Proofs and refutations for probabilistic systems

AK McIver[1]*, CC Morgan[2]*, and C Gonzalia[1]*

[1] Dept. Computer Science, Macquarie University, NSW 2109 Australia
[2] School of Comp. Sci. and Eng., Univ. New South Wales, NSW 2052 Australia

Abstract. We consider the issue of finding and presenting counterexamples to a claim "this *spec* is implemented by that *imp*", that is $spec \sqsubseteq imp$ (refinement), in the context of *probabilistic* systems: using a geometric interpretation of the probabilistic/demonic semantic domain we are able to encode both refinement success and refinement failure as linear satisfaction problems, which can then be analysed automatically by an SMT solver. This allows the automatic discovery, and then presentation, of counterexamples in independently and efficiently checkable form.
In many cases the counterexamples can subsequently be converted into "source level" hints for the verifier.
Keywords: Probabilistic systems, counterexamples, quantitative program logic, refinement, constraint solving.

1 Introduction

One of the strengths of standard model checking is its ability to produce counterexamples as concrete evidence that an implementation or model of a system fails to meet its specification. Moreover in some cases the counterexample can aid debugging by pointing to possible causes of the problem [2].

Unfortunately, with *probabilistic* model checking there is not yet an accepted definition for what a counterexample should be, nor is there a tradition for using counterexamples for debugging. In particular, a single computation path or trace is not normally sufficient counterevidence: it is more likely to be a cumulative trend over many traces that leads to suspect behaviour [8], suggesting a probabilistic computation tree as a candidate for a counterexample. A *tree* however cannot easily be presented as a cogent summary of the possible faults, nor does it indicate how to correct them.

The theme of this paper is a novel approach to presenting counterexamples in the context of probabilistic systems, and how it can be used in practice. Our proposal is guided by the following principles which, we believe, are qualities any good counterexample should possess:

P1 A counterexample should produce a *certificate* of failure that is easy to check, independently of the tool that found it; moreover,
P2 As far as possible the certificate should relate directly to the program text or system model; and finally,

* We acknowledge the support of the Australian Research Council Grant DP0558212.

P3 It should direct the verifier to the possible causes of the problem.

In system verification there is a great variety of behaviours. Whilst identifying the "bad behaviours" amongst the complete set might be hard in the first instance, once observed they should be immediately recognisable as such — in this context that means the counterexample should be checkable with minimum effort. This suggests *P1* and *P2*. Principle *P3* is included as it has the potential to be extremely useful as a debugging tool.

The current proposals [7, 8] for counterexamples in probabilistic systems satisfy none of these properties, largely because they are based on probabilistic trace semantics — whilst (sets of) traces do provide evidence, they are neither easily verifiable, nor can they be directly related to the original system model.

Our approach is based on the refinement style of specification exemplified by the refinement calculus [14, 1] extended to include probability [15, 12]. In this style a specification *spec* is a heavily abstracted system, which is so simple as to be "obviously correct," whereas an implementation *imp* is more detailed, including distributed features or complicated program-code intended to realise some optimisation. Once a set of observable behaviours is agreed on, one writes $\mathit{spec} \sqsubseteq \mathit{imp}$, that *spec is refined by imp*, to mean that all possible behaviours of *imp* are included in those of *spec*.

Our main concern in this paper is when such a hypothesised refinement fails in the probabilistic case. We consider the problems of what constitutes good evidence to refute a refinement, and how can it be used to help the verifier solve the problem, possibly by changing one of *spec* or *imp*. (The former is changed when the counterexample reveals that *spec* is too demanding, and the latter when *imp* contains genuinely incorrect behaviours.)

Our specific contributions in this paper are thus as follows.

1. A description (Sec. 4.3) of how a counterexample to a proposed probabilistic refinement may be encoded as the failure to satisfy a quantitative property; it is a term in the quantitative program logic of Morgan and McIver [12];
2. An implemented procedure (Sec. 4) to compute the semantics of a small probabilistic programming language *pGCL*, and an arithmetic solver, which together compute a certificate in the case that refinement fails, showing adherence to Principles *P1* and *P2*;
3. A method (Sec. 4.4) to use the certificate to produce a suspect schedule, in distributed systems for example, thus fulfilling Principle *P3*.

In Sec. 2 we provide a summary of the overall approach, with later sections elaborating the details of the ideas introduced there.

We assume a (finite) state space S; we write $\mathbb{D}X$ for the set of (discrete) distributions over X, namely the set of 1-summing functions $X \rightarrow [0, 1]$; given a set K we write $\mathbb{P}K$ for its power set. Given two distributions d, d' and scalar $0 \leq p \leq 1$, we write $d\,_p\!\oplus d'$ for the distribution $p \times d + (1-p) \times d'$. We use an explicit dot for left-associating function application; thus $(f(x))(y)$ becomes $f.x.y$.

2 On refinement, and checking for it: an introduction

Our basic model for operational-style denotations of sequential demonic programs *without probability* is $S \leftrightarrow (S \cup \{\bot\})$ or equivalently $S \to \mathbb{P}S_\bot$, in which (in the latter form) some initial state $s \in S$ is taken by (program denotation) $r \in S \to \mathbb{P}S_\bot$ to any one of the final states $s' \in r.s$. A common convention is that if $\bot \in r.s$ then so also are all $s' \in r.s$ — nontermination (final state the "improper" $\bot$) is catastrophic.

The reason for that last, so-called "fluffing-up" convention (aside from its being generated automatically by the Smyth power-domain over the flat order on $S_\bot$) is that it makes the refinement relation between programs very simple: it is subset, lifted pointwise. Thus we say that $r_1 \sqsubseteq r_2$, i.e. Program r_1 *is refined by* Program r_2, just when for all states s we have $r_1.s \supseteq r_2.s$. The fluffing-up means that the same $\supseteq$-convention that refines by reducing nondeterminism also refines by converting improper $\bot$ (nontermination) into proper behaviour.

Except for nontermination, *result sets* given by $r.s$ are fairly small when the program r is almost deterministic. In that case, from a fixed initial state s° the question of whether $r_1 \sqsubseteq r_2$ can feasibly be established by examining every final state $s' \in r_2.s^\circ$ and checking that also $s' \in r_1.s^\circ$.

