* $a \notin H$;
3. $(P\backslash H)_\tau \hat{=} P_\tau \parallel \quad (\parallel_{h \in H} P_h)$.

Thus *Walker*$\backslash\{b\}$ is the action system in which the only "observed actions" are a's. In between any a-action, an arbitrary number of the (now) hidden b-actions may be executed unobserved, but because $\mathsf{wp}.(\mathsf{it}\ \mathit{Walker}_b\ \mathsf{ti}).1 = 1$, only finitely many can occur between every a action.[3] On the other hand Def. 3 forces hidden actions to fire in the case that *only* hidden actions are enabled. In the next section we show how the hidden actions affect refinement.

3.3 Action refinement

The ability to compare programs' behaviour is the mainstay of specification and refinement, and when used in conjunction with hiding is crucial for stepwise development of correct systems. Even though refinements normally contain more detail (and are thus more complicated) than the specifications from which they are derived, refinement ensures that they satisfy at least as many properties. Here we set out a formal treatment of property-preserving refinement (including quantitative properties) together with the principles underlying its definition.

In standard state-based programs, the refinement relation is applied to "the complete execution of the program". In this context however, there is no natural notion of "complete execution", as many actions may need to be fired to

[2] This definition is similar to Butler and Morgan's definition [4] in terms of refinement.

[3] It is possible to introduce "divergence" after hiding — a situation in which hidden actions may iterate indefinitely, and which is equivalent to abortion — however we do not discuss that here.

achieve some goal. Here we combine the principles of event-based formalisms (such as *Event B* [2]) and the probabilistic state-based approach to define a relation between probabilistic action systems in which the labels determine the observable behaviour. Our definition of refinement next compares the behaviour of observable — i.e. labelled — events, event-by-event. [4]

Definition 4. *Let P and Q be action systems operating over the same state spaces. We say that P is* refined *by Q, or $P \preceq Q$, if* [5]

1. P_i ; it P_τ ti $\sqsubseteq$ Q_i ; it Q_τ ti
2. it P_τ ti ; P_a ; it P_τ ti $\sqsubseteq$ it Q_τ ti ; Q_a; it Q_τ ti, *for each* $a \in \alpha.Q$,

where $\sqsubseteq$ is defined at Fig. 1 above.

For example, we see that $Walker\backslash\{b\} \preceq ReflectingWalk$ (where *ReflectingWalk* is defined in Fig. 3) since the effect of an arbitrary number of hidden actions in $Walker\backslash\{b\}$ (formerly the b actions) is summarised by the single τ event in *ReflectingWalk*. (Recall that hidden actions are forced to execute until a visible event is enabled.)

3.4 Property preservation

The significance of a refinement relation is that it preserves properties, and in this section we set out how to define and test quantitative properties, and show that they are preserved by Def. 4. Our approach is via a small testing language in the style of concurrent logics [6], the key idea being that investigating a test is simpler than investigating all behaviours of a program in all contexts. In this case an action system is tested by incorporating it in a special testing program which can only do three things: either it aborts, exhibits a miracle or terminates in a valid state. Only in the latter case does the action system pass the test. Thus algorithms which implement the test then only need check for termination.

Definition 5. *Given an alphabet A we define a test T as follows.*

$$T \quad \hat{=} \quad \{G\} \mid \langle G \rangle \mid T;T \mid \; [\!] \; _{a \in K} a \mid \text{it } T \text{ ti} \; ,$$

where a is any label in A, and G and K are constant symbols.

A test t is to be understood in the context of an interpretation (set out at Def. 6 below) in which the labels correspond to the execution of actions in a given action system (effectively procedure calls), and the constants correspond to predicates over the variables (G), or subsets of actions (K). Thus tests specify sequences of labelled actions and named conditions corresponding to complex temporal formulae which may or may not be satisfied by particular instances of action systems. Our testing language is expressive enough to capture some

[4] Compare the definition of state-based data-refinement [12].

[5] Note that hidden actions cannot be compared directly since in some cases one action system might not have hidden actions, whilst the other might.

important classes of properties sufficient to make our connection to model checking "reachability" results. For example the property "predicate B holds until G does" can be specified by

$$B \triangleright G \quad \hat{=} \quad \mathsf{it}\ \langle \neg G \rangle\ ;\ (\ [\!]_{a:\,A} a)\ \mathsf{ti}\ ;\ \langle G \vee \neg B \rangle\ ;\ \{G\}\ . \tag{4}$$

To see that observe that if $\neg G \wedge B$ holds (for a particular interpretation), the minimal-seeking nature of the nondeterminism will force it to fire any enabled action rather than terminate the iteration, since in that case execution of the coercion $\langle G \vee \neg B \rangle$ will result in a miracle. Similarly if $\neg G \wedge \neg B$ holds the demon will jump out of a loop resulting in a following abort (execution of $\{G\}$), but if G holds, the demon chooses between either a miracle or simply to terminate, the latter being the better option. [6]

Next we can interpret a test over an action system P simply by instantiating each label a by P_a, factoring in the initialisation and the hidden events, in the latter case by allowing an arbitrary number of hidden events to come before or after any visible event.

