The Journal of Logic and Algebraic Programming 81 (2012) 26–45

Contents lists available at SciVerse ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Generating counterexamples for quantitative safety specifications in probabilistic B

Ukachukwu Ndukwu [1]

Department of Computing, Macquarie University, NSW 2109 Sydney, Australia

ARTICLE INFO

Article history:
Available online 14 July 2011

Keywords:
Probabilistic B
Expectations
Quantitative safety
Failures
Counterexamples

ABSTRACT

Probabilistic annotations generalise standard Hoare Logic [20] to quantitative properties of probabilistic programs. They can be used to express critical expected values over program variables that must be maintained during program execution. As for standard program development, probabilistic assertions can be checked mechanically relative to an appropriate program semantics. In the case that a mechanical prover is unable to complete such validity checks then a counterexample to show that the annotation is incorrect can provide useful diagnostic information. In this paper, we provide a definition of counterexamples as failure traces for probabilistic assertions within the context of the *pB* language [19], an extension of the standard B method [1] to cope with probabilistic programs. In addition, we propose algorithmic techniques to find counterexamples where they exist, and suggest a ranking mechanism to return 'the most useful diagnostic information' to the *pB* developer to aid the resolution of the problem.

1. Introduction

Following the seminal works of Clarke and Emerson [9], and Queille and Sifakis [35], the use of model checking [10] in systems construction has grown in popularity. However, an interesting recent dimension in program verification is to take advantage of the numerous benefits derived by combining proof-based methods with model checking. One of the advantages of doing this is that, if a set proof goal fails to be satisfied, then model checking can provide necessary intuition in the form of *counterexamples* that explain the failure.

Counterexample generation in model checking have proved to be beneficial for error-free systems construction. It provides a means of summarising the failure of an exact model of a target system (coded in a formal language) to meet its specification (usually expressed in some temporal logic). Consequently, the form of a counterexample is largely dependent on the accuracy of the system model, and that of its underlying logic of specification.

In a large scale proof-based systems construction as in Event-B [3], counterexamples have successfully been used to provide intuition into the failure of a proof-based model to meet a desired specification. This is made possible by incorporating a model checker known as ProB [25] into the formal language development environment *via* an external plug-in. That is, given an Event-B model and a desired specification, ProB can reveal (in linear-time) execution sequences of the model that violate the specification of interest.

More recently, the notion of counterexample generation has been extended to also include probabilistic systems [4–7,14,15]. This class of systems provide more generalisations of properties of their standard (non-probabilistic) counterparts by enabling a verifier to query a probabilistic system model for more interesting properties like 'the expected value of a random variable over a finite number of runs' of the model. Counterexamples for probabilistic system have been investigated for different types of temporal property specifications ranging from linear-time properties [7] to probabilistic branching time

E-mail address: ukachukwu.ndukwu@mq.edu.au

[1] The author acknowledges support from the Australian Commonwealth Endeavour International Postgraduate Research Scholarship (E-IPRS) Fund.

1567-8326/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2011.06.001

properties [15] for discrete system models (a focus of this work). In these cases, counterexamples were captured as a set of finite paths of the probabilistic models that violate the property of interest.

In this paper, we show how to use probabilistic model checking techniques to generate and present counterexamples for quantitative safety properties specified using an assertional-style formalism for program verification in the probabilistic B language [19] in as much as ProB does for Event-B.

Traditionally, a safety property is used to capture the notion of "nothing bad should happen". It conjectures a set of "good states" such that program execution is restricted only to those states. Safety properties defined in this way are known as program invariants — they are conditions over a program space that characterise all the states (including the initial state) where nothing bad should happen. The mechanical check of proof of program correctness then reduces to finding any "bad state" in the program space where program execution is guaranteed to violate the safety property.

Program assertions are predicates which identify conditions under which the program behaves correctly; they are used to capture qualitative "safety properties." More recently they have been extended to quantitative safety properties for probabilistic programs [27]. An example of the use of assertions in probabilistic program development is: "the program remains in the set of good states with probability of at least 90%." But in reality they can be more general than that, and can be used to capture interesting performance-style properties. Assertions are fundamental to the well-known *B* development method [1] in which they can be checked mechanically relative to a program model of the system being developed. Similarly, the use of quantitative assertions have been shown to extend *B* to allow for the development of probabilistic programs. The resulting extended method is known as probabilistic B or *pB* [19] with [27,30] as its underlying logic.