Once probability is added, at first things look grim (details in Def. 1 below): there can be non-denumerably many output *distributions* for non-looping programs over a finite, even small, state space: this is because of the "convexity" convention (analogous to fluffing-up) that pure demonic choice $\sqcap$ can be refined by any probabilistic choice ${}_p\oplus$ whatever, i.e. for any $0 \le p \le 1$. The reason for convexity is to allow, again, refinement via $\supseteq$ in all cases; but its effect is that even the simple program $s := A \sqcap B$ has as result set all distributions $\{\overline{A}\ {}_p\oplus\ \overline{B} \mid 0 \le p \le 1\}$, where in the comprehension we write $\overline{A}, \overline{B}$ for the point distributions at A, B.[3] Thus if r_2 is being compared for refinement against some r_1, it seems there are uncountably many final distributions to consider.

Luckily the actual situation is not grim at all: those result sets, big though they might be, are convex closures of a finite number of distributions, provided S is finite — and even if the program contains loops. (A set D of distributions is convex closed if whenever $d, d' \in D$ then so is $d\ {}_p\oplus\ d'$ for any $0 \le p \le 1$.) Writing $\lceil\cdot\rceil$ for this closure we are saying that in fact $r.s \in \mathbb{P}\mathbb{D}S_\bot$ is equal to $\lceil D \rceil$ for some finite set of distributions D (depending on r and s). And so by elementary properties of convexity, to check $r_1 \sqsubseteq r_2$ for such programs we need only examine for each s° the (small) sets $D_{1,2}$ of distributions from which $r_{1,2}.s^\circ$ are generated.

This amounts to taking each result distribution $d' \in D_2$ and checking whether that d' is a convex combination of the finitely many distributions in D_1, which –crucially– can be formulated as a linear-constraint problem; and it is not so much worse than in the non-probabilistic case. Even better, however, is that if in fact $d' \notin \lceil D_1 \rceil$, then it is possible to find a certificate for that: because of the *Separating Hyperplane Lemma*, there must be some plane in the Euclidean

[3] Point distributions have probability one at some state and (hence) zero at all others.

space[4] containing $D_{1,2}$ with $\lceil D_1 \rceil$ strictly on one side of it and the inconvenient $d' \in \lceil D_2 \rceil$ (non-strictly) on the other. Finding that plane's normal (a tuple of reals that describes the plane's orientation) is *also* a linear-constraint problem, and can be done with the same engine that attempted to show $d' \in \lceil D_1 \rceil$ (but in fact found the opposite).

Thus the overall strategy –and the theme of this paper– is to calculate $D_{1,2}$ for some initial state s° and probabilistic nondeterministic programs given as $r_{1,2} \in S \rightarrow \mathbb{PD}S_\perp$, and then for each $d' \in D_2$ to attempt to establish $d' \in \lceil D_1 \rceil$. If that succeeds for all such d''s, declare $r_1 \sqsubseteq r_2$ at s°; but if it fails at some d', then produce a certificate (hyperplane normal) for that failure.

As we will see, that certificate can then be used to identify, in a sense "highlight," the key "decision points" through the program r_2 that together caused the refinement failure — and there is our probabilistic counterexample that can be presented to the public and checked –by them independently– using the certificate from the hyperplane.

3 Probabilistic refinement in detail

3.1 Definition of refinement

The transition-style semantics now widely accepted for probabilistic sequential systems models a probabilistic program as a function from initial state to (appropriately structured) sets of distributions over (final) states: each distribution describes the frequency aspects of a *probabilistic choice*, and a *set* of them (if not singleton) represents *demonic nondeterminism.*

Starting with a flat domain $S_\perp \mathrel{\hat{=}} S \cup \{\perp\}$, with $\perp \sqsubset s$ for all proper states s, we construct $\mathbb{D}S_\perp$, the *discrete distributions over* $S_\perp$ and give it an (flat-induced) order so that for $d, d' \in \mathbb{D}S_\perp$ we have $d \sqsubseteq d'$ just when $d.s \leq d'.s$ for all *proper* s. (Note that $d.\perp > d'.\perp$ might occur to compensate.)

Then a set $D \subseteq \mathbb{D}S_\perp$ is said to be *up-closed* if whenever $d \in D$ and $d \sqsubseteq d'$ then also $d' \in D$; it is *convex* if whenever $d, d' \in D$, so too is $d \;{}_p\!\oplus\; d'$ for any $0 \leq p \leq 1$; and finally it is *Cauchy closed* if it contains all its limit points with respect to the Euclidean metric. [4 again]

Definition 1. *[15, 9] The space of (denotations of) probabilistic programs is given by* $(\mathbb{C}S, \sqsubseteq)$ *where* $\mathbb{C}S$ *is the set of functions from* S *to* $\mathbb{PD}S_\perp$*, restricted to subsets which are* convex, up- *and* Cauchy *closed. The order between programs is induced pointwise (again) so that* $r \sqsubseteq r'$ *iff* $(\forall s \colon S \cdot r.s \supseteq r'.s)$.

The refinement relation defines when two programs exhibit the same or similar overall behaviour — from Def. 1 we see that a program is more refined by another whenever the extent of nondeterminism is reduced.

We use a small language *pGCL* that generalises Dijkstra's guarded-command language [5] by adding probabilistic choice (and retaining demonic choice); in

[4] See Sec. 3.6.

identity	$[\![\mathsf{skip}]\!].s$	$\mathrel{\hat{=}}$	$\{\overline{s}\}$
assignment	$[\![x := a]\!].s$	$\mathrel{\hat{=}}$	$\{\overline{s[x \mapsto a]}\}$
composition	$[\![P; P']\!].s$	$\mathrel{\hat{=}}$	$\{\sum_{s' : S} d.s' \times f'.s' \mid d \in [\![P]\!].s; [\![P']\!] \sqsubseteq f'\}$

where $f' \in S \to \mathbb{D}S_\perp$ and in general $r' \sqsubseteq f'$ means $r'.s \ni f'.s$ for all s.

choice	$[\![\mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ P'\ \mathsf{fi}]\!].s$	$\mathrel{\hat{=}}$	if $B.s$ then $[\![P]\!].s$ else $[\![P']\!].s$
probability	$[\![P\ {}_p{\oplus}\ P']\!].s$	$\mathrel{\hat{=}}$	$\{d\ {}_p{\oplus}\ d' \mid d \in [\![P]\!].s; d' \in [\![P']\!].s\}$
nondeterminism	$[\![P \sqcap P']\!].s$	$\mathrel{\hat{=}}$	$\lceil\, [\![P]\!].s \cup [\![P']\!].s \,\rceil$,

where in general $\lceil D \rceil$ is the up-, convex- and Cauchy closure of D.