Definition 6. *For action system P such that $\alpha.P \subseteq A$, we interpret a test t over P as the pGCL program $\{\!|t|\!\}^{\mathcal{V}}_{P}$ (defined below), where $\mathcal{V}$ maps constants G to predicates over S, and constants K to finite, nonempty subsets of actions from A.*

$$\{\!|t|\!\}^{\mathcal{V}}_{P} \quad \hat{=} \quad P_i\ ;\ \mathsf{it}\ P_\tau \mathsf{ti}\ ;\ \|t\|^{\mathcal{V}}_{P}\ , \textit{where}$$

$$\|t\|^{\mathcal{V}}_{P} \quad \hat{=} \quad \begin{cases} \{\mathcal{V}.G\} \textit{ or } \langle \mathcal{V}.G \rangle & , \textit{if } t = \{G\} \textit{ or } \langle G \rangle \\ \|t'\|^{\mathcal{V}}_{P}\ ;\ \|t''\|^{\mathcal{V}}_{P} & , \textit{if } t = t'; t'' \\ [\!]_{a \in \mathcal{V}.K} (\mathsf{it}\ P_\tau \mathsf{ti}\ ;\ P_a\ ;\ \mathsf{it}\ P_\tau \mathsf{ti}) & , \textit{if } t = [\!]_{a \in K} a \\ \mathsf{it}\ \|t'\|^{\mathcal{V}}_{P}\ \mathsf{ti} & , \textit{if } t = \mathsf{it}\ t'\ \mathsf{ti}\ . \end{cases}$$

Observe that since $\mathsf{it}\ Q\ \mathsf{ti}\ ;\ \mathsf{it}\ Q\ \mathsf{ti}\ =\ \mathsf{it}\ Q\ \mathsf{ti}$ for any Q, and that hidden actions surround any visible action, there is no risk that Def. 6 inserts too many (or too few) hidden actions. We can now prove our property-preserving character of refinement, which tells us that properties expressed in the wp-style are all preserved.

Lemma 1. *If action systems $P \preceq Q$ then for any test t and any $\mathcal{V}$ mapping as in Def. 6, we have that $\{\!|t|\!\}^{\mathcal{V}}_{P} \sqsubseteq \{\!|t|\!\}^{\mathcal{V}}_{Q}$.*

Proof. From Def. 6, and Def. 5, we see that interpretations of properties as tests are pGCL programs; the result now follows from the definition of $\sqsubseteq$ at Fig. 1.

Thus (abusing notation so that constants stand for themselves) Def. 6 tells us that $\mathsf{wp}.\{\!|(n < 0) \triangleright (n = 1)|\!\}^{\mathcal{V}}_{Walker}.[n = 1]$, computes the chance that n becomes 1, after being negative by iterating the actions of *Walker*, in this case it is only possible if n is initially negative and odd. Moreover Lem. 1 ensures that

[6] Note that this explanation is valid only when the action system always has some action enabled — in the simple case of magic for example, the miracle will automatically "guarantee" success.

$$\mathit{ReflectingWalk} \mathrel{\hat{=}} \begin{pmatrix} \textbf{var } \mathsf{n} \colon \mathbb{Z} \\ \textbf{initially } n :\in \{-1, 1\} \\ a : (n > 0) \rightarrow \ n := n + 1 \ {}_{1/3}\!\oplus n := n - 2 \\ \tau : (n < 0) \rightarrow \mathsf{if}\ (\mathsf{odd}.n)\ \mathsf{then}\ n := 1\ \mathsf{else}\ n := 0 \end{pmatrix}$$

Fig. 3. A walker with a reflecting barrier.

this result applies to *ReflectingWalk* as well, a fact easily checked by inspecting Fig. 3.

In this section we have set out our correctness criteria in the form of tests, and a definition of property-preserving refinement with the assumption that the action systems operate over the same state spaces. In many cases, however, as systems are developed, the introduction of more variables is required (effectively changing the state space) as the various system features are implemented. In the next section we provide the theory to give access to such refinements.

