Probabilistic affirmation and refutation:
Case studies

CJ Gonzalía\textsuperscript{1}, AK McIver\textsuperscript{2}, and CC Morgan\textsuperscript{3} \textsuperscript{*}

\textsuperscript{1} Dept. Comp. Sci. and Eng., Univ. Nacional del Sur, 8000 Bahía Blanca, Argentina
\textsuperscript{2} Dept. Computer Science, Macquarie University, NSW 2109 Australia
\textsuperscript{3} School of Comp. Sci. and Eng., Univ. New South Wales, NSW 2052 Australia

Abstract. Elsewhere we have defined counterexamples and independent refutation certificates for conjectures of the form “\(A\) is correctly implemented by \(B\)”, where \(A\) and \(B\) are probabilistic programs; and we have implemented a constraint-based tool to generate certificates in the case that the implementation is incorrect. In this paper we extend those results to simple safety properties, and we illustrate the techniques with three case studies.

One key contribution is that our techniques are carried out directly at the level of source code. A second key contribution is that we use theorem-proving and model-checking techniques together, in complementary ways that have not previously been explored for probabilistic systems.

Keywords: Probabilistic systems, counterexamples, quantitative program logic, refinement, constraint solving.

1 Introduction

The great benefit of static proofs of program correctness –using so-called variants and invariants expressed in terms of the program state \([7]\)– is that they can be checked directly at the level of program code. That is, fully formalised variants and invariants can be validated automatically within a trusted theorem prover. Conversely when a proof fails, it is the dynamic behaviour of the system which guides the prover to locate the source of the error (residing in either the proof or the system itself), and provides insight as to how to correct it.

Elsewhere \([1]\) we proposed and implemented a constraint-based automated tool which produces independently checkable certificates either to prove or refute statements of the form \(A \sqsubseteq B\) (\(A\) is correctly implemented, i.e. is refined by \(B\)) where \(A\) and \(B\) are probabilistic programs having a Markov Decision Process (MDP) -style semantics. The purpose of this paper is to apply those ideas to the analysis of safety properties of looping programs. Whilst our context is a proof-based style of verification, we do explore the relationship to techniques for counterexample generation that require the exhaustive state exploration offered by model checking. One of the outcomes of this work is to show how beneficial it is to use both proof-based analysis and model-checking in probabilistic analysis, just as it is in the non-probabilistic case.

\textsuperscript{*} We acknowledge the support of the Australian Research Council Grant DP0558212.
Our approach is based on the *quantitative program logic* first suggested by Kozen [14] for probabilistic programs without nondeterminism, and extended by us to include nondeterminism [19, 16]. Only when nondeterminism is present does the notion of refinement have its full force. We thus allow a Hoare-triple style of analysis [12] to extend to probabilistic properties: Hoare triples are useful as they can be checked at the level of program code: *probabilistic Hoare triples*, for example, can currently be checked automatically within the HOL proving environment [13].

Our principal aim is as follows. Given a system and a safety property, we produce an independently checkable certificate –related to the source code– to support either a proof of correctness or a counterexample for refutation. Our specific contributions in this paper are thus:

1. To explore the relationship between static properties used in proofs and dynamic computation structures. In particular we focus on *result-distributions* and *scheduler strategies* to provide the evidence –the latter at the level of program code– for why a proposed proof of a given static property fails to go through.
2. To describe a number of case studies illustrating these ideas.

Finally we indicate how counterexamples may be computed, and describe the results of some practical experiments.

The key novelty of this work is that the constructions are carried out at the level of source code directly, which is made possible by probabilistic program semantics and logic [16].

We begin with a brief overview of the quantitative program logic and the associated *pGCL* programming language in Sec. 2. In Sec. 3 we show how the logic may be used to define certificates of either proof or refutation of a safety property. In Sec. 3.1 we indicate how our distribution generator [1] can be adapted to compute refutation certificates automatically; and finally in Sec. 4 we give a series of case studies illustrating the techniques.