Iteration is defined via a least fixed-point; but we do not use iteration in this paper.

Fig. 1. Relational-style semantics of probabilistic programs [12].

Fig. 1 we set out how its semantics in the style of Def. 1. Programs without probability behave as usual; programs with probability, but no nondeterminism, abide by classical probability theory; but programs containing both probability *and* nondeterminism can exhibit highly skewed –and confusing– probabilistic behaviour.

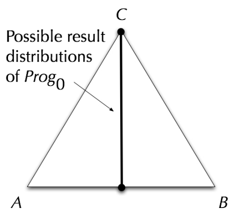

Fig. 2. Picture of *Prog*$_0$'s results

Each point in a triangle defines a discrete distribution over its vertices, here $\{A, B, C\}$, their unique linear combination that gives that point. Since *Prog*$_0$'s (set of) points is a strict subset of *Prog*$_1$'s points, we have *Prog*$_1 \sqsubset$ *Prog*$_0$ and hence also *Prog*$_0 \not\sqsubseteq$ *Prog*$_1$.

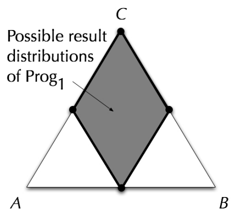

Fig. 3. Picture of *Prog*$_1$'s results

Figs. 2 and 3: *Distribution triangles* depict convex result-sets.

3.2 Example; and difficulty with counterexamples

To illustrate probabilistic refinement, and the difficulties with counterexamples, we consider the two programs below [12, App. A]. Checking *Prog*$_0$'s text suggests that it establishes $s{=}A$ and $s{=}B$ with equal probabilities; and those probabilities could be as high as 0.5 each (if the outer $\sqcap$ resolves always to the left) or as low as zero (if the $\sqcap$ resolves always to the right). Probabilities in-between (but still equal to each other) result from intermediate behaviours of the $\sqcap$.

Checking *Prog*$_1$ however suggests more general behaviour. For example, consider the "thought experiment" where we execute *Prog*$_0$ many times, and keep a record of the results: we expect to see a strong correlation between the number of A's and B's. However with *Prog*$_1$ we cannot rely on an A, B-correlation, as instead it might correlate B, C while ignoring A altogether.[5]

$$\mathit{Prog}_0 \quad \hat{=} \quad (s{:=}\ A\ {}_{0.5}{\oplus}\ s{:=}\ B)\ \sqcap\ s{:=}\ C \tag{1}$$
$$\mathit{Prog}_1 \quad \hat{=} \quad (s{:=}\ A \sqcap s{:=}\ C)\ {}_{0.5}{\oplus}\ (s{:=}\ B \sqcap s{:=}\ C) \tag{2}$$

Figures 2,3 depict the relation between *Prog*$_0$ and *Prog*$_1$ according to the semantics at Def. 1, in particular that they seem to be different. But it is not easy to see this experimentally via counterexample: what concrete property can we use to observe the difference? Indeed even if we tabulate, for the two programs, both the maximum and minimum probabilities of all 6 non-trivial result-sets, we get in Fig. 4 the same results *for both programs.*

Allowed final value(s) of s	A	B	C	A, B	B, C	C, A
Maximum possible probability	1/2	1/2	1	1	1	1
Minimum possible probability	0	0	0	0	1/2	1/2

The table illustrates the maximum and minimum probabilities for *Prog*$_0$ and *Prog*$_1$ with respect to all non-trivial choices of allowed outcome: the programs are not distinguishable this way. But in a larger context, they are: the composite programs

$$\begin{array}{rl} & \mathit{Prog}_0;\ \ \mathsf{if}\ s{=}C\ \mathsf{then}\ (s{:=}\ A\ {}_{0.5}{\oplus}\ s{:=}\ B)\ \mathsf{fi} \\ \text{and} & \mathit{Prog}_1;\ \ \mathsf{if}\ s{=}C\ \mathsf{then}\ (s{:=}\ A\ {}_{0.5}{\oplus}\ s{:=}\ B)\ \mathsf{fi} \end{array}$$

are distinguished by the test $s = A$.
This is a failure of compositionality for such (limited) tests [12, App. A.1].

Fig. 4. Maximum and minimum probabilities.

The fallback position, that perhaps *Prog*$_0$ and *Prog*$_1$ *are* "observably" equal at this level of abstraction, is not tenable either — for we can define a *context* in which such simple tabulations *do* reveal the difference. Define the program *Prog*$_2$ to be the conditional $\mathsf{if}\ s{=}C\ \mathsf{then}\ (s{:=}\ A\ {}_{0.5}{\oplus}\ s{:=}\ B)\ \mathsf{fi}$, and compare *Prog*$_0$; *Prog*$_2$ with *Prog*$_1$; *Prog*$_2$. The former establishes $s{=}A$ with probability 1/2; the latter however can produce $s{=}A$ with a probability as low as 1/4.[5 again]

[5] If the ${}_{0.5}{\oplus}$ goes left, take the $\sqcap$ right — and vice versa.

identity	$\mathsf{wp.skip}.expt$	$\hat{=}$	$expt$
assignment	$\mathsf{wp}.(x := E).expt$	$\hat{=}$	$expt[x := E]$
composition	$\mathsf{wp}.(P; P').expt$	$\hat{=}$	$\mathsf{wp}.P.(\mathsf{wp}.P'.expt)$
choice	$\mathsf{wp}.(\mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ P'\ \mathsf{fi}).expt.s$		
		$\hat{=}$	$\mathsf{wp}.P.expt.s$ if $B.s$ else $\mathsf{wp}.P'.expt.s$
probability	$\mathsf{wp}.(P\ {}_p\!\oplus P').expt$	$\hat{=}$	$p \times \mathsf{wp}.P.expt + (1-p) \times \mathsf{wp}.P'.expt$
nondeterminism	$\mathsf{wp}.(P \sqcap P').expt$	$\hat{=}$	$\mathsf{wp}.P.expt$ **min** $\mathsf{wp}.P'.expt$

The expression *expt* is of non-negative real type over the program variables. As earlier, iteration is given in the usual way via fixed point; but we do not treat iteration here.