3.5 Changing variables

To deal with different state spaces we use the technique familiar to the treatment of standard datatypes with named operations [7]. We partition the state space between "global" and "local" variables — the idea is that two action systems having different state spaces can still be compared by looking at their respective properties restricted to their *shared* global state. [7] As for datatypes we are then able to match corresponding execution paths using a "simulation" function which converts P's "local state" into Q's, so that the refinement relation Def. 4 thus applies.

Definition 7. *Given action systems P and Q with global variables g, and local variables a and c respectively, we say that $P \preceq_g Q$, or Q* refines *P with respect to g if there is a standard (i.e. non-probabilistic), non-miraculous and terminating* simulation *program rep, mapping variables a to c such that*

1. $P_i \ ; \ rep \quad \sqsubseteq \quad Q_i$;
2. $P_a \ ; \ rep \quad \sqsubseteq \quad rep \ ; \ Q_a \ , \quad$ *for* $a \in \alpha.Q$
3. $(\mathsf{it}\ P_\tau\ \mathsf{ti}) \ ; \ rep \quad \sqsubseteq rep \ ; \ (\mathsf{it}\ Q_\tau\ \mathsf{ti})$;
4. $\mathsf{skip}_g \quad \sqsubseteq \quad rep \ ; \ \mathsf{skip}_g$;

where skip_g is a special "do nothing" program which projects the state onto that defined by the global variables g.

If P_τ is magic , the third condition of Def. 7 (normally) reduces to a proof of termination for the iteration — this can be done using *probabilistic variants*, discussed elsewhere [20].

The next lemma sets out our property-preservation criterion.

[7] If they do not share any global state — an unusual situation — then condition 4 in Def. 7 is the same as saying that *rep* must terminate.

Lemma 2. *Given action systems P and Q with shared (global) variables g, any test t, and mapping $\mathcal{V}$ mapping any constants G in t to predicates over g, then if $P \preceq_g Q$ we must have also that $\{\!|t|\!\}^{\mathcal{V}}_P \ ; \ \mathsf{skip}_g \quad \sqsubseteq \quad \{\!|t|\!\}^{\mathcal{V}}_Q \ ; \ \mathsf{skip}_g$.* [8]

Proof. Def. 7, Def. 5, Def. 6 and monotonicity. The complete proof is set out in the appendix.

In this section we set out how to prove property-preserving refinments. In the next section we discuss how those properties may be verified using probabilistic model checking.

4 Probabilistic model checking

The PRISM model checker [15] comprises a system description language together with a property specification language based on probabilistic temporal logic. Although the operational interpretation (including the definition of synchronisation and hiding) are semantically identical to deterministic normal action systems, PRISM does not have a facility for refinement in the style set out here. In terms of a specification and refinement task, Lem. 3, next, shows how we can appeal to PRISM as an "oracle" to compute quantitative properties which by Lem. 2 will be preserved for all subsequent refinements. [9]

Lemma 3. *Given deterministic normal action systems $P_1, \ldots P_n$, we have that*

$$\mathsf{wp}.\{\!|(B \triangleright G)|\!\}^{\mathcal{V}}_P.[G].s_0 \quad = \quad [\mathcal{V}.B \ \mathcal{U} \ \mathcal{V}.G] \ ,$$

where $(B \triangleright G)$ is defined at (4), $P \mathrel{\hat{=}} P_1||_A \ldots ||_A P_n$, A is the union of the alphabets of the P_i, and the PRISM formula $[\mathcal{V}.B \ \mathcal{U} \ \mathcal{V}G]$ is interpreted as "the least probability that $\mathcal{V}.B$ holds until until $\mathcal{V}.G$" does relative to P from intial s_0.[10]

Proof. (Sketch.) This follows from the interpretation of probabilistic temporal logic as distributions over sequences of states generated by the system P [19], and the operational model of PRISM.

5 A stepwise development of probabilistic backoff

In this section we develop a small case study based on the probabilistic backoff procesdure of the IEEE 802.11 standard for wireless communication [1].

[8] In the special case that P's local variables are mapped to Q's local variables, we may express a test more conveniently using local state, and the results interpreted over P will still apply to Q provided all the constants G are related by *rep* in the two interpretations.

[9] Note the conventions relating to "local" and global" variables within PRISM differ from ours.

[10] A similar result holds for "the greatest probability".

