Probabilistic termination in *B*

AK McIver[1], CC Morgan[2], and Thai Son Hoang[2]

[1] Dept. of Computing, Macquarie University, NSW 2109 Australia;
anabel@ics.mq.edu.au
[2] Dept. Comp. Sci. & Eng., University of New South Wales, NSW 2052 Australia;
{carrollm, htson}@cse.unsw.edu.au

Abstract. The *B Method* [1] does not currently handle probability. We add it in a limited form, concentrating on "almost-certain" properties which hold with probability one; and we address briefly the implied modifications to the programs that support *B*.
The *Generalised Substitution Language* is extended with a binary operator $\oplus$ representing "abstract probabilistic choice", so that the substitution $prog_1 \oplus prog_2$ means roughly "choose between $prog_1$ and $prog_2$ with some probability neither one nor zero". We then adjust *B*'s proof rule for loops — specifically, the variant rule — so that in many cases it is possible to prove "probability-one" correctness of programs containing the new operator, which was not possible in *B* before, while remaining almost entirely within the original Boolean logic.
Applications include probabilistic algorithms such as the IEEE 1394 Root Contention Protocol ("FireWire") [9] in which a probabilistic "symmetry-breaking" strategy forms a key component of the design.

1 Introduction

A coin is "almost certain" to come up heads eventually, if flipped often enough; mathematically, we say that heads will eventually appear *with probability one.* Such "coin flips" have many applications in computer programming, in particular in the many symmetry-breaking protocols found in distributed systems.

An example is the IEEE 1394 FireWire protocol [9], in which a leader is elected from a collection of processes executing identical code over an acyclic connected network. At its final "root-contention" stage, two processes can enter livelock while each tries repeatedly to elect the other. The protocol breaks the livelock by having the processes flip two coins, one each, continuing until the outcomes differ; at that stage, the process with "heads" becomes the leader, and the one with "tails" concedes. Because it is almost certain that the outcomes will eventually differ, the contention is resolved.

Using *standard* (i.e. non-probabilistic) *B*, we cannot express such protocols exactly. The closest we can come is to use demonic choice for coin flips, along the lines of Fig. 1. We cannot prove termination of that program, however: the demonic choices $\square$ could choose to make xx and yy equal every time... forever. Thus any standard *B* presentation of this protocol [2] would have to include an *informal* termination argument at this point.

```
$xx, yy := heads, heads$;
WHILE $xx = yy$ DO
    $xx := heads \;\Box\; xx := tails$;
    $yy := heads \;\Box\; yy := tails$
END
```

Program 1a: *Demonic coin-flips*

In Prog. 1a, the binary operator $\Box$ is B's demonic choice.[3] Variable xx represents the coin of some process X, and yy is the coin of process Y. The loop body — representing one "round" of attempting to break the livelock — is executed until the two processes make different choices. At that stage, if it is ever reached, the process whose variable holds *heads* becomes the new leader.

Fig. 1. Demonic abstraction of FireWire's symmetry-breaking *root-contention protocol*

The contribution of this paper is to show how to give a *formal* argument, for such situations, without much extra complexity in the logic: we augment the syntax of *GSL* with an "abstract probabilistic choice" operator $\oplus$; we extend the distribution laws of *GSL* to deal with it; we adjust the proof obligations for loops so that they are sound for loop bodies containing the new operator, carefully identifying any limitations; and we provide the justification for the soundness of all the above.

Our main technical result is Thm. 1 (Sec. 6). It states roughly that as long as *some* probability is used in programs like Prog. 1a, rather than demonic choice, termination occurs with probability one *no matter what probability was used* (provided certain general conditions are met, which we later explain). We write the loop as in Fig. 2, using $\oplus$ for the unknown probability, and we use a modified WHILE-loop proof obligation to show its correctness. We call $\oplus$ "abstract" because we do not know the precise value it uses.

For soundness, it is sufficient (Sec. 7) that the abstract choices $\oplus$ are implemented by "concrete" probabilistic choices ${}_p\oplus$ that are "bounded away from zero and one" — that is, that there is a fixed constant $\varepsilon > 0$ for the *whole execution* such that all actual probabilistic choices ${}_p\oplus$ used to implement $\oplus$ satisfy $\varepsilon \leq p \leq 1{-}\varepsilon$ at the moment they are executed. The p-values can vary dynamically — we allow demonic choice from a range of probabilities — provided

[3] We are ignoring several practical details of some B realisations, in particular that WHILE-loops and demonic choice do not usually appear together, since the former are restricted to implementation machines while the latter is banned there; and we leave out the INVARIANT and VARIANT sections of the loop. Also, we move freely between *GSL* (e.g. $\Box$) and *AMN* (e.g. WHILE).

```
$xx, yy := heads, heads$;
WHILE $xx = yy$ DO
   $xx := heads \oplus xx := tails$;
   $yy := heads \oplus yy := tails$
END
```

Program 2a: *Abstract coin-flips*

In Prog. 2a, the binary operator $\oplus$ is the proposed "abstract" probabilistic choice. Provided the actual probability used is neither zero nor one, the loop will almost-certainly terminate and a new leader will be elected.

Fig. 2. Probabilistic abstraction of FireWire's symmetry-breaking protocol

they remain within the inverval $[\varepsilon, 1-\varepsilon]$ for the chosen ε. We call that a *proper* implementation of $\oplus$.[4]

The two adjustments we make to the WHILE proof-obligations are (A) to change the termination argument so that the variant must be bounded *above* as well as below; and in proving its strict decrease (B) to interpret $\oplus$ "angelically" rather than "demonically" — that is, we require only that the body *can* decrease the variant, not that it must. For all other uses — for example, (C) to prove preservation of the invariant — operator $\oplus$ is interpreted demonically.

The full rule is given at Thm. 1 (p16), where (A), (B) and (C) are labelled.

In Sec. 2 we briefly recall the details[5] of *pGSL* [11], the fully probabilistic version of *GSL*; in Sec. 3 we explain the role of "almost-certain" properties, and how the standard variant rule fails to deal with them; in Sec. 4 we appeal to the "zero-one" law from probability theory that underlies our approach, and we adapt it to *pGSL*; in Sec. 5 we give the modified variant rule that arises from it. Section 6 assembles the pieces into a single rule, and Sec. 7 shows how it can all be expressed in our original Boolean domain. The remaining sections describe an example — the root-contention protocol — and discuss implementation issues.

[4] It is a slightly stronger condition than just lying strictly between zero and one, analogous to the use of *uniform-* rather than simple continuity of functions in analysis.

[5] One detail is that, because *GSL* allows "naked guarded commands" [15, 20], our *pGSL* as defined earlier [11] allows infinite expectations, generated by guards and by the miraculous substitution MAGIC. However, the theory [17] on which the current paper rests is itself based on *pGCL* [18], extending the original "Dijkstra-style" *GCL* [3] which is miracle-free — and so its expectations lie in the interval $[0, 1]$ (that is, without ∞). We finesse this slight mismatch by excluding infeasible substitutions in our treatment here, in particular by treating IF $\cdots$ END as a whole.

2 Full probabilistic reasoning in *pGSL*

To explain and justify our approach, we must temporarily appeal to the fully explicit probabilistic B logic *pGSL*, where $prog_1\,{}_p\!\oplus prog_2$ means "choose program $prog_1$ with probability p, and program $prog_2$ with probability $1{-}p$". Although the justification is somewhat detailed, the results are simple; and once they are re-inserted into the simpler Boolean domain (Sec. 7), no arithmetic is required.

A comprehensive introduction to *pGSL* is given elsewhere [11].