Fig. 5. Structural definitions of wp [15, 12].

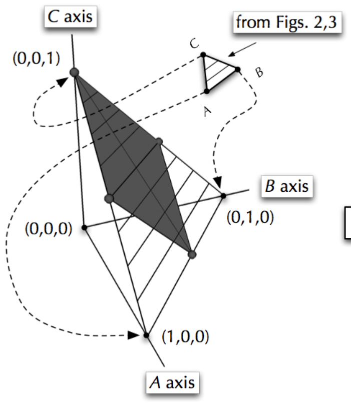

Fig. 6. Position the "distribution triangle" in 3-space, on the base of the non-negative $A+B+C \leq 1$ tetrahedron...

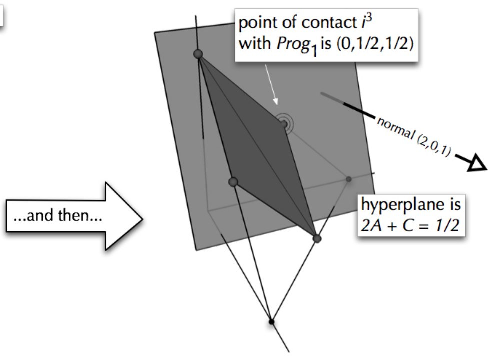

Fig. 7. . . . approach from below, with a hyperplane of normal (2,0,1), until a point in some result set is "touched."

The distribution-triangle of Figs. 2,3 becomes the base $A+B+C = 1$ of a tetrahedron in the upwards octant of Euclidean 3-space; a distribution over $\{A, B, C\}$ is now simply a point with the discrete probabilities as its A, B, C co-ordinates.
The random variable defined $(A, B, C) \mapsto (2, 0, 1)$ is represented by an e-indexed family of hyperplanes $2A+C = e$ all having the same normal $(2, 0, 1)$. The minimum expected value of that random variable over *any* set of distributions is the least e for which the representing hyperplane touches the set. For $Prog_1$'s distributions in particular, that value is $1/2$ (the plane shown in Fig. 7); for $Prog_0$ the e would be 1 (touching in fact at all the points in $Prog_0$'s line, that plane not shown).
The fact that the e's for $Prog_0$ and $Prog_1$ are different, for some normal, is what distinguishes the two programs; and, given any normal, the program logic of Fig. 5 can deliver the corresponding e directly from the source text of the program.
The "only" problem is to find that distinguishing normal.

Figs. 6 and 7: Distributions in 3-space, and touching hyperplanes.

3.3 Expected values of random variables certify counterexamples

We are rescued from the difficulties of Fig. 4 by the fact that Prog_0 and Prog_1 can after all be distinguished statically (rather than via lengthy simulations and statistical tests, as suggested by the above "thought experiment") provided we base our analysis on *random variables* rather than pure probabilities, i.e. *functions* over final states (to reals) rather than simple *sets* of final states.[6]

Rather than ask "What is the minimum guaranteed probability of achieving a given *postcondition* on the final states?" (precisely what was shown above to be non-compositional), we ask "What is the minimum guaranteed expected value of a given *random variable* over the final states?"

In our example above, a distinguishing random variable is e.g. the function $(A, B, C) \mapsto (2, 0, 1)$, giving minimum (in fact guaranteed) expected value 1 for Prog_0 but only 1/2 for Prog_1 (from all initial states, for these programs).

3.4 A logic of expectation transformers

The minimum expected values, explained informally in Sec. 3.3, can be found *at the source level* using a quantitative programming logic that generalises Dijkstra's predicate-transformer semantics [5].[7] We call it a logic of *expectation transformers.*

Definition 2. *Random variables (functions of type $\mathbb{E}S \mathrel{\hat{=}} S \to \mathbb{R}_{\geq 0}$) are written in the logic as non-negative real-valued expressions over the program variables. They are ordered by pointwise $\geq$. The expectation-transformer denotation of the logic is then $(\mathcal{T}S, \sqsubseteq)$, where $\mathcal{T}S \mathrel{\hat{=}} \mathbb{E}S \to \mathbb{E}S$, and $t \sqsubseteq t'$ iff $(\forall e \colon \mathbb{E}S \cdot t.e \leq t'.e)$.*

With this apparatus we present in Fig. 5 the expectation-transformer logic for *pGCL*; it corresponds to our earlier "set-of distribution" semantics of Fig. 1 in the same way as classical predicate transformers correspond to classical relational semantics.

3.5 Equivalence of relational- and transformer semantics

Our two definitions Def. 1 and Def. 2 give complementary views of programs' meaning; crucial for our work here is that those views are equivalent in the following sense:

Theorem 1. *[12, 15] Here (and briefly in Sec. 3.6), distinguish the two refinement orders by writing $\sqsubseteq_{\mathcal{R}}$ for the refinement order given in Def. 1; similarly write $\sqsubseteq_{\mathcal{T}}$ for the refinement order given in Def. 2. Then for any two pGCL programs P, P' we have $[\![P]\!] \sqsubseteq_{\mathcal{R}} [\![P']\!]$ iff $\mathsf{wp}.P \sqsubseteq_{\mathcal{T}} \mathsf{wp}.P'$.*

With Thm. 1 we can use just $\sqsubseteq$ for refinement between *pGCL* programs, in either semantics, which is why we do not usually distinguish them (thus dropping the subscripts $\mathcal{R}, \mathcal{T}$).

Next we see how a third, geometric view supports this equivalence.

[6] This startling innovation is due to Kozen [11]; but he did not treat demonic choice, and so our (non-)compositionality example was not accessible to him.

[7] This is again due to Kozen, again only in the deterministic case [11].

3.6 Distributions and random variables in Euclidean space

Fig. 6 shows how discrete distributions in $\mathbb{D}S_\perp$ can be embedded in $|S|$-dimensional Euclidean space: distribution d becomes a point whose s-coordinate is just $d.s$. (Representing $d.\perp$ is unnecessary, as it is determined by 1-summing.) *Arithmetically* convex sets of distributions become *geometrically* convex sets of points in this space.