As for traditional safety, the *pB* development environment generalises *quantitative safety* by the use of 'expectations' over random variables. The technique relies on the *pB* developer to suggest appropriate quantitative program invariants using random variables such that the expected value over the random variables must always lie above a specified threshold. The invariants can then be checked automatically — i.e. to ensure that the program's operation sequences do not violate the given threshold. In order to do so, a *pB* mechanical prover sub-tool known as the *proof obligation generator* constructs a number of constraints which if satisfied will imply the inductive definition of the invariants. Each generated constraint is called a *proof obligation*; when the proof obligation generator has (as a separate task) established the satisfaction of a constraint, we say that the proof obligation has been discharged.

Although quantitative assertional-style reasoning for probabilistic systems is very general, it can sometimes fail, resulting in undischargeable proof obligations which leave the *pB* developer with no clear understanding of the cause(s) of failure. But naturally, failure might be traced to either of two possibilities: (a) a wrongly coded model of the system or even (b) an incorrect specification. As a result, one of the challenges of development in *pB* is how to characterise behaviours of a system that demonstrate the failure of a quantitative safety property. We shall propose a formalism where failure to meet some expectations threshold would imply a violation of the safety property. We shall show however, that the failure of inductive invariance amounts to the cause of the violation. Our approach here is to initiate methods using probabilistic model checking techniques to compute useful diagnostic information to help a *pB* designer to identify and fix either of the possible causes of failure. Our aim is to ensure that the final machine for deployment is error-free.

In our precursor paper [32], we described how to formulate a quantitative safety property in *pB* development as a probabilistic model checking problem prior to experimental analysis. Our investigations captured faulty program behaviours with respect to some quantitative safety property of interest. **Our contribution** in this paper is on how to generate and present useful and precise diagnostic feedback to the *pB* developer in order to aid better system construction. We summarise our technique as follows:

- We show how to use a generalisation of the Probabilistic Computational Tree Logic or PCTL [16] similar to bounded model checking techniques [8] over reward structures [23] of Markov Decision Processes (MDPs) [34] to compute counterexamples consisting of only failure traces detailing the violation of a *pB* model safety property. We consider a failure trace as one which leads to a state (and an operation) where the model execution fails to observe its definition of safety inductivity. Failure traces defined in this way are very informative and are generalisations of the failure paths proposed, for example, by Katoen and coworkers [15].
- Based on an MDP interpretation of a faulty *pB* model, we show how to extract what we term 'the most useful diagnostic information' by using a scheduler that points to the closest root-cause of the failure. To do this, we propose a variation of the *k*-shortest path algorithm [12] based on the expectation-transformer semantics of McIver and Morgan [27] to locate the counterexample traces from the scheduler; in the case that there are several traces, we propose a ranking methodology which uses the probability of occurrence of the traces or even their residual expectations to order the counterexamples information.
- Finally, we illustrate the technique showing how to compute the most useful diagnostic information summarising the failure of inductivity of the *pB* model safety property whose preliminary experimental analysis was initiated in [32].

The rest of this paper is structured as follows: Section 2 discusses probabilistic annotations and its application to safety in *pB* development; Section 3 explains *pB* expectations as MDPs; we explain the strategy for counterexample generation in Section 4; Section 5 is the automation and practical demonstration of that strategy; we discuss related work in Section 6; and finally we conclude in Section 7.