2.1 Brief introduction to *pGSL*

The numeric program logic *pGSL* uses real- rather than Boolean-valued expressions to describe program behaviour: the numbers represent "expected values" rather than the normal predicates that definitely do, or do not hold. Given a state space S, let the predicates over S be $\mathcal{P}S$, and let the *expectations* over S be $\mathcal{E}S$.

Consider the simple program

$$xx := -yy \quad {}_{\frac{1}{3}}\!\oplus \quad xx := +yy\ , \tag{1}$$

over integer variables xx, yy, using a construct ${}_{\frac{1}{3}}\!\oplus$ which we interpret as "choose the left branch with probability 1/3, and choose the right branch with probability $1 - 1/3$". Recall that for any predicate *post* over *final* states, and a standard *GSL* substitution *prog*, the predicate $[prog]post$ acts over *initial* states: it holds just in those initial states from which *prog* is guaranteed to reach *post*. If *prog* is probabilistic, as Prog. (1) is, what can we say about the *probability* that $[prog]post$ holds in some initial state?

It turns out that the answer is just $[prog]\langle post\rangle$, where we use angle brackets $\langle\cdot\rangle$ to convert predicates to expectations, whose values here are restricted to the unit interval: $\langle false\rangle$ is 0 and $\langle true\rangle$ is 1, and in general $\langle post\rangle$ is the characteristic function of (the set denoted by) *post*.

Now in fact we can continue to use substitution, once we generalise $[prog]$ itself to expectations instead of predicates: to emphasise the generalisation, however, we use the slightly different notation $[\![prog]\!]$ for it. We begin with the two definitions

$$\begin{array}{rcc}
[\![xx := E\,]\!]exp & \widehat{=}^{6} & \text{“}exp \text{ with } xx \text{ replaced everywhere by } E\,\text{”} \qquad (2)\\
[\![prog_1\ {}_p\!\oplus\ prog_2]\!]exp & \widehat{=} & p \times [\![prog_1]\!]exp \qquad\qquad\qquad (3)\\
 & & +\ (1{-}p) \times [\![prog_2]\!]exp\ ,
\end{array}$$

in which *exp* is an expectation; and for Prog. (1) we now calculate the probability that the predicate

$$\text{the final state will satisfy } xx \geq 0 \tag{4}$$

holds in a given initial state. We have

[6] We use "$\widehat{=}$" for "is defined to be".

$$
\begin{array}{lll}
 & [\![xx{:=}\ -yy \quad {}_{\frac{1}{3}}\oplus \quad xx{:=}\ +yy]\!]\langle xx \geq 0\rangle & \\
\equiv^{7} & \quad (1/3) \times [\![xx{:=}\ -yy]\!]\langle xx \geq 0\rangle & \text{using (3)} \\
 & + (2/3) \times [\![xx{:=}\ +yy]\!]\langle xx \geq 0\rangle & \\
\equiv & (1/3)\langle -yy \geq 0\rangle + (2/3)\langle +yy \geq 0\rangle & \text{using (2)} \\
\equiv & (1/3)\langle yy \leq 0\rangle + (2/3)\langle yy \geq 0\rangle \ . & \text{arithmetic}
\end{array}
$$

Our answer is the final arithmetic formula above — call it a "pre-expectation" — and the probability we seek is found by reading off the formula's value for various initial values of *yy*:

When *yy* is initially negative,	$\langle true\rangle/3 + 2\langle false\rangle/3$
(4) holds finally with probability	$= 1/3 + 2(0)/3$
	$= 1/3$
When *yy* is initially zero,	$1/3 + 2(1)/3$
	$= 1$
When *yy* is initially positive,	$0/3 + 2(1)/3$
	$= 2/3$.

Those results correspond with our operational intuition about the effect of probabilistic choice ${}_{\frac{1}{3}}\oplus$ in Prog. (1); note in particular how in the "initally zero" case the two branches' probabilities are automatically summed to one, since both establish the postcondition.

2.2 Concise summary of *pGSL*

The rest of *pGSL* is not much more than the above: the definitions of the remaining substitutions are given in Fig. 3.

Implication-like relations between expectations are

$$
\begin{array}{lcl}
exp_1 \Rrightarrow exp_2 & \widehat{=} & exp_1 \text{ is everywhere no more than } exp_2 \\
exp_1 \equiv exp_2 & \widehat{=} & exp_1 \text{ is everywhere equal to } exp_2 \\
exp_1 \Lleftarrow exp_2 & \widehat{=} & exp_1 \text{ is everywhere no less than } exp_2 .
\end{array}
$$

Note that $\models pred_1 \Rightarrow pred_2$ exactly when $\langle pred_1\rangle \Rrightarrow \langle pred_2\rangle$, and so on; that is the motivation for the symbols chosen.

In its full generality, an expectation is a function describing how much each program state is "worth". The special case of an embedded predicate $\langle pred\rangle$ assigns to each state a worth of 0 or of 1: states satisfying *pred* are worth 1, and states not satisfying *pred* are worth 0. The more general expectations arise when one estimates, in the *initial* state of a probabilistic program, what the worth of its *final* state will be. That estimate, the "expected worth" of the final state, is obtained by summing over all final states

[7] Later we explain the use of '≡' rather than '='.

The probabilistic generalised substitution language *pGSL* acts over "expectations" rather than predicates: *expectations* take values in $[0,1] \cup \{\infty\}$.

$[\![xx{:=}\ E]\!]\, exp$	The expectation obtained after replacing all free occurrences of *xx* in *exp* by E, renaming bound variables in *exp* if necessary to avoid capture of free variables in E.
$[\![pre \mid prog]\!]\, exp$	$\langle pre \rangle \times [\![prog]\!]\, exp$, where $0 \times \infty \widehat{=} 0$.
$[\![prog_1 \ \Box\ prog_2]\!]\, exp$	$[\![prog_1]\!]\, exp \ \mathsf{min}\ [\![prog_2]\!]\, exp$
$[\![pre \rightarrow prog]\!]\, exp$	$1/\langle pre \rangle \times [\![prog]\!]\, exp$, where $\infty \times 0 \widehat{=} \infty$.
$[\![\text{SKIP}]\!]\, exp$	exp
$[\![prog_1 \ {}_p\!\oplus\ prog_2]\!]\, exp$	$p \times [\![prog_1]\!]\, exp \ + \ (1{-}p) \times [\![prog_2]\!]\, exp$
$[\![@xx \cdot pred \implies prog]\!]\, exp$	$(\mathsf{min}\ xx \mid pred \cdot [\![prog]\!]\, exp)$, where *xx* does not occur free in *exp*.
$prog_1 \sqsubseteq prog_2$	$[\![prog_1]\!]\, exp \Rrightarrow [\![prog_2]\!]\, exp \quad$ for all *exp*

- *exp* is an expectation (possibly but not necessarily $\langle pred \rangle$ for some predicate *pred*);
- *pre* is a predicate (not an expectation);
- $\times$ is multiplication;
- $prog, prog_1, prog_2$ are probabilistic generalised substitutions;
- p is an expression over the program variables (possibly but not necessarily a constant), taking a value in $[0,1]$; and
- $(\mathsf{min}\ xx \mid pred \cdot [\![prog]\!]\, exp)$ is the infimum over all *xx* satisfying *pred* of the value (in this case) $[\![prog]\!]\, exp$.
- *xx* is a variable (or a vector of variables).

We give the definitions including infeasible or "miraculous" commands [16, Sec. 1.7], but in the main text will avoid them by treating IF $\cdots$ END as a whole, thus effectively restricting our expectations to $[0,1]$ (that is, without ∞).