Fig. 7 shows how a random variable in $\mathbb{E}S$ can be embedded in the same space: random variable f becomes a (family of) hyperplanes with a collective normal whose s-coordinate is just $f.s$. [8]

The crucial connection is that if the point representing d lies on a plane in the family f then the constant term of that particular plane is the expected value over the distribution d of the random variable that f's normal represents.

Def. 1 -style refinement remains the inclusion of one set of points (imp) wholly within another ($spec$), just as in our earlier Figs. 2,3.

Def. 2 -style refinement is equivalent, but can be formulated in terms of hyperplanes: take any (random-variable-representing-) hyperplane, and position it strictly below the positive octant in the space. (The results sets lie entirely in that octant.) Now move it up –along its normal– until it first touches a point (i.e. distribution) in one of the result sets. The constant term then gives exactly the wp for the program producing that first-touched distribution with respect to the random variable, written as an expectation in the logic of Fig. 5.

Then one program refines another just when for all such planes the less-refined program ($spec$) is always touched before the more-refined one (imp) is — because that means the constant term for $spec$ is always less that that for imp, whence the wp's are similarly ordered as they must be.

The two views justify Thm. 1 informally; we explain it in the contrapositive. If $spec \not\sqsubseteq_{\mathcal{R}} imp$ then for some initial state s° we have a distribution d' with $d' \in [\![imp]\!].s^\circ$ but $d' \notin [\![spec]\!].s^\circ$. Because $[\![spec]\!].s^\circ$ is convex, by the *Separating Hyperplane Lemma* there must be a plane separating d' from it in the sense that d' is in the plane but $[\![spec]\!].s^\circ$ lies strictly on one side of it.[9] Because our result sets are up-closed, the normal of that plane can be chosen non-negative; and thus if that plane approaches the positive octant from below, it will reach d' in $[\![imp]\!].s^\circ$ strictly before reaching any of $[\![spec]\!].s^\circ$, thus giving $spec \not\sqsubseteq_{\mathcal{T}} imp$.

The reverse direction is trivial: if $spec \not\sqsubseteq_{\mathcal{T}} imp$ then some plane reaches $[\![imp]\!].s^\circ$ before it reaches $[\![spec]\!].s^\circ$; hence we cannot have $[\![spec]\!].s^\circ \supseteq [\![imp]\!].s^\circ$; hence $spec \not\sqsubseteq_{\mathcal{R}} imp$.

[8] A hyperplane in N-space is a generalisation of a plane, in 3-space $ax+by+cz = e$. The tuple (a, b, c) is its *normal* and e is its *constant term*.

[9] The *SHP* Lemma states that any point not in a closed and bounded convex set can be *separated* from the set by a plane that has the point on one side and the set strictly on the other.

4 Proofs and refutations

With the above apparatus we address our main issue: given two *pGCL* programs *spec*, *imp* over some finite state space S, what computational methods can we use either to prove that $spec \sqsubseteq imp$, or to find –and present convincingly– a counterexample? We treat the two outcomes separately.

4.1 Calculating result sets

In order to prove refinement, i.e. $spec \sqsubseteq imp$, we must –in effect– investigate every possible outcome (distribution) of the implementation *imp* (element of its result set) and see whether it is also a possible outcome of the specification *spec* (is an element of that result set too). Because of the structure of these sets, that they are convex closures of a finite number of "vertex" distribution points,[10] it is enough to check each vertex of the implementation result set against the collection of vertices of the specification result set.

These sets are calculated in the same way (for *spec* and for *imp*), simply by "coding up" the relational semantics given in Fig. 1 in a suitable (functional) programming language. The principal data-type is *finite set of distributions*, with each distribution being in turn a suitably normalised real-valued function of the finite state space (representable thus as a simple tuple of reals).

We discuss *sequential composition* $S;T$ as an example. Components S and T separately will have been analysed to give structures of type *initial state to set of final distributions*; the composition is implemented by taking the generalised Cartesian product of the T structure –converting it to a set of functions from initial state to final distribution– and then linearly combining the outputs of each of those functions, varying over its initial-state input, using the coefficients given by the probabilities assigned to each state by the S structure in each of its output distributions separately. That gives a set of output distributions for each single output distribution of S; and the union is taken of all of those.

The number of result distributions generated by the program as a whole is determined by the number of syntactic nondeterministic choices and the size of the support of the probabilistic branching, and it is affected by the order in which these occur. For example a D-way demonic branch each of whose components is a P-way probabilistic branch will generate only D distributions (since each P-way branch is a single distribution). However the opposite, i.e. a P-way branch each of whose components is a D-way branch, will generate $|D|^{|P|}$ output distributions — because the effect of calculating those distributions for the whole program is simply to convert it to (the representation of) a normal form in which all nondeterministic branching occurs before any probabilistic branching.[11]

[10] Sufficient mathematical conditions for this are that either the state space is finite and "raw" nondeterminism $\sqcap$ is finite, with loops allowed, or that the program is finite, that is it has no loops. We do not know whether it holds for infinite state spaces with loops, or finite state spaces with general (non-tail) recursion.

[11] For example the program $(x{:=}\ \pm 1)\ {}_{1/3}{\oplus}\ (x{:=}\ \pm 2)$ normalises to
$(x{:=}\ 1\ {}_{1/3}{\oplus}\ 2) \sqcap (x{:=}\ 1\ {}_{1/3}{\oplus}\ {-2}) \sqcap (x{:=}\ {-1}\ {}_{1/3}{\oplus}\ 2) \sqcap (x{:=}\ {-1}\ {}_{1/3}{\oplus}\ {-2})$.

Suppose we have M sequentially composed components each one of which is an at most D-way demonic choice between alternatives each of which has at most P non-zero-probability alternatives. The computed results-set is determined by at most $D^{1+P+P^2+\cdots+P^{M-1}}$ vertices. Whilst this makes computing result distributions theoretically infeasible, in practice it is rarely the case that probabilistic and nondeterministic branching interleaves to produce this theoretical worst case.