## 2 Probabilistic guarded commands

The language *pGCL* generalises Dijkstra’s guarded-command language [7] by adding probabilistic choice (and retaining demonic choice); in Fig. 1 we set out how a *pGCL* program *P* can be given either a transition-style semantics \([P]\) (at right), or a reward-based semantics \(W_{p}.P\) (at left), the latter generalising the predicate transformers of Dijkstra. 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. Segala [24] gives a nice description of the issues.

Programs are composed of assignments (\(\equiv\)), sequential composition (\(;\)), various choices (nondeterministic \(\sqcap\), Boolean *if… then… else… fi*, or probabilistic \(p\oplus\)), and either strong (\(do\ldots od\)) or weak iteration (\(it\ldots ti\)). The transition-style semantics \([P]\) of a program *P* delivers a set of probability distributions over
Fig. 1. Transformer and Relational semantics for pGCL [16]

2.1 Systems as demonic games

A probabilistic system is defined by a (finite) set of named deterministic pGCL programs, $P_0, \ldots P_N$, each operating independently. The operation of the system proceeds in discrete steps: at each step an adversarial scheduler selects a $P_n$ to execute, so that a single step of the system is a nondeterministic choice
\[0 \leq n \leq N\], the generalised nondeterministic choice over a finite set. Thus the operation of a system can be thought of as demonic game \([6, 17]\) with a single player. Both asynchronous distributed systems with adversarial schedulers and formal models of software systems using action systems \([3]\) correspond to this model.

A typical analysis of such systems (of either kind) usually considers “runs” which, roughly speaking, define the set of possible execution orders of the \(P_n\)’s. More formally, those execution orders can be made on the basis of the execution trace in terms of a finite sequence of states through which the system evolved. Let \(S^\ast\) be the set of finite sequences of states.

**Definition 1 (Execution schedule).** Given \(pGCL\) programs \(P_0 \cdots P_N\), an execution schedule is a map \(\aleph : S^\ast \to \{0 \cdots N\}\) so that \(\aleph \alpha\) is the index of the program scheduled to execute after the non-empty execution trace \(\alpha\). (A more uniform formalisation would give the distribution of initial states as \(\aleph(\langle \rangle)\); but we prefer to give initial states explicitly.)

A probabilistic computation tree formalises the idea of probability distributions over execution traces, required to give a semantics to temporal properties. Such distributions are as usual given with respect to Borel algebras based on the traces \([9]\). In the following definition we restrict to deterministic programs.

**Definition 2 (Trace distribution).** Given a system defined by deterministic programs \(P_0 \cdots P_N\), initial state \(s_0\) and execution schedule \(\aleph\), we define the corresponding trace distribution \(\langle | P_\aleph | \rangle.s_0\) of type \(S^\ast \to [0, 1]\) to be

\[
\langle | P_k | \rangle.s_0.(s') = 1 \text{ if } s' = s_0 \text{ else } 0
\]

and \(\langle | P_k | \rangle.s_0.(\alpha s') = \langle | P_k | \rangle.s_0.(\alpha s) \times [\langle | P_{\aleph(\alpha s)} | \rangle.s.s']\).

Because each \(P_n\) is deterministic the set \([P_n].s\) will contain exactly one distribution, and we then abuse notation by writing just \([P_n].s.s'\) for the chance that execution of \(P_n\) from initial \(s\) delivers final \(s'\).

For any concretely defined system, we can depict a trace distribution as a computation tree. For example, in Fig. 2 we set out three small programs. A computation tree appears on the left, where the given schedule produces two paths of probability 1/2 each. On the right we see the aggregated distribution of final states achieved at each stage.

If we take \(K\) steps from some \(s_0\) according to the schedule, then the probability of ending in state \(s'\) is given by \([P_R^K].s_0.s' = \sum_{|\alpha|=K} \langle | P_\aleph | \rangle.s_0.(\alpha s')\).