Fig. 3. *pGSL* — the probabilistic Generalised Substitution Language [11] the worth of the final state multiplied by the probability the program "will go there" from the initial state.

Naturally the "will go there" probabilities depend on "from which initial state", and so the expected worth is a function of the initial state.

When the worth of final states is given by $\langle pred \rangle$, the expected worth of the initial state turns out to be just the probability that the program will reach *pred*, as we saw in the previous section. That is because

$$
\begin{array}{ll}
 & \text{expected worth of initial state} \\[1em]
\equiv & \qquad \begin{array}{rl} & (\text{probability } \mathit{prog} \text{ reaches } \mathit{pred}) \\ \times & (\text{worth of states satisfying } \mathit{pred}) \end{array} \\[1em]
 & + \quad \begin{array}{rl} & (\text{probability } \mathit{prog} \text{ does not reach } \mathit{pred}) \\ \times & (\text{worth of states not satisfying } \mathit{pred}) \end{array} \\[1em]
\equiv & \qquad (\text{probability } \mathit{prog} \text{ reaches } \mathit{pred}) \times 1 \\
 & + \quad (\text{probability } \mathit{prog} \text{ does not reach } \mathit{pred}) \times 0 \\[1em]
\equiv & \text{probability } \mathit{prog} \text{ reaches } \mathit{pred}\ ,
\end{array}
$$

where matters are simplified by the fact that all states satisfying *pred* have the same worth.

2.3 Some *pGSL* idioms

More generally, analyses of programs *prog* in practice lead to conclusions of the form

$$p \quad \equiv \quad [\![prog]\!]\langle post\rangle \tag{5}$$

for some p and *post* — that is, where the pre-expectation is not of the form $\langle pre\rangle$. Given the above, we can interpret in two equivalent ways:

1. the expected worth $\langle post\rangle$ of the final state is at least[8] the value of p in the initial state; or
2. the probability that *prog* will establish *post* is at least p.

Each interpretation is useful, and in the following example we see them both — we look at one round of the root-contention protocol (much idealised in Fig. 2, and with explicit probabilities included) and ask for the probability that the coins will differ after that round:

$$\left[\!\!\left[\begin{array}{l} xx := heads \ {}_{\frac{1}{2}}\!\oplus\ xx := tails\ ; \\ yy := heads \ {}_{\frac{1}{2}}\!\oplus\ yy := tails \end{array} \right]\!\!\right] \langle xx \neq yy\rangle$$

[8] We must say 'at least' in general, because of possible demonic choice in S; and some analyses give only the weaker $p \Rrightarrow [\![prog]\!]\langle post\rangle$ in any case.

$\equiv$ $\quad {}_{\frac{1}{2}}\oplus$, $:=$, and sequential composition

$$[\![xx{:=}\ heads\ {}_{\frac{1}{2}}\oplus\ xx{:=}\ tails]\!](\langle xx \neq heads\rangle/2 + \langle xx \neq tails\rangle/2)$$

$\equiv$ $\quad {}_{\frac{1}{2}}\oplus$ and $:=$

$$\begin{aligned}&(1/2)(\langle heads \neq heads\rangle/2 + \langle heads \neq tails\rangle/2)\\ +\ &(1/2)(\langle tails \neq heads\rangle/2 + \langle tails \neq tails\rangle/2)\end{aligned}$$

$\equiv$ $(1/2)(0/2 + 1/2) + (1/2)(1/2 + 0/2)$ $\quad$ definition $\langle\cdot\rangle$

$\equiv$ $1/2$. $\quad$ arithmetic

We can then use the second interpretation above to conclude that the faces differ with probability (at least[9]) 1/2.

But half-way through the above calculation we find the more general expression

$$[\![xx{:=}\ heads\ {}_{\frac{1}{2}}\oplus\ xx{:=}\ tails]\!](\langle xx \neq heads\rangle/2 + \langle xx \neq tails\rangle/2)\ ,$$

and what does that mean on its own? It must be given the first interpretation, since its post-expectation is not of the form $\langle pred\rangle$, and it means

> the expected value of
>
> $$\langle xx \neq heads\rangle/2 + \langle xx \neq tails\rangle/2$$
>
> after executing $xx{:=}\ heads\ {}_{\frac{1}{2}}\oplus\ xx{:=}\ tails$,

which the calculation goes on to show is in fact 1/2. But for our overall conclusions we do not need to think about the intermediate expressions — they are only the "glue" that holds the overall reasoning together.

Finally — a generalisation of (5) — we mention an idiom we will need later; it is

$$p \times \langle pre\rangle \quad \Rrightarrow \quad [\![prog]\!]\langle post\rangle\ , \tag{6}$$

which means "the probability that *prog* will establish *post* is at least p from any initial state satisfying *pre*". If *pre* holds then $p \times \langle pre\rangle$ is just p; and if it does not hold then $p \times \langle pre\rangle$ is zero, making (6) trivially true.

2.4 Treating standard programs probabilistically

Although we have used the double brackets $[\![\ ]\!]$ for probabilistic substitution, to distinguish it from the ordinary standard substitution (single brackets), if *prog* is standard then in fact there is little distinction to make: an important property of our approach is that if we choose to deal with standard programs within the extended probabilistic framework, we incur no penalty since — as the following lemma shows — the calculations are effectively the same.

[9] Knowing there is no demonic choice in the program, we can say it is exactly 1/2.

Lemma 1. Embedding of standard programs *— If prog is a* standard *and feasible program — that is, one which contains no probabilistic choices and is non-miraculous — then for any postcondition post we have*

$$[\![prog]\!]\langle post\rangle \quad \equiv \quad \langle [prog]post\rangle \ .$$

That is, it is immaterial whether one calculates the precondition $[\cdot]$ *or pre-expectation* $[\![\cdot]\!]$ *for a standard program with respect to a standard postcondition: although one uses true/false in the first case, and 1/0 in the second, the same calculations are performed either way.*

Less formally, recall that in (standard) *GSL* we typically deal with conclusions of the form $pre \Rightarrow [prog]post$, which means "the final state is guaranteed to satisfy *post* if the initial state satisfied *pre*". In *pGSL*, instead we have conclusions $preE \Rrightarrow [\![prog]\!]postE$, which means "the expected value of *postE* in the final state is at least the expected value of *preE* in the initial state".

If we use *pGSL* for standard reasoning, i.e. our pre- and post-expectations are of the form $\langle pre\rangle$ and $\langle post\rangle$ and our program contains no ${}_p\oplus$, then we are dealing with $\langle pre\rangle \Rrightarrow [\![prog]\!]\langle post\rangle$ which, interpreted as above, means "the expected value of $\langle post\rangle$ in the final state is at least the expected value of $\langle pre\rangle$ in the initial state". But we know from elementary probability that the expected value of a characteristic function of a predicate[10] is just the probability that the predicate holds: so we have really said "the probability that *post* holds in the final state is at least the probability that *pre* held in the initial state".

For standard (${}_p\oplus$-free) programs however, predicates either hold (probability 1) or they do not (probability 0). And for x, y in $\{0, 1\}$, to say $x \leq y$ is only to say "y is 1 if x was 1". Thus for standard programs specifically, we have said "the probability that *post* holds in the final state is 1 if the probability that *pre* held in the initial state was 1" — and this is just the usual interpretation in standard *GSL*.

Thus we can use probabilistic substitutions $[\![\cdot]\!]$ for all cases.