4.2 Proving refinement

Now suppose our state-space is of finite size N; then distributions can be represented as as points within Euclidean N-space. The procedure outlined above will thus generate

- for *spec* some set $\boldsymbol{S} \mathrel{\hat{=}} \boldsymbol{s}^{1..K}$ of N-vectors, and
- for *imp* some (other) set $\boldsymbol{I} \mathrel{\hat{=}} \boldsymbol{i}^{1..L}$ of N-vectors.

In each case the actual "implied" sets of result distributions are the convex closures $\lceil \boldsymbol{S} \rceil$ and $\lceil \boldsymbol{I} \rceil$ and we are checking that $\lceil \boldsymbol{I} \rceil \subseteq \lceil \boldsymbol{S} \rceil$,

- equivalently that each $\boldsymbol{i}^l \in \lceil \boldsymbol{S} \rceil$,
- equivalently that each $\boldsymbol{i}^l = \boldsymbol{c}^l \cdot \boldsymbol{S}$ for some $\boldsymbol{c}^l$, where $(\cdot)$ is the matrix multiplication of the non-negative 1-summing row-vector $\boldsymbol{c}^l$ of length K and the K-row-by-N-column representation of the set $\boldsymbol{S}$ of distributions,
- equivalently for that l that this constraint set has a solution in scalars $c^l_{1..K}$:
  - $0 \leq c^l_k$ for $1 \leq k \leq K$ and $\sum_{1 \leq k \leq K} c^l_k = 1$;
  - $\boldsymbol{i}^l_n = \sum_{1 \leq k \leq K} c^l_k s^k_n$ for $1 \leq n \leq N$.

That last set of $K{+}1{+}N$ (in)equations (for each l) can be dealt with by a suitable satisfaction solver (Sec. 6). If they can be solved, then the refinement holds at that point $\boldsymbol{i}^l$; and if that happens for all $1 \leq l \leq L$ then the refinement holds generally. If not, then we have found an "inconvenient" implementation behaviour $\boldsymbol{i}^l$, and the refinement fails.

We say that *the certificate to support a proposed refinement* is the $K{\times}L$ matrix $\boldsymbol{c}$ of scalars that gives the appropriate K-wise interpolation of $\boldsymbol{S}$ for each $\boldsymbol{i}^l \in \boldsymbol{I}$. It can be checked as such separately by elementary arithmetic.[12]

In our example, to find the certificate to check the refinement $\mathit{Prog}_1 \sqsubseteq \mathit{Prog}_0$, we need to solve two systems of linear equations, one for each vertex distribution in Prog_0's relational semantics (Fig. 2). For $\boldsymbol{i}^1 \mathrel{\hat{=}} (1/2, 1/2, 0)$ the system is

- $0 \leq c^1_k$ for $1 \leq k \leq 4$;
- $c^1_1 + c^1_2 + c^1_3 + c^1_4 = 1$;
- $c^1_1(0,0,1) + c^1_2(1/2,0,1/2) + c^1_3(0,1/2,1/2) + c^1_4(1/2,1/2,0) = (1/2,1/2,0)$.

The solution $\boldsymbol{c}^1 = (0,0,0,1)$ thus forms part of the certificate for verifying refinement. The complete certificate would also need the vector $\boldsymbol{c}^2 = (1,0,0,0)$ for Prog_0's other vertex point $(0,0,1)$.

[12] These certificates are the essential components of Principles *P1,2* that make our conclusions independent of the correctness of our tools.

4.3 Refuting refinement

In the case the refinement fails, that is for some $1 \leq l \leq L$ there is no $\boldsymbol{c}^l$ (in the sense of the previous section), we can do better than simply "the solver failed."

We refer to Fig. 7 and its surrounding discussion, and see that if $\boldsymbol{i}^l \notin \lceil \boldsymbol{S} \rceil$ then there must be a hyperplane that separates $\boldsymbol{i}^l$ from $\lceil \boldsymbol{S} \rceil$, i.e. a hyperplane with $\boldsymbol{i}^l$ on one side and all of $\lceil \boldsymbol{S} \rceil$ strictly on the other: in Fig. 7 that is the plane shown, having $\boldsymbol{i}^3 \mathrel{\hat{=}} (0, 1/2, 1/2)$ non-strictly on its lower side and all of Prog_0's results strictly on the upper side.

Formulated in the expectation logic of Fig. 5, refinement failure $\mathit{spec} \not\sqsubseteq \mathit{imp}$ at some initial state s° requires an expectation *expt* with the strict inequality $\mathsf{wp}.\mathit{spec}.\mathit{expt}.s^\circ > \mathsf{wp}.\mathit{imp}.\mathit{expt}.s^\circ$. That *expt* is given by the normal $(2, 0, 1)$ of the separating plane in Fig. 7, and $\mathsf{wp}.\mathit{imp}.\mathit{expt}.s^\circ$ is its constant term $1/2$ when it touches Prog_1 at $\boldsymbol{i}^3$. To touch Prog_0 it would need to move higher, to constant term 1, which is thus the value of $\mathsf{wp}.\mathit{imp}.\mathit{expt}.s^\circ$ for that same *expt* $(A, B, C) \mapsto (2, 0, 1)$.

To find such a hyperplane, we must solve for the N-vector $\boldsymbol{h}$ in the equations

$$- \left(\textstyle\sum_{1 \leq n \leq N} h_n s_n^k\right) > \left(\textstyle\sum_{1 \leq n \leq N} h_n i_n^l\right) \quad \begin{array}{l}\text{for all } 1 \leq k \leq K \\ \text{and the inconvenient } l \text{ in particular,}\end{array}$$

thus K inequations in this case.

Note well that if we have obtained $\boldsymbol{i}^l$ from a failure of refinement determined as in Sec. 4.2, then the equations immediately above are guaranteed to have a solution. That solution $\boldsymbol{h}$ together with initial state s° is the *certificate refuting the proposed refinement.*[12 again]

In our example we saw that Sec. 4.2 failed for $\boldsymbol{i}^3$; to find our certificate for that failure we therefore solve

$$h_1/2 + h_2/2 \ > \ h_2/2 + h_3/2 \quad \text{and} \quad h_3 \ > \ h_2/2 + h_3/2 \ ,$$

for which one solution is of course the normal $\boldsymbol{h} \mathrel{\hat{=}} (2, 0, 1)$ shown in Fig. 7.