The importance of that definition is that it gives the explicit link between trees and state-based properties, ensuring consistency between them. For example, suppose we wanted to investigate the following conjecture concerning the protocol of Fig. 2: “After any execution, the chance that the state is \(B\) is at least the chance that it is \(A\).” The tree at left does not satisfy that property, since looking at the distributions over states, at right, we see 0 probability for \(s=B\) finally but 1/2 for \(s=A\). As we shall see, algorithmic analyses of these properties
\[ P_0 \triangleq \text{skip} \quad \mathbb{E} \langle X \rangle \triangleq 1 \]
\[ P_1 \triangleq s := A \quad \mathbb{E} \langle X, A \rangle \triangleq 0 \]
\[ P_2 \triangleq s := B \quad \mathbb{E} \langle X, B \rangle \triangleq 2 \]

Fig. 2. A schedule, an execution tree, and distributions over endpoints

are performed on end-point distributions (Fig. 2 right-hand side), where proofs are done in the expectation transformers (\( W_p \) at Fig. 1); but to convince a verifier that a specification is false requires recovering an entire trace distribution possibly with the strategy that led to it (as at left), and that requires the relational semantics ([·] at Fig. 1). Proofs and refutations of such properties is the topic of the next section.

3 Properties, certificates and counterexamples

The \( W_p \) semantics from Fig. 1 allows us to express quantitative properties directly as pre/post-results, generalising standard Hoare triples.

Definition 3 (Probabilistic Hoare triple [16]).
Given a system defined by \( pGCL \) program \( P \), and two expectations \( \text{expt}_0, \text{expt}_1 \), the probabilistic Hoare triple is defined
\[
\{ \text{expt}_0 \} \quad P \quad \{ \text{expt}_1 \} \quad \text{is valid just if} \quad \text{expt}_0 \leq W_p.P.\text{expt}_1.
\]

Given a system defined by (program) \( P \), and initial state \( s_0 \), and expectation \( \text{expt} \), we can define a quantitative safety specification as a threshold on the greatest guaranteed expected value of \( \text{expt} \), after arbitrarily many executions of \( P \) from initial state \( s_0 \). The threshold of \( p \) holds provided the following Hoare Triple is valid:
\[
\{ p \} \quad s := s_0; \quad \text{it} \quad P \quad \text{ti} \quad \{ \text{expt} \}.
\] (1)

An affirmation certificate of (1) is an invariant.

Definition 4 (Affirmation-of-safety certificate). The expectation \( \text{expt}' \) is a certificate affirming the triple (1) if \( \text{expt}' \leq W_p.P.\text{expt}' \), and \( p \leq \text{expt}'.s_0 \).
Invariants expressed in terms of program variables can be checked automatically whenever $P$ itself is iteration-free [13]. Conversely if (1) fails to hold then there must be some finite iteration of $P$ which demonstrates the failure.

**Definition 5 (Refutation-of-safety certificate).** Given is a system defined by $P \doteq \cap_{0 \leq n \leq N} P_n$ with initial state $s_0$. We say that an aggregate distribution is a certificate refuting (1) if there is some finite $k$ such that $\text{Exp} \cdot \text{expt} < p$, where distribution $d \in [P^k] \cdot s_0$ and $\text{Exp} \cdot d$ means “the expected value with respect to $d$.” The corresponding refuting schedule is any $\aleph$, where $d = [P^k_k] \cdot s_0$.

Inspired by state-exploration techniques, in Fig. 3 we set out a simple algorithm for finding refuting aggregate distributions, together with a counterexample strategy if it occurs within $K$ steps of the protocol specified by $P_0 \ldots P_N$. The idea is to compute the possible result (or aggregate) distributions $[P^k] \cdot s_0$, resulting from increasingly many iterations $k \leq K$ of the system, until a failure to satisfy the quantitative safety property is observed [1]. In that case a backward analysis then yields the (relevant prefix of a) refutation schedule [22]. Note that in general the computation of result distributions can be done very compactly and the first point of failure then determines a finite state space from which to begin backward iterations.