5.1 Two senders and a receiver

We begin with a very simple specification of part of a network consisting of two "sending stations" and one "receiver", given by

$$Network_0 \quad \hat{=} \quad Sender_A \;||_{\mathcal{A}}\; Receiver_{AB} \;||_{\mathcal{A}}\; Sender_B \;,$$

where the definitions of the sender and receiver are set out in Fig. 4, and $\mathcal{A}$ is the set of all labelled events. This very straightforward specification depicts a scenario in which senders A and B both need to send messages to a receiver, who alternately listens, and then acknowledges any message which arrives safely. Although in reality there may be collisions, and message loss, due to shared channels those details are not included, since the intention is that "in the end" both senders should send their messages intact. The $Network_0$ comprises a single action system, which nondeterministically delivers a message first from one of the senders, and then from the other in some order. Amongst the facts that may be proved about this system is that an acknowledgement cannot precede a sending event, by investigating formula $[(\neg s_a = wait \wedge \neg s_a = sent)\; \mathcal{U}\; (\neg s_a = sent)]$ using PRISM. [11]

$Sender_A \;\hat{=}$

$$\left(\begin{array}{l} \textbf{var}\ s_a \colon \{wait, sent, recv\} \\ \textbf{initially}\ s_a := wait \\ send_a : (s_a = wait) \rightarrow s_a := sent \\ ack_a : \ (s_a = sent) \rightarrow s_a := recv \end{array}\right)$$

$Receiver_{AB} \;\hat{=}$

$$\left(\begin{array}{l} \textbf{var}\ r \colon \{listen, ack\} \\ \textbf{initially}\ r := listen \\ send_a : (r = listen) \rightarrow r := ack \\ ack_a : \ (r = ack) \rightarrow r := listen \\ send_b : (r = listen) \rightarrow r := ack \\ ack_b : \ (r = ack) \rightarrow r := listen \end{array}\right)$$

Fig. 4. A sender and a receiver.

5.2 Introducing collisions

Next we introduce the possibility of collisions, indicating that both senders attempt to send over a shared "channel" at the same time. To do this we introduce an action system $Channel_{AB}$ depicted in Fig. 5 to model the shared channel, and augment the senders with a new event "*clash*". the channel can be blocked — indicating that two senders try to use it at the same time — or clear; but can only be blocked for a limited time, since the probabilistic choice ensures that event *clash* may only occur for a finite number of times. As we're interested in

[11] Normally this safety property requires a greatest fixed point, however since s_a eventually satisfies *sent* or *wait* the least fixed point is valid.

the expected number of times the clashing event occurs before one of the senders' messages gets through, we include a variable t whose only purpose is to count clashing events. The intermediate

$$Sender'_A \mathrel{\hat{=}} \left(\begin{array}{l} \textbf{var } s_a : \{wait, sent, recv\} \\ \textbf{initially } s_a := wait \\ send_a : (s_a = wait) \rightarrow s_a := sent \\ ack_a : \ (s_a = sent) \rightarrow s_a := recv \\ clash : \ \mathsf{skip} \end{array}\right) \qquad Channel_{AB} \mathrel{\hat{=}} \left(\begin{array}{l} \textbf{var } c : \{block, clear\},\ t : \{0 \ldots T\} \\ \textbf{initially } c := block\ ;\ t := 0 \\ clash : (c = block) \rightarrow t := t + 1; \mathsf{skip}\ {}_p\!\oplus c := clear \end{array}\right)$$

Fig. 5. Adding a clashing event and a channel.

$$Network_1 \quad \hat{=} \quad Sender'_A \ ||_{\mathcal{A}}\ Receiver_{AB} \ ||_{\mathcal{A}}\ Sender'_B \ ||_{\{clash\}} Channel_{AB}\ ,$$

does not include the precise details of the mechanism employed to clear the channel, but only the overall effect *viz.* that it does indeed clear. Our first task is to show that $Network_1$ is a refinement of $Network_0$.

Lemma 4.

$$Network_0 \quad \preceq_g \quad Network_1 \setminus \{clash\}\ ,$$

where g represents the variables s_a, t and r.

Proof. Use the function rep $\hat{=} c :\in \{block,\ clear\}\ ;\ t :\in \{0, \ldots, N\}$, noting that it distributes through any guard since it is standard so that the comment after Def. 7 applies, and this we only need prove the termination of (it $Network_1 \setminus \{clash\}$ ti)*. But from Def. 3*

$$(Network_1 \setminus \{clash\})_\tau = (Channel_{AB})_{clash}\ ,$$

and it $(Channel_{AB\,clash})$ ti *must terminate with probability* 1*, a fact that can be checked using a probabilistic variant [20].*

Lem. 4 together with Lem. 2 imply that the properties verified for $Network_0$ still hold of $Network_1$, and we do not need to check them again directly. Moreover at this point it is possible to obtain some indication of the performance of the system by analysing $Network_1$. For example we can estimate the expected number of clashes by investigating the various probabilities that the two senders are both in the state *sent*, and t is set to a specific integer. To do this we model check the property "$[\neg(s_a = sent \wedge s_b = sent)\ \mathcal{U}\ (t \leq n)]$". The expected number of clashes may then be derived from these probabilities.