We emphasise that simply the failure of Sec. 4.2 to show some inconvenient d' is not in a convex closure $\lceil \boldsymbol{S} \rceil$ is not above challenge: how do we know the solver itself is not incorrect? The refutation certificate generated for d' by this section –given to us by the hyperplane duality– is independently verifiable, and that is its importance.[13]

4.4 Source-level refutation

Finally in this section we consider how to turn the certificate for refuting refinement into a hint presented at the source level.

For our example we imagine a distributed system comprising a number of processors, each executing its local code. A scheduler coordinates the behaviour of the entire system, by determining which of the processors is able to execute

[13] Hyperplanes are used similarly in probabilistic process algebras to generate distinguishing contexts [4].

		Resulting weakest pre-expectation ↓
	$s:= A \;_{0.5}\!\oplus\; s:= B$	1
	$s:= A \;_{0.5}\!\oplus\; s:= C$	1.5
least →	$s:= C \;_{0.5}\!\oplus\; s:= B$	0.5
	$s:= C \;_{0.5}\!\oplus\; s:= C$	1

The pre-expectation is calculated wrt. $(A, B, C)\mapsto(2, 0, 1)$ in each case.

Fig. 8. The four resolutions of $Prog_1$.

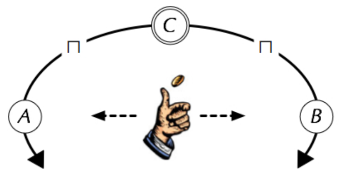

Fig. 9. ⊓-Adversarial scheduling.

an (atomic) local execution step; the overall behaviour of the system can be analysed via an interleaving-style semantics [3]. In the most general setting we can represent the scheduler's choice by nondeterminism; in the case that the distributed protocol contains a vulnerability due to the scheduling (i.e. the events can be ordered so as to break the specification) we shall show how the certificate for failure can be used to find automatically the failing schedule.

As an illustration, consider the simple distributed system of Fig. 9 where initially Processor C is scheduled, then a probabilistic choice $_{1/2}\!\oplus$ is taken whether to continue clockwise or anti-clockwise; the adversarial scheduler can however on the very next step decide whether to remain at C or to move in the direction chosen. One might *specify* with $Prog_0$ that next-in-line Processors A, B should be fairly treated wrt. each other, whether the move occurs or not; but the *implementation* we suggested immediately above first chooses the direction to move via $(s := A) \;_{1/2}\!\oplus\; (s := B)$, and then demonically either confirms the move (skip), or inhibits it $(s:= C)$. The effect is an equivalent but differently written formulation of $Prog_1$ (which we know does *not* refine $Prog_0$):

$$\overbrace{(s:= A) \;_{1/2}\!\oplus\; (s:= B)}^{\text{choose schedule}}; \qquad \overbrace{\mathsf{skip} \sqcap (s:= C)}^{\text{execute schedule, or inhibit}} \tag{3}$$

Because the witness $expt \mathrel{\hat{=}} (A, B, C)\mapsto(2, 0, 1)$ to $Prog_0 \not\sqsubseteq Prog_1$ is based on *semantics*, it applies to this form (3) of $Prog_1$ too, even though it is now more confusingly presented. In general, no matter how many statements are composed, the bad-resolution -selecting process can be carried out on each component separately, rear-to-front: the minimised pre-expectation for one component becomes the post-expectation to be minimised for the one immediately before, and so on to the beginning. That greatly reduces the complexity of finding the schedule.[14]

[14] This trick is well known in game theory [16].

To see how this works, we take the certificate for failure of $Prog_0 \sqsubseteq Prog_1$, and refer to (3) and Fig. 5 to compute[15]

$$\begin{array}{rl} & \mathsf{wp}.(\mathsf{skip} \sqcap (s := C)).\langle 2,0,1\rangle \\ = & \mathsf{wp}.\mathsf{skip}.\langle 2,0,1\rangle \;\; \mathbf{min} \;\; \mathsf{wp}.(s := C).\langle 2,0,1\rangle \\ = & \langle 2,0,1\rangle \;\; \mathbf{min} \;\; \langle 1,1,1\rangle \\ = & \langle 1,0,1\rangle \end{array}$$

Observe how the **min** in the calculation corresponds to the resolution of $\sqcap$ in the code, so that in computing the minimum we also select the bad schedule. In this case, the last-line minimum is achieved from the previous line by taking pointwise choices $(A,B,C) \mapsto \langle$*right*, *left*, *don't-care*$\rangle$, which gives the failing schedule for the second statement: at A take $s := C$ (go right); at B take skip (go left); at C take either. Thus the conditional if $s{=}A$ then $(s := C)$ else skip fi describes concisely and at the source level a schedule that defeats the specification, i.e. if A is suggested by the first statement $(s := A)\ {}_{1/2}\!\oplus (s := B)$ then *inhibit* and stay at C, otherwise *accept* the move to B.

Again we achieve independence from the correctness of our tools,[12 yet again] since it is trivial syntactically that our selection *is* a resolution of *imp*; it is also obvious what its single result distribution is *and* that *spec* cannot produce it.

This is a typical failure in such systems: the scheduler "exploits" a probabilistic outcome that the specifier/developer did not realise was a vulnerability.

5 Finding adversarial schedules in distributed systems

More generally than Sec. 4.4 we fix a set of N processors, each executing "local" code $P_1, \ldots, P_N$ respectively, and overall implementing some protocol. The asynchronous execution of the protocol can be modelled by assuming that each computation step is taken by one of the P_n's, chosen arbitrarily by the adversarial scheduler — in other words is the nondeterministic choice $\sqcap_{1\leq n \leq N}\ P_n$, where we have introduced notation for the generalised nondeterministic choice over a finite set; we also write $Prog^K$ for K sequential compositions of the program *Prog*. The analysis of protocols like these normally considers "runs" that define the set of possible execution orders of the P_n's, which execution orders can be made on the basis of the current state. We describe these runs explicitly as follows.

Definition 3. *Given processors's local code $P_1, \ldots, P_N$, an* execution schedule *is a map $\sigma \in \mathbb{N} \to S \to \{1..N\}$ so that $\sigma.k.s$ defines the number of the processor that would be selected in the k-th step of the protocol if the state at that point were s. We write $\sigma_K \in \{0..K\} \to S \to \{1..N\}$ for the K-bounded execution schedule, namely the schedule σ restricted to the first K steps of the protocol.*

In the following definition we allow P to be subscripted with a function $f \in S \to \{1..N\}$ –rather than a constant– so that P_f from state s behaves as $P_{f.s}$ would; the application of a schedule can then be defined as follows.