3 Almost-certain properties, and the failure of the standard variant rule

3.1 Absolute- versus almost-certain correctness

We saw that $[\![prog]\!]\langle post\rangle$ is the greatest guaranteed probability that predicate *post* will hold after execution of *prog*. When that probability is one, we say that *post* is established almost-certainly:

> Probabilistic program *prog* establishes postcondition *post* from precondition *pre almost-certainly* just when
>
> $$\langle pre\rangle \quad \Rrightarrow \quad [\![prog]\!]\langle post\rangle \ .$$

[10] The *characteristic function* of a predicate returns 1 for states satisfying the predicate, and 0 otherwise — thus our $\langle\cdot\rangle$ merely converts a predicate into its characteristic function.

```
n := 2;                          n := 1;
WHILE n ≠ 0 DO                   WHILE n ≠ 0 DO
   n := n − 1                       n := n − 1 0.5⊕ SKIP
END                              END
```

Program 4a:
Absolute correctness

Program 4b:
Almost-certain correctness

Prog. 4a terminates absolutely after exactly two iterations.

Prog. 4b terminates almost-certainly: although one cannot predict the number of iterations, the chance that it runs forever is zero. (In fact, it terminates after an "expected", rather than exact, two iterations: if the program were run many times and the number of iterations averaged, that average would be two.)

Fig. 4. Absolute- versus almost-certain correctness

Although almost-certain properties can be crucial to the correctness of many programs, the standard *GSL* $[\cdot]$-logic is too weak to be able to verify them in most cases: it is not defined for probabilistic choice, whether abstract $\oplus$ or concrete ${}_p\oplus$. In effect, the best the standard logic can do is to treat the probabilistic behaviour as *demonic* nondeterminism, as we did in Fig. 1.[11]

We say that a property holds *absolutely* if it can be proved using techniques of the standard *GSL* logic, e.g. $pre \Rrightarrow [prog]post$.[12] Figure 4 contrasts absolute- and almost-certain correctness.

3.2 The failure of the standard variant rule

The standard variant rule is not strong enough to prove almost-certain termination of probabilistic loops.

In Prog. 4b the loop terminates almost-certainly because, to avoid termination, the SKIP branch must be chosen every time. That is, since the probability of choosing the second branch i times in succession is $1/2^i$, the probability of choosing it "forever" is $1/2^\infty$, effectively zero. Thus the chance that the first branch is eventually selected — leading to immediate termination — is one.

[11] A useful way — though approximate — of looking at the benefit probabilistic choice offers beyond demonic choice is to regard it as a kind of "fairness". Although a demonic coin can come up tails every time, no matter how often it is flipped, a probabilistic coin must eventually give heads (and tails, too) provided its probabilistic bias is proper.

[12] We overload "$\Rrightarrow$" deliberately, because in both guises it expresses essentially the same idea: we now have $\langle pred_1\rangle \Rrightarrow \langle pred_2\rangle$ iff $pred_1 \Rrightarrow pred_2$.

```
$n := 1$;
WHILE $n \neq 0$ DO
    $n := n - 1 \;\oplus\;$ SKIP
END
```

Program 5a:
Almost-certain correctness

```
$n := 1$;
WHILE $n \neq 0$ DO
    $n := n - 1 \;\Box\;$ SKIP
END
```

Program 5b:
Demonic $\underline{\text{in}}$correctness

Like Prog. 4b, the abstract-probabilistic Prog. 5a terminates almost-certainly: although without knowing the actual probabilities used to implement $\oplus$ at runtime we cannot pre-calculate an expected number of iterations, that makes termination no less certain.

In Prog. 5b, the abstract probabilistic choice has been replaced by demonic choice. Because *in principle* that choice could be resolved to SKIP on all iterations, our logic does not prove termination of the loop at all.

Fig. 5. Almost-certain correctness versus "demonic" incorrectness

Consider now an attempt to prove the termination of Prog. 4b using the standard variant rule. Although the variant of Prog. 4a is just n, we cannot use n for Prog. 4b, since if SKIP is selected then n does not decrease. In fact, for that reason, we cannot establish that *any* variant V will surely decrease: no matter what expression it is, SKIP will leave it unchanged.

And no other existing form of standard *GSL* reasoning can show that such loops terminate, either. Consider the more general loop Prog. 5a. In a technical sense (which we make precise below), the "closest" standard program to it is the demonic Prog. 5b: it is the "best behaved" standard program that behaves "no better than" Prog. 5a. Thus no standard technique can attribute more properties to Prog. 5a than it attributes to Prog. 5b — and since Prog. 5b is indeed *not* certain to terminate, no standard technique can establish termination for Prog. 5a.

The important step is therefore to understand how *pGSL* allows us to strengthen the standard variant technique to exploit the special connection between abstract probability and almost-certain correctness. It turns out that the key lies in probability theory's so-called "zero-one" laws, to which we now turn.

4 A probabilistic *zero-one* law for loops

4.1 Direct calculation of almost certainties

A naive way to determine the probability of termination of a probabilistic WHILE-loop is by direct calculation. In Prog. 4b, for instance, the probability that eventually $n = 0$ can be computed as the infinite sum over the disjoint probabilities that the assignment "$n := n - 1$" causes termination on the i-th iteration:

$$\sum_{i:=1}^{\infty} 1/2^i \quad = \quad 1/2 + 1/2^2 + 1/2^3 + \cdots \quad = \quad 1\ . \tag{7}$$

Unfortunately the computation of infinite limits in general is not practical — often the limit can be finessed by fixed-point methods, but even that is not straightforward.

Fortunately we can do better by appealing to probability theory's so-called *zero-one* laws in combination with *pGSL*'s logic. In Sec. 4.3 below we give a simple zero-one law, essentially transcribed from a probability text [5], but couched in terms of probabilistic WHILE-loops; first, however, we introduce the "demonic retraction", convenient for expressing the law in *pGSL*.

4.2 Demonic retractions of probabilistic programs

In our discussion above of the failure of the standard variant rule, we remarked that in fact *no* standard rule would be sufficient, and we referred to the "closest" standard program. We now make that more precise.

Recall that $[\![prog]\!]\langle post\rangle$ is the greatest guaranteed probability that *post* is established by execution of the probabilistic program *prog*. If *prog* were a refinement of some standard program $prog_d$ (writing "d" for "demonic"), then we would necessarily have

$$[\![prog_d]\!]\langle post\rangle \quad \Rrightarrow \quad [\![prog]\!]\langle post\rangle\ . \tag{8}$$

That is because *prog* — being a refinement $\sqsubseteq$ of $prog_d$ (recall Fig. 3) — must establish any postcondition at least as probably as $prog_d$ does. But the left-hand side of (8) can take values only in $\{0,1\}$, since it is a standard program applied to a standard postcondition, and thus (8) is equivalent to

$$[\![prog_d]\!]\langle post\rangle \quad \Rrightarrow \quad \lfloor [\![prog]\!]\langle post\rangle \rfloor\ ,$$

where $\lfloor\cdot\rfloor$ is the mathematical *floor*[13] function. That leads us to the following definition.

Definition 1. <u>Demonic *retraction*</u> — *Let prog be a probabilistic program in pGSL, and let post and pre be predicates in $\mathcal{P}S$. Since $\langle pre\rangle$ is $\{0,1\}$-valued, we note that post is almost-certainly established under execution of prog from any initial state satisfying pre just when*

$$\langle pre\rangle \quad \Rrightarrow \quad \lfloor [\![prog]\!]\langle post\rangle \rfloor\ . \tag{9}$$

As a syntactic convenience, we therefore make the definition

$$\lfloor prog \rfloor post \quad \widehat{=} \quad ([\![prog]\!]\langle post\rangle = 1)\ ,$$

[13] The *floor* of a real number is the greatest integer that does not exceed it.

in which — note — program prog may be probabilistic but $\llfloor prog \rrfloor post$ *is Boolean-valued, and therefore is a predicate (not an expectation). That allows us to write (9) as simply*

$$pre \quad \Rrightarrow \quad \llfloor prog \rrfloor post \;,$$

where as noted above we write $\Rrightarrow$ *for Boolean as well as "numeric" implication. We call* $\llfloor prog \rrfloor$ *the* demonic retraction *of* $[\![prog]\!]$.