5.3 Probabilistic backoff

Our final refinement is to introduce the randomised backoff procedure in each sender. When each process detects that it is in collision with another — again the details of how they do this have been suppressed — it sets its "backoff counter" to some random number and then counts down. As there is a good chance that the two backoff counters will be set to different values, the implication is that one of the senders will try to re-send at a time when the other is still "counting down", thus breaking the deadlock.

$$RandSender_A \mathrel{\hat{=}} \left(\begin{array}{l} \textbf{var } s_a\colon \{wait, sent, recv\},\ bc_a\colon \{0 \ldots N\} \\ \textbf{initially } s_a := wait;\ bc_a := 0 \\ send_a : (s_a = wait) \rightarrow s_a := sent \\ ack_a : \ (s_a = sent) \rightarrow s_a := recv \\ clash : \ (bc_a = 0 \wedge s_a = wait) \rightarrow \mathit{flip}(bc_a) \\ tick : \ (bc_a > 0) \rightarrow bc_a := bc_a - 1 \\ tick : \ (bc_a = 0 \wedge s_a \neq wait) \rightarrow \mathsf{skip} \end{array} \right) \quad Channel'_{AB} \mathrel{\hat{=}} \left(\begin{array}{l} \textbf{var } t\colon \{0, \ldots, T\} \\ \textbf{initially } t := 0 \\ clash : t := t + 1 \end{array} \right) ,$$

where $\mathit{flip}(x)$ sets the value of variable x to be n with probability $1/r^{n+1}$, if $n < N$, and to N with probability $1 - (1/r)^N$.

Fig. 6. A sending station with a backoff procedure, and channel to count clashes.

The new system,

$$Network_2 \quad \hat{=} \quad RandSender_A \ ||_{\mathcal{A}'} \ Receiver_{AB} \ ||_{\mathcal{A}'} \ RandSender_B \ ,$$

is the parallel composition of the three components, this time synchronising on the all events given by $\mathcal{A}' \mathrel{\hat{=}} \mathcal{A} \cup \{clash, \quad tick\}$; we show that it is a refinement of $Network_1$.

Lemma 5.

$$Network_1 \quad \preceq_g \quad Network_2 \backslash \{tick\} \ ,$$

where g represents s_a, t and r.

Proof. We use rep $\hat{=}$ if $(c = clash$ then $(bc_a, bc_b :\in (bc_a = bc_b))$ else $(bc_a, bc_b :\in (bc_a \neq bc_b))$*, and the argument now follows as for Lem. 4, noting that the probabilities when $bc_a = bc_b$ after the execution of $\mathit{flip}(bc_a)$ and $\mathit{flip}(bc_a)$ must add up to p, a condition which induces a relation between p and r, i.e. $p = \sum_{i<N} r^{2(i+1)} \times (1-r)^2 + (1-r)^{2N}$. Finally we must use Def. 2 to separate the single actions representing the synchronisation of clash and tick into actions residing in $RandSender_A$ and $RandSender_B$.*

We end by noting that the facts gathered about $Network_0$ and $Network_1$ — that no acknowledgement may be sent before a send event, and the expected number of times that the senders try to send at the same time — still apply to $Network_2$, without the need for any further checks.

5.4 Experimental results

Our experiments with PRISM implementations of examples of the above small systems demonstrate the large increase in model size with later refinements, thus abstraction of this kind may be thought of as easing state space explosion. For example with N set to 5 and T set to 100, the PRISM model for $Network_1$ consisted of 1608 states and 2009 transitions, whilst the PRISM model for $Network_2$ consisted of $20,008$ states and $39,611$ transitions. In a more realistic case study in which we include timing constraints we expect the increase to be greater still.

6 Comparisons with other work and research directions

Our contribution can be viewed as the first step towards a fully integrated system in which developers may use automation to explore (to some extent) the quantitative performance of their design decisions at early stages in their development. Just as we have come to expect from a development based on qualitative properties alone, the formal refinement relation ensures that the integrity of performance analysis is preserved as the system matures. Whilst our results show that model checking does not need to be used again once the specified performance property has been verified of an abstract system, the refinement process may continue in principle all the way to the code level. More experience is needed to decide when best to use the model checking results within a development.