In theory we do not need the forward generation stage to compute the strategy; in practice however we find it useful because it allows us to simplify the invariant expression. Moreover forward generation also identifies a sub-state space in which the counterexample is located; and an alternative analysis would then be to translate the system into the format of a state-of-the-art probabilistic model checker such as PRISM, which would then compute the strategy [20].

1. Let $P \doteq \cap_{0 \leq n \leq N} P_n$; compute $p_k \doteq (\min \text{Exp} \cdot \text{expt} | d \in [P^k] \cdot s_0)$ for increasing $k \leq K$ until $p_k < p$, where $[P^k]$ is the semantics of $P$ iterated $k$ times. Let the first such index be $k^{\otimes}$.

2. Define $\text{expt}^{\otimes} \doteq \text{expt}$ and work backwards towards $s_0$ from there for $0 \leq k < k^{\otimes}$ to define $\text{expt}_k \doteq Wp.P.\text{expt}_{k+1}$ and finally construct $\aleph$ so that for all $|\alpha| = k$ and all $s'$ we have $Wp.P_{\aleph}(\alpha, s'), \text{expt}_{k+1} = \text{expt}_k \cdot s'$.

For the first step we only compute finitely many distributions over endpoints, as the property space is determined exactly by the finite convex hull, which in turn determines the complete result set [16]. For the second step, computation of each $\aleph$-value requires solving $O(|S|)$ linear programming problems [22].

**Fig. 3.** Simple procedure for computing refutation-of-safety certificate.

### 3.1 Implementations

Elsewhere [1] we have implemented a forward generator for computing $[P] \cdot s_0$ for a concretely-defined system in $pGCL$. For performing safety analysis we use
it to compute endpoint distributions for (1) of Fig. 3; if an endpoint distribution is discovered which shows that the safety property does not hold then the discovered distribution defines a finite state space from which to do the backward analysis –(2) in Fig. 3– to discover the failing schedule.

For our small case studies, the backwards analysis was performed by hand; for larger systems as we mentioned above, better would be to use the available MDP model checkers [21] appropriately enhanced to keep track of the schedules [20].

4 Case studies in quantitative safety

We consider three examples illustrating the analysis of quantitative safety properties. The first uses a possibly negative-valued expectation to specify fairness in a simple voting scheme. The second uses an expectation to ensure a relationship between variables is always satisfied “on average”. The third describes an analysis for distributed leader election.

4.1 Fair voting

Voting can be used to resolve conflict situations which often arise in distributed systems, such as leader election, or attempts to gain exclusive access to a critical section. We analyse the following simple voting scheme, giving here (in contrast to our earlier algebraic treatment of it [2]) a direct static proof of its fairness property using a novel invariant.

$N$ processes strive for exclusive access to a critical section by executing a distributed election protocol, in which an adversarial scheduler “moderates” which of the processes is allowed to participate. The protocol must have the property that for any pair of processes their winning chances are equal in any competition in which they both participate.

It is normal to assume first that voting occurs in a series of competitions, so that when a processor fails, it can try again in the next competition. Thus the scheduler is then able not only to choose the execution order of the processes, but also which ones to schedule in any given competition. The only restriction is that eventually each process must be scheduled in some competition [23]. Under that weak constraint, the above additional quantitative fairness condition ensures that the scheduler cannot favour one process over another.

A simple scheme to implement distributed voting is set out at Fig. 4. Each voter $n$, if selected, first checks whether it has already voted, and if it has not, sets itself to be the winner, via the assignment $w := n$ with probability $1/k$, after incrementing the variable $k$ recording the current number of participants. The behaviour of a single step of the voting protocol is the nondeterministic choice taken over the (finite) voters; and when the voting protocol takes an arbitrary number of steps, it becomes the partial iteration.
\begin{verbatim}
P_n : if \neg v_n then
\hspace{1cm} v_n := true; record his participation ...
\hspace{1cm} k := k + 1; increase the count ...
\hspace{1cm} w := n_1/k \oplus \text{skip} and now flip to win.
else skip
\end{verbatim}

We have $1 \leq n \leq N$ and $w$ initially 0. Variables $w, k$ record respectively the current winner, and the number of participants; $v_n$ is a local variable registering $n$’s participation in the current competition.