4.3 A zero-one law

Demonic retraction allows a convenient statement of our zero-one law, that if the probability of a loop's termination is bounded away from zero, then in fact it is one.

Lemma 2. *Zero-one law for loops — Let* WHILE G DO *prog* END *be a loop, let* I *be a predicate (its invariant), and let* δ *be strictly greater than zero. If both*

$$I \wedge G \quad \Rrightarrow \quad \llfloor prog \rrfloor I \tag{10}$$

$$\textit{and} \qquad \delta \times \langle I \rangle \quad \Rrightarrow \quad [\![\text{WHILE } G \text{ DO } prog \text{ END}]\!]\langle true \rangle \tag{11}$$

hold, then in fact $I \;\Rrightarrow\; \llfloor \text{WHILE } G \text{ DO } prog \text{ END} \rrfloor (I \wedge \neg G)$.

Proof. It is a standard result from Markov-process theory; a proof specialised to our context can be found elsewhere [17].

Informally, the lemma says that if I is almost-certainly invariant, and if the probability of termination everywhere in I is at least some positive constant δ, then in fact the loop almost-certainly establishes both the invariant and the negated guard from any initial state satisfying the invariant.[14] (Recall (6) for the idiom $\delta \times \langle I \rangle$.)

4.4 Distribution of demonic retraction

We shall use Lem. 2 to construct a variant rule for almost-certain termination. Part of its suitability is that its first antecedent (10) can be established in the usual induction-over-the-syntax style of *pGSL* reasoning: in fact, $\llfloor \cdot \rrfloor$ distributes just as $[\cdot]$ does, except that it is defined for ${}_p\oplus$ as well. Especially important is that the distribution takes place in the *Boolean* domain — the arithmetic ${}_p\oplus$ is converted to a Boolean $\wedge$ by $\llfloor \cdot \rrfloor$-distribution, being treated effectively as demonic choice $\square$.

Lemma 3. Demonic *distributivity — Demonic retraction has the same*[15] *distributivity properties as the standard substitution* $[\cdot]$, *and extends it as follows: if* $0<p<1$ *then*

$$\llfloor prog_1 \;{}_p\oplus\; prog_2 \rrfloor post \quad \equiv \quad \llfloor prog_1 \rrfloor post \;\wedge\; \llfloor prog_2 \rrfloor post \;.$$

[14] It's called a "zero-one" law because it says that the probability of correctness must be either zero or one — it cannot be somewhere in between.

[15] Recall (Footnote 5) we do not treat infeasible substitutions directly: the relevant distribution law here would be for IF $\cdots$ END as a whole.

Proof. *If* $0<p<1$ *and* $0 \le x, y \le 1$ *then* $px + (1-p)y$ *is one iff* x, y *are both 1.*

Note that there is no rule for distributing $\llbracket \cdot \rrbracket$ through WHILE.[16] Dealing with $\llbracket$WHILE G DO *prog* END$\rrbracket$ is after all the purpose of Lem. 2 (in its eventual formulation as Thm. 1), and we must now turn to its second antecedent (11).

5 Probabilistic variants for almost-certain termination

To establish (11) we formulate a probabilistic variant rule. Before stating it, we need several definitions.

5.1 Angelic retraction, and definiteness

"Angelic" retraction is the dual of demonic: it expresses "not almost-certain not to succeed".

Definition 2. Angelic *retraction* — *Let prog be a probabilistic program in pGSL, and let post and pre be predicates in* $\mathcal{P}S$. *We note that post has* "some chance" *of being established under execution of prog from any initial state satisfying pre if*

$$\langle pre \rangle \quad \Rrightarrow \quad \lceil \llbracket prog \rrbracket \langle post \rangle \rceil \ , \tag{12}$$

where $\lceil \cdot \rceil$ *is the mathematical* ceiling[17] *function.*

As a syntactic convenience, we make the definition

$$\lceil\!\lceil prog \rceil\!\rceil post \quad \widehat{=} \quad (\llbracket prog \rrbracket \langle post \rangle \neq 0) \ ,$$

in which — again — program prog may be probabilistic but $\lceil\!\lceil prog \rceil\!\rceil post$ *is a predicate. That allows us to write (12) as simply*

$$pre \quad \Rrightarrow \quad \lceil\!\lceil prog \rceil\!\rceil post \ .$$

We call $\lceil\!\lceil prog \rceil\!\rceil$ *the* angelic retraction *of* $\llbracket prog \rrbracket$.

We will also need a stronger form of "some chance", since in some cases we cannot allow that chance to be arbitrarily small.

Definition 3. Definiteness — *A probabilistic program prog is said to be "definite" if there is some positive constant* Δ *so that, for all postconditions post, if prog establishes post with some non-zero probability, then in fact it establishes it with probability at least* Δ *— that is, program prog is* definite *iff there is a* $\Delta > 0$ *such that for all post*

$$\Delta \times \lceil \llbracket prog \rrbracket \langle post \rangle \rceil \quad \Rrightarrow \quad \llbracket prog \rrbracket \langle post \rangle \ . \tag{13}$$

Further, we say that a family of programs is uniformly definite *if there is a single* Δ *with respect to which all members of the family are definite.*

In Sec. 7 we state simple sufficient conditions for a program to be definite; App. A gives examples of programs that are not.

[16] That does not mean that $\llbracket$WHILE$\cdots\rrbracket$ is undefined; rather it means that it cannot be expressed in terms of $\llbracket \cdot \rrbracket$ alone applied to the body.

[17] The *ceiling* of a real number is the least integer that it does not exceed.

5.2 A probabilistic variant rule

We can now formulate a rule for establishing the second antecedent (11) of Lem. 2.

Lemma 4. <u>*Probabilistic variant rule*</u> — *In the loop* WHILE G DO *prog* END*, with its invariant I, suppose we have an integer-valued expression V over the state space such that*

- *V is* bounded above and below*: There are integer constants L, H such that*

$$I \wedge G \quad \Rrightarrow \quad L \leq V \leq H \text{ ; and} \tag{14}$$

- *V has* some chance *of decrease: For all N,*

$$I \wedge G \wedge (V = N) \quad \Rrightarrow \quad [\![prog]\!](V < N)\ . \tag{15}$$

Provided the loop body prog is definite, there is a positive δ — as required by (11) — such that the loop terminates with at least that probability from any state satisfying I; that is, we have

$$\delta \times \langle I \rangle \quad \Rrightarrow \quad [\![\text{WHILE } G \text{ DO } prog \text{ END}]\!]\langle true \rangle\ , \tag{16}$$

as required in Lem. 2.

Proof. Suppose the loop has not terminated, thus that $I \wedge G$ holds in the current state. By (14) the probability of termination from the current state cannot be less than the probability of $H - L + 1$ successive decreases of V; by (15) we know that each decrease occurs with some non-zero probability; because prog is definite we know that probability is at least Δ for some $\Delta > 0$; and the overall probability of termination is therefore no less than Δ^{H-L+1}.

Thus we satisfy our conclusion (16) with $\delta := \Delta^{H-L+1}$.

Now we have reduced the second antecedent (11) of Lem. 2 to the angelic property (15), and for that we can use distribution properties analogous to those of Sec. 4.4.