Research in the modelling and evaluation of system performance has primarily focussed on process algebra such as PEPA [11] and TIPP [10] and Stochastic Petri nets [17]. These approaches allow specification of delays, which are realised by exponential distributions in a Markov Process simulation. Equivalence between processes — where it is available — is via various weak and strong bisumulations, and indeed our use of *rep* is similar to Hillston's propert-preserving weak bisimulations. However in none of these cases can the performance properties be transferred *directly* to the code level, in a structured manner as they can with the specification/refinement paradigm presented here.

Other approaches using probabilistic action systems include Sere and Troubitsyna [26] and Hallerstede [9]. In the former case hiding is not available, whilst the latter models an action system as a Markov Decision Process with specified rewards. Mechanised tool support automates the selection of a refinement which guarantees the optimal cost with respect to the "long-run average". Although the system allows refinement, equality between action systems is necessary to preserve the integrity of the properties in more refined systems. In contrast the approach we have taken is closer to the normal refinement style, in which the

value of minimal properties increases up the refinement order; moreover maximal properties can also be ordered via refinement to prove worst case upper bounds, and are investigated elsewhere [5]. We note that our theory founded on expectations gives access to probabilities, expected times [20] and long-run averages [18], though more work is required for tools to compute the latter property directly.

There are many extensions of process algebras to include probability [13, 14], including the use of simulations [16, 25]. However, these approaches tend to deal directly with operational features of systems, compared to this work whose domain-theoretical basis exposes the (standard) mathematical structures underlying the probabilistic and nondeterministic features of the semantics. In particular it establishes straightforwardly the definitions of the associated quantitative transformer logic, allowing us to address the practical goal of preservation of system properties.

Stepwise developments of wireless-like protocols have been carried out by Stoelinga [27].

Future work will explore the use of annotating programs with delays and other performance criteria, and developing tool support for the proof of program refinements.

A Appendix

We begin with a general fact about (probabilistic) action systems.

Lemma 6. *For any program Prog, the following equality holds:*

$$\mathsf{it}\ \mathit{Prog}\ \mathsf{ti} = \sup_{n\to\infty} \mathit{Prog}_n\ ,$$

where $\mathit{Prog}_0 \ \hat{=}\ \mathsf{abort}$, *and for* $n > 0$, $\mathit{Prog}_{n+1} \ \hat{=}\ \mathsf{skip} \,[\!]\, \mathit{Prog}; \mathit{Prog}_n$.

Proof. This follows from the definition of iteration, and the standard fact that the least fixed point is the limit of iterating from the bottom element in the program space, which in this case is abort.

A.1 Proof of Lem. 2

We reason as follows, referring to Def. 7 for properties of our assumptions about $\preceq_g$.

$$
\begin{array}{lll}
 & \{\!|t|\!\}^{\mathcal{V}}_{P}\ ;\ \mathsf{skip}_g & \\
\sqsubseteq & P_i\ ;\ \mathsf{it}\ P_\tau\ \mathsf{ti}\ ;\ \|t\|^{\mathcal{V}}_{P}\ ;\ \mathsf{skip}_g & \text{Def. 6} \\
\sqsubseteq & P_i\ ;\ \mathsf{it}\ P_\tau\ \mathsf{ti}\ ;\ \|t\|^{\mathcal{V}}_{P}\ ;\ \mathit{rep}\ ;\ \mathsf{skip}_g & \text{Def. 7 (4)} \\
\sqsubseteq & P_i\ ;\ \mathsf{it}\ P_\tau\ \mathsf{ti}\ ;\ \mathit{rep}\ ;\ \|t\|^{\mathcal{V}}_{Q}\ ;\ \mathsf{skip}_g & \|t\|^{\mathcal{V}}_{P}\ ;\ \mathit{rep} \sqsubseteq \mathit{rep}\ ;\ \|t\|^{\mathcal{V}}_{Q}\text{, see below} \\
\sqsubseteq & P_i\ ;\ \mathit{rep}\ ;\ \mathsf{it}\ Q_\tau\ \mathsf{ti}\ ;\ \|t\|^{\mathcal{V}}_{Q}\ ;\ \mathsf{skip}_g & \text{Def. 7 (3)} \\
\sqsubseteq & Q_i\ ;\ \mathsf{it}\ Q_\tau\ \mathsf{ti}\ ;\ \|t\|^{\mathcal{V}}_{Q}\ ;\ \mathsf{skip}_g & \text{Def. 7 (1)} \\
= & \{\!|t|\!\}^{\mathcal{V}}_{Q}\ ;\ \mathsf{skip}_g\ . & \text{Def. 6}
\end{array}
$$

For the "see below" part we reason by structural induction on tests.