\textbf{Fig. 4.} Local code for node $n$ in a distributed voting scheme.

\[
\text{Vote} \equiv \text{it } P_1 \cap P_2 \cap P_3 \text{ ti}, \quad (2)
\]

where the adversarial choice—in this case who participates next— is modelled by the nondeterministic choice. (Although we do not explicitly treat fairness of the scheduler, the current analysis is still valid in that case.)

The intuition behind this is that the later a process casts its vote, the lower is the chance of winning — but the probabilities are carefully chosen so that the apparent advantage for early voters is offset by their higher chance of being usurped.

Next we specify the pairwise fairness property using a (possibly negative-valued) -expectation

\[
\text{fair}_{n,n'} \equiv [v_n \land v_{n'}] \times ([w = n] - [w = n']),
\]

which takes the values 1, -1 or 0, depending on whether $w$ is $n, n'$ or neither — and we require the relationship to hold only when both $n$ and $n'$ have participated (both $v_n$ and $v_{n'}$ hold). Our full specification of safety for the three-voter system is given by this, for all $n \neq n'$:

\[
\{0\} w, k, v_{1,2,3} = 0, 0, \text{false}_{x3}; \text{Vote} \{\text{fair}_{n,n'}\}. \quad (3)
\]

For $n, n' = 1, 2$ the distribution generator revealed a problem after three steps, as shown in Fig. 5 where both Voters 1,2 vote, but Voter 2 has a higher chance of winning than Voter 1. Formally, the value of $\text{fair}_{1,2}$ in that final distribution is

\[
1/2 \times ([\text{true} \land \text{true}] \times ([2 = 1] - [2 = 2]) + \\
1/6 \times ([\text{true} \land \text{true}] \times ([3 = 1] - [3 = 2]) + \\
1/3 \times ([\text{true} \land \text{true}] \times ([1 = 1] - [1 = 2])
\]

which simplifies to $1/2 \times (0 - 1) + 1/6 \times (0 - 0) + 1/3 \times (1 - 0)$, that is $-1/6$ which is less than the 0 the precondition that specification (3) requires.

Finally we can discover the schedule leading to the counterexample distribution by a backwards analysis using the program logic of Fig. 1. In brief the steps are these:

\footnote{Here we are making the (possibly too strong) assumption that the code in Fig. 4 is executed “atomically”}
1. Use the refuting distribution to simplify the postcondition of the specification: call that $F$.
2. With respect to that simplified postcondition, calculate separately the preconditions $W_P f_P F$ for each deterministic program separately. Call that $F_n$ for each $P_n$.
3. For each state $s$ find the $n_s$ so that $F_{n_s} s$ is least among all the $F_n s$’s. (If there are several, choose one.) Then $P_{n_s}$ is the program to schedule in state $s$ at the last stage in order to reach the refuting distribution.
4. Set $F' = \cap_n F_n$ and repeat the process, working back towards $s_0$.

The strategy for the scheduler to favour $P_2$ (over $P_1$ say) is by scheduling $P_1$, then $P_2$, and then $P_3$ only if necessary to defeat $P_1$. (The dotted line is the implicit skip caused by the exit.)

Fig. 5. Schedule contradicting fair voting

The analysis we have performed allows us to understand the relationship between the simple protocol and the scheduling, and thus helps to design a scheduling policy having the desired fairness property we want. For example, if $P_1$ and $P_2$ must both participate in any competition with exactly $k$ other participants, then the scheduler cannot favour $P_1$ over $P_2$. Details are set out elsewhere [15].