5.3 Distribution of angelic retraction

Angelic distributivity is — like demonic distributivity — almost the the same as normal $[\cdot]$-distributivity; it has however an extra restriction.

Lemma 5. <u>Angelic *distributivity*</u> — Angelic *retraction has the same*[18] *distributivity properties as the standard substitution $[\cdot]$ except as follows:*

- *If $0<p<1$ then*

$$[\![prog_1\ {}_p\!\oplus\ prog_2]\!]post \quad \equiv \quad [\![prog_1]\!]post\ \vee\ [\![prog_2]\!]post\ .$$

[18] Recall Footnote 15.

– *Provided the family prog of programs determined by values of xx satisfying pred is uniformly definite, we have*

$$\lceil\!\lceil @xx \cdot (pred \implies prog) \rceil\!\rceil post \quad \equiv \quad (\forall xx \cdot \; pred \Rightarrow \lceil\!\lceil prog \rceil\!\rceil post) \; .$$

Proof. *The proof of the first case is by arithmetic, as for the demonic case. In the second case, the only difficulty is that an infinite set of non-zero probabilities can nevertheless have infimum zero; but that is excluded by the uniform definiteness provision.*

Note that in the crucial case of probabilistic choice we have the angelic $\vee$ instead of the demonic $\wedge$. Again there is no distribution rule for WHILE.

6 Re-assembling the proof rule for loops

If we draw together all the material from above, we can state in summary that to prove almost-certain correctness of a WHILE-loop whose body is definite, we must find an invariant and variant and proceed as usual, except that

– We ensure the variant is bounded above (as well as below);
– In proving "safety" properties, such as preservation of the invariant, we interpret probabilistic choice demonically using $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$; and
– In proving "liveness" properties, such as decrease of the variant, we interpret probabilistic choice angelically using $\lceil\!\lceil \cdot \rceil\!\rceil$.

The crucial point is that in the above we do not refer to the *values* of the probabilities, except (see Sec. 7) in checking the provisos. Stating that formally, we have our main theorem:

Theorem 1. *<u>Almost-certain correctness of loops</u> — Suppose we have a loop* WHILE G DO *prog* END, *where prog is definite. Furthermore let I be a predicate and V be an integer-valued expression over the program variables; and let L and H be constant integers. Then*

$$\begin{array}{llcll}
\textit{If} & G \wedge I & \Rrightarrow & L \le V \le H & (A)^{19} \\
\textit{and, for all } N, & G \wedge I \wedge (V{=}N) & \Rrightarrow & \lceil\!\lceil prog \rceil\!\rceil (V{<}N) & (B) \\
\textit{and} & G \wedge I & \Rrightarrow & \lfloor\!\lfloor prog \rfloor\!\rfloor I \; , & (C) \\
 & & & & \\
\textit{then} & I & \Rrightarrow & \lfloor\!\lfloor \text{WHILE } G \text{ DO } prog \text{ END} \rfloor\!\rfloor I \; . &
\end{array}$$

For establishing the antecedents involving $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$ and $\lceil\!\lceil \cdot \rceil\!\rceil$ we use their distribution laws, which are almost the same as for standard substitution $[\cdot]$. Using them effectively is the subject of our final technical section.

19 Recall Page 3 for these labels.

7 Returning to purely-Boolean reasoning

7.1 Using abstract choice $\oplus$; ensuring definiteness

To establish the ($0<p<1$)-provisos of the distribution laws (Sections 4.4 and 5.3), and to ensure that a loop body is definite, the following are sufficient:

1. Ensure that the probabilities used in choices are *proper* over the whole execution of the program: that is, find a positive constant ε such that every p in any choice ${}_p\oplus$ satisfies $\varepsilon \leq p \leq 1-\varepsilon$ every time the choice is executed; and
2. Do not allow (nested) WHILE-loops in loop bodies.

Condition 1 trivially establishes the ($0<p<1$)-provisos of the distribution laws for ${}_p\oplus$, so in their use we may simply ignore the p altogether. We are then in effect using "quasi distribution-laws"

$$\begin{array}{lcl} \llfloor prog_1 \oplus prog_2 \rrfloor post & \equiv & \llfloor prog_1 \rrfloor post \ \wedge\ \llfloor prog_2 \rrfloor post \\ \llceil prog_1 \oplus prog_2 \rrceil post & \equiv & \llceil prog_1 \rrceil post \ \vee\ \llceil prog_2 \rrceil post\ , \end{array} \qquad (17)$$

obtained from the actual ${}_p\oplus$-laws simply by ignoring the p.[20]

That the two conditions together are sufficient for loop bodies to be definite can be seen as follows. If *prog* contains no loops, then there is a single upper bound K on the number of ${}_p\oplus$'s executed by it from any initial state. If *prog* is guaranteed to establish *post* with any non-zero probability, then in fact that probability must be at least ε^K, where ε is the constant with respect to which its ${}_p\oplus$'s are proper (Condition 1). Simply take $\Delta \mathrel{\widehat{=}} \varepsilon^K$ in (13).

Nested loops are "not allowed" in B, in any case, so the restriction is no hardship: if the effect of a nested loop is needed, it is encapsulated in a separate machine and its specification is used instead.[21]

The two conditions are sufficient also to ensure that the family of programs within any specification is uniformly definite.

7.2 An example

With Prog. 5a we can give a simple example of purely Boolean reasoning for almost-certain correctness: the program is shown to terminate almost-certainly using the invariant $0 \leq n \leq 1$ and the variant n. For decrease of the variant we refer to (15), and calculate

[20] We write "quasi" because $prog_1 \oplus prog_2$ is not actually a program, nor even a class of programs: for example, we cannot tell by looking at ${}_{1/n}\oplus$ on its own whether it is proper or not — it depends on the context, in particular on whether the surrounding program allows n to be arbitrarily large. That is why we must formulate our "real" laws with the p present.

[21] See Appendix A.

$$
\begin{array}{lll}
 & [\![n := n-1 \;\oplus\; \text{SKIP}]\!](n < N) & \\
\text{if} & [\![n := n-1]\!](n < N) \;\vee\; [\![\text{SKIP}]\!](n < N) & \text{using (17)} \\
\text{if} & [\![n := n-1]\!](n < N) & \text{take left-hand disjunct} \\
\text{if} & n-1 < N & \text{assignment} \\
\text{if} & (n \neq 0) \wedge (0 \leq n \leq 1) \wedge (n = N)\ , &
\end{array}
$$

which is the required antecedent. Note no explicit probabilities are used.

8 Application to root contention

In the final *root-contention* stage of the FireWire protocol it is possible for two processes to send "you be leader" messages to each other, creating a potential livelock; each process detects this by receiving such a message from the process to which it has just sent one. The livelock is broken by each process separately choosing, with probability 1/2, either to resend the message after a "short" time or a "long" time; almost certainly one process will eventually choose "short" while the other chooses "long"; and then the "short" process becomes leader, because its message arrives before the other's has been sent. (This is idealised: the actual protocol allows a range of times.)