Case: $t = \{G\}$.

$$
\begin{array}{lll}
 & \|\{G\}\|^{\mathcal{V}}_{P} \ ; \ rep & \\
= & \{\mathcal{V}.G\} \ ; \ rep & \text{Def. 6} \\
= & rep \ ; \ \{\mathcal{V}.G\} & rep \text{ does not affect global variables; it is terminating and strict} \\
= & rep \ ; \ \|\{G\}\|^{\mathcal{V}}_{Q} & \text{Def. 6}
\end{array}
$$

Case: $t = [G]$. As for $t = \{G\}$.

Case: $t = [\![\ ]\!]_{a:\,K} a$.

$$
\begin{array}{lll}
 & \|\,[\![\ ]\!]_{a \in K} a\|^{\mathcal{V}}_{P} \ ; \ rep & \\
= & [\![\ ]\!]_{a \in \mathcal{V}.K}(\mathsf{it}\ P_\tau\ \mathsf{ti}\ ;\ P_a\ ;\ \mathsf{it}\ P_\tau\ \mathsf{ti}) \ ; \ rep & \text{Def. 6} \\
\sqsubseteq & [\![\ ]\!]_{a \in \mathcal{V}.K} rep \ ; (\mathsf{it}\ Q_\tau\ \mathsf{ti}\ ;\ Q_a\ ;\ \mathsf{it}\ Q_\tau\ \mathsf{ti}) & \text{see below} \\
= & rep \ ; \ [\![\ ]\!]_{a \in \mathcal{V}.K}(\mathsf{it}\ Q_\tau\ \mathsf{ti}\ ;\ Q_a\ ;\ \mathsf{it}\ Q_\tau\ \mathsf{ti}) & rep \text{ is standard} \\
= & rep \ ; \ \|\,[\![\ ]\!]_{a \in K} a\|^{\mathcal{V}}_{Q} & \text{Def. 6}
\end{array}
$$

For the "see below", we reason as follows:

$$
\begin{array}{lll}
 & \mathsf{it}\ P_\tau\ \mathsf{ti}\ ;\ P_a\ ;\ \mathsf{it}\ P_\tau\ \mathsf{ti}\ ;\ rep & \\
\sqsubseteq & \mathsf{it}\ P_\tau\ \mathsf{ti}\ ;\ P_a\ ;\ rep\ ;\ \mathsf{it}\ Q_\tau\ \mathsf{ti} & \text{Def. 7 (3)} \\
\sqsubseteq & \mathsf{it}\ P_\tau\ \mathsf{ti}\ rep\ ;\ Q_a\ ;\ \mathsf{it}\ Q_\tau\ \mathsf{ti} & \text{Def. 7 (2)} \\
\sqsubseteq & rep\ ;\ \mathsf{it}\ Q_\tau\ \mathsf{ti}\ ;\ Q_a\ ;\ \mathsf{it}\ Q_\tau\ \mathsf{ti} & \text{Def. 7 (3)} \\
= & rep\ ;\ \|a\|^{\mathcal{V}}_{Q} & \text{Def. 6}
\end{array}
$$

Case: $t = t'; t''$.

$$
\begin{array}{lll}
 & \|t'; t''\|^{\mathcal{V}}_{P} \ ; \ rep & \\
= & \|t'\|^{\mathcal{V}}_{P}; \|t''\|^{\mathcal{V}}_{P} \ ; \ rep & \text{Def. 6} \\
\sqsubseteq & rep \ ; \ \|t'\|^{\mathcal{V}}_{Q} \ ; \ \|t''\|^{\mathcal{V}}_{Q} & \text{Structural induction twice} \\
= & rep \ ; \ \|t'; t''\|^{\mathcal{V}}_{P} & \text{Def. 6}
\end{array}
$$

Case: $t = \mathsf{it}\ t'\ \mathsf{ti}$.

$$
\begin{array}{lll}
 & \|\mathsf{it}\ t'\ \mathsf{ti}\|^{\mathcal{V}}_{P} \ ; \ rep & \\
= & \mathsf{it}\ \|t'\|^{\mathcal{V}}_{P}\ \mathsf{ti} \ ; \ rep & \text{Def. 6} \\
= & \sup_{n\to\infty}(Prog^{P}_{n}) \ ; \ rep & \text{Set } Prog^{P} \mathrel{\hat{=}} \|t'\|^{\mathcal{V}}_{P} \text{ in Lem. 6} \\
= & \sup_{n\to\infty}(Prog^{P}_{n} \ ; \ rep) & rep \text{ distributes through sup} \\
\sqsubseteq & \sup_{n\to\infty}(rep \ ; \ Prog^{Q}_{n}) & \text{Set } Prog^{Q} \mathrel{\hat{=}} \|t'\|^{\mathcal{V}}_{Q} \text{ and see below.} \\
\sqsubseteq & rep \ ; \ \sup_{n\to\infty} Prog^{Q}_{n} & rep \text{ is monotone} \\
= & rep \ ; \ \mathsf{it}\ \|t'\|^{\mathcal{V}}_{Q}\ \mathsf{ti} & \text{Def. 6} \\
= & rep; \|\mathsf{it}\ t'\ \mathsf{ti}\|^{\mathcal{V}}_{Q} \ . & \text{Lem. 6}
\end{array}
$$