4.2 A library bookkeeping system

Our second case study is an accounting package for a library, originally specified in the B software design system, extended to allow for probabilistic variable assignments [11]. The system, set out in Fig. 6 below specifies three main operations of a library, viz. borrowing and returning books, and performing a stocktake. The probability of losing the books is modelled in the operation Return.
One of the purposes of the accounting software is to keep track of the library’s stock, including an estimate of the lost books, so that the library can budget for their replacement.

In detail the operations are as follows. The variables used to track the stock are \( \text{booksInLibrary} \) and \( \text{booksLost} \); the system also records the loans still in play using the variables \( \text{loansStarted} \) and \( \text{loansEnded} \). When a book is borrowed, the system decrements \( \text{booksInLibrary} \) and increments \( \text{loansStarted} \); when a book is returned, either it “really is,” reversing the effect of \( \text{Borrow} \), or actually it is reported lost so that \( \text{booksLost} \) is incremented. The book loss occurs with some probability \( p \). Finally the \( \text{StockTake} \) operation computes the cost of replacement and resets the variables.

This specification places no restrictions on when the operations are used, except for the obvious guards on \( \text{Borrow} \) and \( \text{Return} \); and so the system is a nondeterministic choice:

\[
\text{Library} \ = \ \text{it Borrow} \cap \text{Return} \cap \text{StockTake} \ 
\]

A correctly designed accounting system would use variables which should accurately record the expected losses over time. This system uses \( \text{loansEnded} \) and \( \text{booksLost} \), and since we know that there is a probability \( p \) that a book is lost on each \( \text{Return} \) action, system \( \text{Library} \) would be expected to lose a proportion \( p \) over a number of returns. With this in mind we define the random variable

\[
\text{Inv}_L \ = \ p \times \text{loansEnded} - \text{booksLost} ,
\]

which captures the idea that the expected number of books lost is expected to be no more than \( p \) times the number of loans ended and is thus providing an average forecast for how much it costs to replace books during the library operation. Assuming an initial state \( s_0 \) such that all variables are 0, our required safety property becomes

\[
\{0\} \ \text{it Library ti} \ \{\text{Inv}_L\} .
\]

In this example for \( p = 1/2 \) (rather high for most libraries...) the forward analyser discovered a counterexample distribution to \( (6) \) after just three steps, which a subsequent backward analysis (as in the previous section) then revealed to be the consequence of the schedule depicted in Fig. 7. With \( p = 1/2 \) the expected value of \( \text{Inv}_L \) in the final distribution of states is

\[
\begin{align*}
\text{book returned, but records reset} & \quad 1/2 \times (1/2 \times 0 - 0) + (1-1/2) \times (1/2 \times 1 - 1) , \\
\text{book lost} & \quad -1/4 which is less than the 0 specified.
\end{align*}
\]

\[5\ A \text{ euphemism...?} \]
\[ \text{Borrow} \triangleq \text{booksInLibrary} := \text{booksInLibrary} - 1; \]
\[ \quad \text{loansStarted} := \text{loansStarted} + 1 \]

\[ \text{Return} \triangleq \text{booksLost} := \text{booksLost} + 1 \]
\[ \quad \text{p} \odot \text{booksInLibrary} := \text{booksInLibrary} + 1; \]
\[ \quad \text{loansEnded} := \text{loansEnded} + 1 \]

\[ \text{StockTake} \triangleq \text{totalCost} := \text{cost} \times \text{booksLost}; \]
\[ \quad \text{booksInLibrary} := \text{booksInLibrary} + \text{booksLost}; \]
\[ \quad \text{loansStarted} := \text{loansStarted} - \text{loansEnded}; \]
\[ \quad \text{loansEnded} := 0; \]
\[ \quad \text{booksLost} := 0 \]

For simplicity we leave out the conditionals which usually in such specifications ensure the operations are scheduled only when meaningful, e.g. that \text{Borrow} is invoked only when \text{booksInLibrary} > 0.

\textbf{Fig. 6.} Accounting software for a library

The adversarial administrator\(^5\) here follows the strategy of stocktaking only when no books are lost, thus resetting the records of good behaviour and leaving only the books-lost “bad behaviour” cases for the auditors to discover when they check the invariant. (As in Fig. 5, the dotted line represents exit.)