[15] We abbreviate the expectation using $\langle \cdots \rangle$.

Definition 4. *Let σ_K be an K-bounded execution schedule; the resulting K-bounded execution sequence is then written*

$$(\sqcap_{0 \le n \le N} P_n)^{\sigma_K} \quad \hat{=} \quad P_{\sigma.0}; \cdots ; P_{\sigma.K}$$

We can now investigate the behaviour of *bounded execution sequences* of the protocol, by considering parameterised specifications. For example, suppose $Spec_K$ denotes a specification of the protocol up to K steps, and our aim is to investigate whether such bounded properties hold of the program.

In such a distributed system, we say that a *certificate to refute a proposed specification* $Spec_K \sqsubseteq (\sqcap_{0 \le n \le N} P_n)^K$ is a K-bounded schedule σ_K such that $(\sqcap_{0 \le n \le N} P_i)^{\sigma_K}$ is not a refinement of $Spec_K$. The next lemma shows how to compute one.

Lemma 1. *Suppose that $Spec_K \not\sqsubseteq (\sqcap_{1 \le n \le N} P_n)^K$, and that $(expt, s^\circ)$ is an (expectation, (initial) state) counterexample pair for the whole failure, as at Sec. 4.3. Define expectations $expt_K \cdots expt_0$ by $expt_K \hat{=} expt$, and $expt_{k-1} \hat{=} \mathsf{wp}.(\sqcap_{1 \le n \le N} P_n).expt_k$, for $1 \le k < K$. Now define the schedule σ_K to give a result $\sigma_K.k \hat{=} f_k$, where each $f_k \in S \to \{1..N\}$ is crafted –as we did at the end of Sec. 4.4– so that $\mathsf{wp}.P_{f_k}.expt_k \;=\; \mathsf{wp}.(\sqcap_{0 \le i \le n} P_i).expt_k$. Then the resulting σ_K is a counterexample schedule.*

Proof. (Sketch.) As in Sec. 4.4 the hyperplane-generated expectation can "prune" nondeterministic choice from the (purported) implementation so that only the failing behaviour is left: one simply considers all deterministic resolutions and picks the one for which the pre-expectation wrt. the witness is minimised. The formal proof appears elsewhere [13].

We illustrate Lem. 1 with a small example case study elsewhere [13].

Finally we note that once we have the overall certificate $(expt, s^\circ)$, assuming the complexity of computing $\mathsf{wp}.P_n.expt$ is constant for every $expt$ and n, the complexity of breaking it up into a finer-grained failing schedule σ_K is $O(KN)$.

6 Implementing the search for certificates

In this section we describe how the search for certificates for failure can be implemented using an SMT solver.

Given two *pGCL* programs *spec* and *imp* we first compute the vertices generating their result distributions, as described in Sec. 4.1; and we formulate the satisfiability problem of Sec. 4.2 to attempt to prove refinement. It is exported to a to a general SMT solver [6] which, if successful, provides a certificate $\boldsymbol{c}$ of refinement.

If it fails, the dual problem (as Sec. 4.3) is formulated for that failure, with the inconvenient distribution providing the coefficients i^l_n and s^k_n, and then we solve for the hyperplane-normal coefficients h_n. Success there is guaranteed, and the normal $\boldsymbol{h}$ is the certificate of refutation.

An alternative approach is to attempt first to refute the refinement (Sec. 4.3) for each implementation distribution. If refutation fails for all of them, then we calculate a certificate of refinement (Sec. 4.2).

7 Conclusions and future work

We have shown how to generate automatically a witness to the failure of a hypothesised refinement $spec \sqsubseteq imp$. We have not yet specifically automated the subsequent production of a source level certificate generator, although a small change to the wp-generator implemented in the HOL system [10] will be a good place to start.

This work differs significantly from other work using SMT-solvers [7] which is unable to produce an efficiently checkable certificate in the form of an expectation, nor a source-level counterexample.

References

1. R.-J.R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer, 1998.
2. E. Clarke, Y. Lu, O. Grumberg, S. Jha, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.
3. E. Cohen. Separation and reduction. In *Mathematics of Program Construction, 5th International Conference*, volume 1837 of *LNCS*, pages 45–59. Springer, July 2000.
4. Y. Deng, R. van Glabeek, C.C. Morgan, and C. Zhang. Scalar outcomes suffice for finitary probabilistic testing. In De Nicola, editor, *Proc ESOP '07*, LNCS. Springer, 2007.
5. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
6. Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T)*. In *CAV 2006*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
7. C. Gonzalia and A.K. McIver. Automating refinement checking in probabilistic system design. To appear in ICFEM '07., 2007.
8. Tingting Han and Joost-Pieter Katoen. Counterexamples in probabilistic model checking. Number 4420 in LNCS, 2007. Proceedings of TACAS 2007.
9. Jifeng He, Karen Seidel, and AK McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28:171–92, 1997.
10. Joe Hurd, A.K. McIver, and C.C. Morgan. Probabilistic guarded commands mechanised in HOL. In A. Cerone and A. de Pierro, editors, *Proc 4th QAPL*, volume 112 of *ENTCS*. Elsevier, 2005.
11. D. Kozen. A probabilistic PDL. *Jnl Comp Sys Sci*, 30(2):162–78, 1985.
12. A.K. McIver and C.C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems.* Tech Mono Comp Sci. Springer, New York, 2005.
13. A.K. McIver, C.C. Morgan, and C. Gonzalia. Proofs and refutations for probabilistic systems. Available at `http://www.ics.mq.edu.au/∼anabel/FM08.pdf`, 2007.
14. C.C. Morgan. *Programming from Specifications.* Prentice-Hall, second edition, 1994.
`web.comlab.ox.ac.uk/oucl/publications/books/PfS/`.
15. C.C. Morgan, A.K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Trans Prog Lang Sys*, 18(3):325–53, May 1996.
`doi.acm.org/10.1145/229542.229547`.
16. J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior.* Princeton University Press, second edition, 1947.