In Fig. 2 we gave an abstraction of root contention, using *heads* and *tails* for the two choices; here (compare Sec. 2.2) is the calculation showing that the loop satisfies criterion (B) of Thm. 1. The variant is $\langle xx = yy \rangle$, bounded above and below by 1 (criterion (A), using the loop guard), and the invariant is just *true*, trivially maintained (criterion (C)). We have

$$
\left[\!\!\left[\begin{array}{l} xx := \mathit{heads} \;\oplus\; xx := \mathit{tails}\ ; \\ yy := \mathit{heads} \;\oplus\; yy := \mathit{tails} \end{array} \right]\!\!\right] (\langle xx = yy \rangle < N)
$$

$$
\begin{array}{lll}
\equiv & & \oplus,\ :=\ \text{, sequential composition} \\
 & \quad [\![xx := \mathit{heads} \oplus xx := \mathit{tails}]\!](\langle xx = \mathit{heads} \rangle < N) & \\
 & \vee \quad [\![xx := \mathit{heads} \oplus xx := \mathit{tails}]\!](\langle xx = \mathit{tails} \rangle < N) & \\
 & & \\
\Lleftarrow & [\![xx := \mathit{heads} \oplus xx := \mathit{tails}]\!](\langle xx = \mathit{heads} \rangle < N) & \text{drop disjunct} \\
\Lleftarrow & \langle \mathit{tails} = \mathit{heads} \rangle < N & \text{as above} \\
\equiv & 0 < N & \text{definition } \langle\cdot\rangle \\
\Lleftarrow & 1 = N & \\
\Lleftarrow & xx = yy \;\wedge\; \mathit{true} \;\wedge\; (\langle xx = yy \rangle = N)\ , & \text{if } xx = yy \text{ then } \langle xx = yy \rangle \text{ is } 1
\end{array}
$$

which is the antecedent of (B).

9 Implementation issues

The introduction of the $\oplus$ operator into *GSL* requires changes in the *B* implementation, where the developer writes the programs (usually called machines) in the Abstract Machine Notation *AMN*; only after analysis is the program translated into the more concise *GSL* form. We introduce a construct ACHOICE into *AMN* for that purpose, so that

```
ACHOICE
   $prog_1$
OR                    corresponds to        $prog_1 \oplus prog_2$ .
   $prog_2$
END
```

Because we have to prove that variants are bounded above as well as below, a new clause BOUND is introduced to declare the upper bound of a variant. (We assume that the lower bound is zero.) Thus a WHILE-loop now appears

```
WHILE $G$ DO
   $prog$
INVARIANT $I$
VARIANT $V$
BOUND $H$
END .
```

To calculate $[\![\cdot]\!]$ for the loop as a whole, any abstract $\oplus$ in the body is simply treated as $\Box$, generating $\wedge$, except within variant-decrease proof obligations where it generates $\vee$ instead. To implement this behaviour, we split the proof obligations for WHILE into two parts:

1. Obligations for partial correctness, including preserving the invariant of the loop: $\oplus$ treated as $\Box$; and
2. Obligations for decrease of the variant: $\oplus$ treated angelically.

Unfortunately, this split might result in duplicated proof obligations elsewhere. Consider the following example, where an occurence of $\oplus$ followed by an operation with a precondition:

$$prog_1 \oplus prog_2; \quad pre \,|\, prog_3$$

While proving the preservation of the invariant I, we treat $\oplus$ as $\Box$, i.e. we have to prove that both $prog_1$ and $prog_2$ establish *pre*. But the proof of decrease of the variant must be handled separately, because of the angelic interpretation of $\oplus$; and in that case we find that we must prove that *either* $prog_1$ or $prog_2$ establishes *pre*.

This repetition of $[prog_1]pre$ and $[prog_2]pre$ is clearly not a problem in theory, but it is certainly inconvenient in practice if the proofs require manual assistance — since it will have to be given twice. A possible solution is to rely more heavily on the theory of probabilistic loops [17], where we find that only partial correctness is required for preservation of the invariant by the loop body: partial correctness applied to preconditions allows them simply to be discarded.

10 Summary and conclusions

Our contribution has been to observe that for a program all of whose probabilistic choices are proper — are bounded away from zero and one — many interesting almost-certain properties do not depend on the probabilities' actual values, and to specialise that observation to the context of the B method.

To use these results:

1. Specify and refine a system "as normal", introducing $\oplus$ as required.
2. Use ordinary substitution rules $[\cdot]$ except when treating $\oplus$; in those cases, treat $\oplus$ as demonic choice *except* when proving decrease of loop variants, where it is treated angelically.
3. Bound variants above as well as below, if the associated loop bodies contain $\oplus$.[22]
4. If the final program contains abstract choices $\oplus$, implement them (outside of B, in the target language/hardware) with proper concrete choices ${}_p\oplus$, and do not use nested WHILE-loops.

A simple technique to ensure proper implementation of $\oplus$ (Item 4) is to use random devices (whether hardware or software) whose probabilities vary between *fixed*[23] neither-zero-nor-one bounds; since the number of such devices in any program is finite, proper implementation is assured. Choice at runtime within the fixed bounds can be demonic, as for example it would be in the actual root-contention protocol where the delays can be selected demonically from a range.

To calculate expected time to termination, also important, full (numeric) *pGSL* is required, because the answer sought is a number (of e.g. iterations). The "slightly extended" *GSL* we've used here achieves a separation of concerns: prove termination (alone) using a simple logic; use a more complex logic for more discriminating results. Interesting also is *pGSL*'s expressivity, for example the formula (13) which concisely captures a subtle property.

11 Related and future work

We have focussed broadly on the practical verification and refinement of probabilistic programs. The mathematical foundations are explained in earlier, more general work [14] which specialises a quantitative formulation of the temporal logic $qM\mu$ [12] to the "almost-certain" fragment.

Rao [23] takes a similar approach to the almost-certain verification of Unity programs; his results incorporate a Unity-style fairness assumption, which ours do not.

[22] In fact, for simplicity require upper bounds for variants in all cases, since in non-probabilistic loops the variant's initial value is trivially respected as an upper bound.

[23] By "fixed" we mean not a function of the state.

Hurd [7] and Paulin [21] have investigated the verification of probabilistic programs in higher-order logic; their concern so far has been the proof of particular programs, rather than to establish general techniques appropriate for a sequence of program refinements as might occur in an extended development.

Earlier work [6] in probabilistic program verification already makes use of the zero-one laws from probability theory in an "operational" rather than a program-logic setting. The quantitative program logic used here is a generalisation of Kozen's (determinstic) probabilistic PDL [8], where we have incorporated demonic [18] and angelic [10] choice.

Stoelinga [24] gives a nice summary of other work relating to to the verification of FireWire; Fidge and Shankland, and Abrial use probabilistic- and standard predicate transformers respectively.

Fidge and Shankland, going beyond "almost-certain", address the question of "how long?" To analyse the expected time-to-consensus of FireWire and similar algorithms in this style, rather than simply termination, one must use "full" numeric logic instead of the probability-one abstraction. They did this [4] using the *Probabilistic Guarded Command* langauge *pGCL* [13], which is essentially *pGSL* in the original (Dijkstra-) style.

It was Abrial's development of the protocol [2] that provided the target for specialisation of our almost-certain temporal logic [14] and led directly to this presentation. He used Event- rather than "original" *B*, and that introduces some new concerns which could be addressed in future work.

In *Event-B* there is no explicit loop construct — there are only naked guarded commands (i.e. *events*). An abstract event system can be refined by a concrete one where we have more events; and each old (but concrete) event must then refine its abstract counterpart in the usual way. On the other hand, the new events generate different obligations; they must be proved

1. to refine SKIP; and
2. together to decrease a variant.

We must also prove that the overall concrete event system does not deadlock more often than the abstract one — if the abstract one is live then the new one must be; and if the abstract one terminates, then the new one must not terminate "earlier".

Rule (1) would be verified using the "demonic" $\llbracket\cdot\rrbracket$. Rule (2) above ensures that the new events must eventually deadlock (if left alone); that is in order to give the old events the possibility to be "executed" as in the abstraction. Thus in "probabilistic Event-B" the second rule would have to be changed so that the new events are ensured probabilistically to decrease the (bounded) variant on every step (under its guard and the invariant, of course): here we would find $[\![\cdot]\!]$ used in some form.