\textbf{Fig. 7.} Counterexample schedule for the Library
The problem here is that information for making an accurate estimate of average book loss is being erased by stockTake; we correct that by changing the way the system keeps its records [11]. As well as keeping track of the status of the loans, a new variable called averageLoss is introduced to keep track of the long-term average number of books forecast to be lost. Revising stockTake now to be

\[ StockTake' \doteq averageLoss \doteq p \times loansEnded - booksLost; \]

and

\[ Inv'_L \doteq p \times loansEnded - booksLost + averageLoss, \quad (7) \]

which now expresses the average loss of books over the lifetime of the library, rather than between unfortunately scheduled stocktakes. With the revised stocktake function and invariant relationship, the counterexample Fig. 7 is no longer valid and indeed the invariance of (7) in the revised system can be proved [11] using Inv'_L as a certificate for proof.

4.3 Asynchronous leader election: message passing

Our final example is taken from Fokkink and Pang’s analysis of leader election algorithms on asynchronous rings based on the original Itai-Rodeh algorithms [8]. Fokkink et al. explored possibilities for simplification showing that simplification is only possible under certain circumstances. Here we demonstrate the relationship between static reasoning over a complicated asynchronous system and the generation of counterexamples to locate the problems in proofs. Specifically we have a novel static proof of one of the simplifications suggested by Fokkink and Pang (details below), and we show how the refutation certificate given as a scheduler strategy locates exactly the part of the proof (Invariant B below) that fails to generalise, indicating exactly the limitations of the leadership algorithm.

In Fig. 8 we set out a formalisation for the original Itai-Rodeh algorithm without round numbers — the original algorithm included round numbers, and one of Fokkink and Pang’s contributions was to explore circumstances where they could be removed. A message \( x \) is a tuple \( (x, loc.x, bit.x, hop.x, tk.x) \), where \( loc.x, bit.x, hop.x \) and \( tk.x \) are respectively the message’s location (the node it will reach next), a bit (which can be clean or dirty), its hopcount (how far the message has travelled), and finally the message ticket, which is the lottery value chosen by its origin node. A node \( i \) on the other hand is a triple \( (i, state.i, ntk.i) \), where \( state.i \) and \( ntk.i \) are respectively its state (active, passive, dormant or leader), and its current ticket value in the leadership contest.

The protocol works roughly as follows. All nodes are initially dormant; but at any point a dormant node can become active and contend for leadership.\(^6\) A newly active node generates a message \( x \), which has a clean bit \( bit.x \) and

\(^6\) A node’s possible return to dormancy is not discussed here; it does not occur in the leadership protocol.
whose \( tk.x \) is set to be the same ticket \( ntk.i \) chosen by its generating node \( i \), which in turn was selected randomly. Such messages then travel around the ring, increasing their hop count as they go. At each intermediate node \( i' \neq i \) a message encounters on the way back to its source \( i \) its \( tk.x \) is compared with the node’s \( ntk.i' \); if \( id.x > ntk.i' \) then the intermediate node becomes passive; if \( id.x = ntk.i' \) the message becomes dirty; and if \( id.x < ntk.i' \) then the message is annihilated. If a message returns to its source \( i \) with a clean bit and that source is still active, then the node becomes leader; if it returns with a dirty bit, the node re-activates, choosing a (possibly) new ticket and generating a new message; if the node is passive, the message is discarded. (Note that dormant nodes can spontaneously become active, but passive ones cannot.)

To verify this algorithm we used standard probabilistic-program logic [16], whose advantage over ad-hoc operational arguments—as we have said—is that its proofs are readily automated [13]. One invariant \( A \) implies that at most one leader is possible; a second invariant \( B \) implies that the election remains active unless a leader is elected; and an integer variant \( C \) shows that the election cannot remain active indefinitely. All three \( A, B, C \) can be expressed as simple first-order expressions over the program variables.