For the final "see below", we reason, by induction on n that

$$Prog_n^P \ ; \ rep \quad \sqsubseteq \quad rep \ ; \ Prog_n^Q \ .$$

The base case ($n = 0$) follows since LHS is abort. For the inductive step we reason as follows:

$$\begin{array}{lll}
 & Prog_{n+1}^P \ ; \ rep & \\
= & (\mathsf{skip} \ \| \ Prog^P ; Prog_n^P) \ ; \ rep & \\
\sqsubseteq & (rep \ \| \ Prog^P ; rep ; Prog_n^Q) & \text{induction hypothesis} \\
\sqsubseteq & (rep \ \| \ rep ; Prog^Q ; Prog_n^Q) & \text{Structural induction: } (Prog^P ; rep \sqsubseteq rep ; Prog^Q) \\
= & (rep \ ; \ (\mathsf{skip} \ \| \ Prog^Q ; Prog_n^Q) & rep \text{ is standard} \\
= & rep \ ; \ Prog_{n+1}^Q \ . &
\end{array}$$

References

1. IEEE 802.11 standard. http://grouper.ieee.org/groups/802/11/main.html.
2. J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *First Conference on the B Method*, pages 169–190. Laboratoire LIANA, L'Institut Universitaire de Technologie (IUT) de Nantes, November 1996.
3. R.-J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
4. M.J. Butler and C.C. Morgan. Action systems, unbounded nondeterminism and infinite traces. *Formal Aspects of Computing*, 7(1):37–53, 1995.
5. O. Celiku and A. McIver. Cost-based analysis of probabilistic programs mechanised in HOL. *Nordic Journal of Computing*, 2004.
6. M. de Nicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34, 1984.
7. P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–62, 1991. Reprinted in [23].
8. G.R. Grimmett and D. Welsh. *Probability: an Introduction*. Oxford Science Publications, 1986.
9. S. Hallerstede and M. Butler. Performance analysis of probabilistic action systems. 2005.
10. H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the tipp tool. *Performance Evaluation*, 39:5–35, 2000.
11. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
12. C.A.R. Hoare, Jifeng He, and J.W. Sanders. Prespecification in data refinement. *Inf. Proc. Lett.*, 25(2):71–6, May 1987.

13. B. Jonsson and K.G. Larsen. Specification and refinement of probabilistic processes. In *Proc. 6th Conf. LICS*, 1991.
14. B. Jonsson, K.G. Larsen, and Wang Yi. Probabilistic extensions of process algebras. *Handbook of Process Algebras*, (1):685–710, 2001.
15. M. Kwiatkowska, G. Norman, and D.Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. 2002. Accepted for TACAS 2002.
16. K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Proc. 16th ACM Symp. PoPL*, 1989.
17. M. Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with generalised stochastic petri nets*. Wiley, New York, 1995.
18. A.K. McIver. A generalisation of stationary distributions, and probabilistic program algebra. In Stephen Brookes and Michael Mislove, editors, *Electronic Notes in Theo. Comp. Sci.*, volume 45. Elsevier, 2001.
19. A.K. McIver and C.C. Morgan. Results on the quantitative μ-calculus $qM\mu$. To appear in *ACM TOCL*, 2004.
20. Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Technical Monographs in Computer Science. Springer Verlag, New York, 2004.
21. C.C. Morgan. Of probabilistic Wp and CSP. *25 years of CSP*.
22. C.C. Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994.
23. C.C. Morgan and T.N. Vickers, editors. *On the Refinement Calculus*. FACIT Series in Computer Science. Springer Verlag, Berlin, 1994.
24. PRISM. Probabilistic symbolic model checker. `www.cs.bham.ac.uk/~dxp/prism`.
25. Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, 1995.
26. Kaisa Sere and Elena Troubitsyna. Probabilities in action systems. In *Proc. of the 8th Nordic Workshop on Programming Theory*, 1996.
27. Mariëlle Stoelinga and Frits Vaandrager. Root contention in IEEE 1394. *Lecture Notes in Computer Science*, 1601:53–74, 1999.