In our example of Sec. 8, we would end up at some point with the following new events:

$$\begin{array}{lcl} xx = yy & \Longrightarrow & (xx := \mathit{heads} \oplus xx := \mathit{tails}) \parallel (yy := \mathit{heads} \oplus yy := \mathit{tails}) \\ xx < yy & \Longrightarrow & \text{“Process } X \text{ is the leader”} \\ xx > yy & \Longrightarrow & \text{“Process } Y \text{ is the leader”} \ , \end{array}$$

where for convenience we introduce the order *heads* $<$ *tails*.

Of course, the above is still far too abtract. In the real protocol, we have no "coins" such as xx, yy (although, as Stoelinga points out [24], introducing them might be an improvement). And the last two events (daemons) do not exist as such: instead there are some timings and watchdogs. But that system could be considered a faithful abstraction (for the time being).

Acknowledgements

This work was begun by McIver and Jean-Raymond Abrial while they attended a meeting of *IFIP WG 2.3* hosted by the Turku Institute for Computer Science in Finland, for which we thank the organisers.

We thank Colin Fidge and Ken Robinson for their comments, and Abrial both for the interesting discussions that continued after the meeting and for his direct contributions to the paper, especially the remarks concerning *Event-B* in Sec. 11.

Morgan, together with Ken Robinson, thanks the Australian Research Council for its 3-year grant *A00103115* for research into *pGSL*. They and Hoang are grateful also for the kind assistance of *B-Core (UK)* with implementation issues.

References

1. J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. J.-R. Abrial, D. Cansell, and D. Mery. A mechanically proved and incremental development of the IEEE 1394 Tree Identify Protocol. *Formal Aspects of Computing*, 14(3), 2002.
3. E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall International, Englewood Cliffs, N.J., 1976.
4. C. J. Fidge and C. Shankland. But what if I don't want to wait forever? *Formal Aspects of Computing*, 2002. To appear.
5. G. Grimmett and D. Welsh. *Probability: an Introduction.* Oxford Science Publications, 1986.
6. S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, 5:356–380, 1983.
7. Joe Hurd. A formal approach to probabilistic termination. In *Proceedings 15th International Conference on Theorem Proving and Higher Order Logic, 20–23 August 2002. Hampton, Virginia*, volume 2410 of *LNCS*. Springer Verlag, 2002.
8. D. Kozen. A probabilistic PDL. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, New York, 1983. ACM.
9. S. Maharaj, J.M.T. Romijn, and C. Shankland. An introduction to the IEEE 1394 FireWire. *Formal Aspects of Computing*, 2002. To appear.
10. A.K. McIver and C. Morgan. Demonic, angelic and unbounded probabilistic choices in sequential programs. *Acta Informatica*, 37:329–354, 2001.
11. Carroll Morgan. The generalised substitution language extended to probabilistic programs. In *Proc. 2nd International B Conference B'98*, volume 1393 of *LNCS*, 1998. Also available at [22, B98].

12. Carroll Morgan and Annabelle McIver. An expectation-based model for probabilistic temporal logic. *Logic Journal of the IGPL*, 7(6):779–804, 1999. Also available at [22, MM97].
13. Carroll Morgan and Annabelle McIver. *pGCL*: Formal reasoning for random algorithms. *South African Computer Journal*, 22, March 1999. Also available at [22, pGCL].
14. Carroll Morgan and Annabelle McIver. Almost-certain eventualities and abstract probabilities in the quantitative temporal logic *qTL*. In *Proceedings CATS '01*. Elsevier, 2000. Also available at [22, PROB-1]; to appear in *Theoretical Computer Science*.
15. C.C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988. Reprinted in [19].
16. C.C. Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994. At `web.comlab.ox.ac.uk/oucl/publications/books/PfS`.
17. C.C. Morgan. Proof rules for probabilistic loops. In He Jifeng, John Cooke, and Peter Wallis, editors, *Proceedings of the BCS-FACS 7th Refinement Workshop*, Workshops in Computing. Springer Verlag, July 1996. At `www.springer.co.uk/ewic/workshops/7RW`; also available at [22, M95].
18. C.C. Morgan, A.K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.
19. C.C. Morgan and T.N. Vickers, editors. *On the Refinement Calculus*. FACIT Series in Computer Science. Springer Verlag, Berlin, 1994.
20. G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
21. C. Paulin-Mohring. Randomized algorithms in type theory. Presentation at Daghstuhl Workshop, August 2001.
22. PSG. Probabilistic Systems Group: Collected reports. At `web.comlab.ox.ac.uk/oucl/research/areas/probs/bibliography.html`.
23. J.R. Rao. Reasoning about probabilistic parallel programs. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.
24. M.I.A. Stoelinga. Fun with FireWire: Experiments with verifying the IEEE 1394 Root Contention Protocol. In S. Maharaj, C. Shankland, and J.M.T. Romijn, editors, *Formal Aspects of Computing*, 2002. To appear.

A Definiteness is necessary in Thm. 1

The loop *BadLoop*, defined

```
$BadLoop$  $\widehat{=}$  $kk := 1$;
                 WHILE $kk \neq 0$ DO
                     $kk := 0 \;{}_{1/2^{kk}}\!\oplus\; kk := kk + 1$
                 END ,
```

fails to terminate with probability $\prod_{kk:=1}^{\infty}(1-1/2^{kk})$, which is about .29 : that is, it is "quite likely" to terminate (probability .71), but does not terminate almost-certainly. Its body contains an improper probabilistic choice ${}_{1/2^{kk}}\!\oplus$: for any $\varepsilon > 0$ there is a possible execution in which the choice is executed with *kk* so large than $1/2^{kk} < \varepsilon$. As a result, the body is not definite: consider the postcondition $kk = 0$, which the body can establish with arbitrarily small but still non-zero probability.

Yet *BadLoop* satisfies all the antecedents of Thm. 1 except the restriction that the body be definite; in particular, the variant $\langle kk \neq 0\rangle$ (recall the variant of Sec. 8) is decreased on every iteration with some non-zero (but ever smaller) probability. Without that restriction, therefore, we could conclude incorrectly that *BadLoop* terminates almost-certainly.

The same can occur even with proper choices, if a (nested) loop is used. The WHILE-program *CountHeads*, defined as an operation

```
$CountHeads$  $\hat{=}$  $xx := heads;\ nn := 0;$
                WHILE $xx = heads$ DO
                   $xx := heads\ {}_{1/2}\!\oplus\ xx := tails;$
                   $nn := nn + 1$
                END ,
```

contains only proper choices — choice ${}_{1/2}\!\oplus$ for the flip — and yet it is not definite: we have $[\![CountHeads]\!]\langle nn > N\rangle \equiv 1/2^N$, which for large-enough N is less than any $\varDelta{>}0$. If we use it within *WorseLoop*, defined

```
$WorseLoop$  $\hat{=}$  $CountHeads;\ kk := 1;$
               WHILE $nn \leq kk$ DO
                  $CountHeads;$
                  $kk := kk + 1$
               END ,
```

we see the same effect as before: termination is with probability .71, and all antecedents of Thm. 1 are satisfied except definiteness. (And this time all choices are proper.) Again definiteness saves us, in this case excluding *CountHeads* from the loop body.

To use *CountHeads* within the body of *WorseLoop* we would, by the rules of B, have to use a specification instead: the tightest we could get would be along the lines of $(@nn' \cdot nn' \in \mathbb{N}^+ \Longrightarrow nn := nn')$. That specification is definite, but does not satisfy (B): the variant is not guaranteed to decrease with any non-zero probability, and thus soundness is preserved.