The proof obligations are that every possible single step of the algorithm: takes an \( A \)-satisfying state to an \( A \)-satisfying state again; does the same for \( B \); and is guaranteed with some non-zero probability to decrease \( C \) strictly. This suffices to establish termination (election) with probability 1 [16, Lem 2.7.1]. We give the details of \( A, B, C \) elsewhere [18], and examples of similar proofs (Dining Philosophers, Distributed Consensus) can be found in the literature [16, Chap 3].

Recall that \( k \) is the greatest ticket number, so there are \( k + 1 \) possible tickets. In the case \( k=1 \) these obligations are all met; for \( k>1 \) however the proof of \( B \)’s invariance fails and the counterexample is the one earlier found by Fokkink. Our distribution generator was able to confirm the counterexample automatically.

Fokkink’s counterexample schedule involved thirteen steps of the protocol on a three-node ring. First all three processors become active (one after the other) and choose a \( ntk \); with some non-zero probability they receive 2,2,1. The 2’s messages then complete a full circuit, along the way passivating the node with \( ntk \) value 1, and dirtying each other so that their originating nodes try again, re-activating. This time they receive 0,0, and now circulation of the 1-valued message passivates both. All three nodes are now passive.

5 Conclusions and future work

We explored the practical use of counterexample generation in a context of proof-based verification for simple safety properties of probabilistic software. We showed how to use a selection of algorithmic methods to generate or validate such certificates and illustrated the techniques on a range of case studies.

One key contribution is that our techniques are carried out directly at the level of source code. A second key contribution is that we use theorem-proving
Incoming message is processed as follows:

\[
(* \text{if location is active}*)
\]
if \( \text{state}.(\text{loc}.x) = \text{active} \) then

\[
\text{if } \text{hop}.x = N \land \text{bit}.x \quad (* \text{return home clean, so become leader} *)
\]
then \( \text{state}.(\text{loc}.x) := \text{leader}; \)

else if \( \text{hop}.x = N \land \neg \text{bit}.x \quad (* \text{return home dirty} *)
\]
(* so reactivate with new random tk *)
then \( \text{ntk}.(\text{loc}.x) := \text{pChoose}\{0, \ldots, k\}; \)
\( \text{tk}.x, \text{hop}.x, \text{bit}.x := \text{ntk}.(\text{loc}.x), 1, \text{true}; \)

(* not yet home *)
(* so compare tk's *)
else if \( \text{ntk}.(\text{loc}.x) = \text{tk}.x \quad (* \text{get dirty} *)
\]
then \( \text{hop}.x, \text{bit}.x := \text{hop}.x+1, \text{false}; \)

else if \( \text{tk}.x > \text{ntk}.(\text{loc}.x) \quad (* \text{passivate the node} *)
\]
then \( \text{state}.(\text{loc}.x), \text{hop}.x := \text{passive}, \text{hop}.x+1; \)

else if \( \text{tk}.x < \text{ntk}.(\text{loc}.x) \quad (* \text{annihilate the message} *)
\]
then \( \text{loc}.x := \bot; \)

else \( \text{hop}.x := \text{hop}.x + 1; \quad (* \text{location is passive, message goes through} *)
\]

\[\text{Fig. 8. Protocol for asynchronous leader election, without round numbers (assume loc.x is automatically incremented when a message moves on).}\]

and model-checking techniques together, in complementary ways that have not previously been explored for probabilistic systems.

However there is much more work to be done to improve the analysis techniques, both for finding the schedule in the first instance, and afterwards. For example once the strategy has been discovered the result is effectively a Markov Chain as explained above Def. 2 which could be investigated more carefully using methods suggested by Han and Katoen [10].

Others have used SMT-based solvers in model checking contexts [4]. There is also a broad literature on the topic of locating optimal strategies in MDPs relative to \( p\text{CTL} \) properties [6, 5]; it is likely that a combination of several of these methods will deliver the best feedback for a prover.

References