1.1. Summary of notation

Function application is represented by a dot, as in $f.x$ (rather than $f(x)$). We use an abstract finite state space S. Given predicate *pred* we write $[pred]$ for the *characteristic* function mapping states satisfying *pred* to 1 and to 0 otherwise, punning 1 and 0 with "True" and "False", respectively. We write $\mathcal{E}S$ as the set of real-valued functions from S, i.e. the set of expectations; and whenever $e, e' \in \mathcal{E}S$ we write $e \Rrightarrow e'$ to mean that $(\forall s \in S.\ e.s \leq e'.s)$. We let $\mathbb{D}S$ be the set of all discrete probability distributions over S; and write $Exp.\delta.e = \sum_{s\in S}(\delta.s) \times e.s$ for the expected value of e over S where $\delta \in \mathbb{D}S$ and $e \in \mathcal{E}S$. Finally we write S^* for the finite sequences of states in S.

2. Probabilistic annotations

When probabilistic programs execute they make random updates; in the semantics, that behaviour is modelled by (discrete) probability distributions over possible final values of the program variables. Given a *nondeterministic* program *Prog* operating over (abstract)[2] state space S we write $[\![Prog]\!] : S \to \mathbb{D}S$ for the semantic function taking initial states to sets of distributions over final states. For example, the program fragment

$$pInc \triangleq s := s{+}1 \ {}_p\oplus\ s := s{-}1 \tag{1}$$

increments (state) variable s with probability p, or decrements it with probability $1{-}p$.

In particular the semantics $[\![pInc]\!]$ [27] maps each initial state s to a probability distribution returning p or $(1{-}p)$ for (final) states $s' = (s{+}1)$ or $s' = (s{-}1)$, respectively. Since each initial state s results in a single distribution $[\![pInc]\!].s$ we say that *pInc* is *deterministic*; when nondeterminism is also present, the result for each initial state would be a set of distributions. Rather than working with this semantics directly, we shall focus on the dual logical view generalising Hoare Logic [20].

Probabilistic Hoare Logic [27] takes into account the probabilistic judgements that can be made about probabilistic programs, in particular it can express when predicates can be established only *with some probability*. However, as we shall see, it is even more general than that, capable of expressing general expected properties of random variables over the program state. We use *Real*-valued annotations of the program variables (interpreted as random variables); a program annotation is said to be valid exactly when the expected value over the post-annotation is at least the value given by the pre-annotation. In standard Hoare triples notation this can be expressed as

$$\{pre\}\ \ Prog\ \ \{post\}, \tag{2}$$

and is valid exactly when $Exp.([\![Prog]\!].s).post \geq pre.s$[3] for all states S, where *post* is interpreted as a random variable over final states and *pre* as a real-valued function.

With the notational convention set out in the previous section, we can give an example of a correct annotation for the program fragment *pInc*

$$\{p[s = -1] + (1{-}p)[s = 1]\} \quad pInc \quad \{[s = 0]\}, \tag{3}$$

which expresses the fact that from initial state $s = -1$, the probability of establishing $s = 0$ is p, and from $s = 1$ it is $(1{-}p)$.

Hoare triples can be checked mechanically using a generalisation of Dijkstra's weakest precondition or *Wp* semantics defined on the program syntax of a simple programming language, as set out in Fig. 1. As for standard *Wp* this formulation allows annotations to be checked mechanically [19,21]; moreover we see that annotation (2) is valid exactly when $pre \Rrightarrow Wp.Prog.post$.

In this paper, we shall concentrate on certifying probabilistic safety expressible using probabilistic annotations. Informally, a probabilistic safety property is a random variable whose expected value cannot be decreased on execution of the program. (This idea generalises standard safety, where the *truth* of a safety predicate cannot be violated on execution of the program.) Safety properties are characterised by *inductive invariants*: for example, the valid annotation $\{Expt \times [pred]\}\ Prog\ \{Expt\}$ says that *Expt* is an inductive invariant for *Prog* provided it is executed in an initial state satisfying *pred*. To illustrate, the annotation

$$\{s\} \quad pInc \quad \{s\}, \tag{4}$$

means that the expected value of s is never decreased.

Inductive invariants will be a significant component of the specifications in our *pB* models, to which we now turn.

2.1. Probabilistic safety in pB

pB [19] is an extension of standard B [1] to support the specification and refinement of probabilistic systems. Systems are specified by a collection of *pB machines* which consist of operations describing possible program executions, together with variable declarations and invariants prescribing correct behaviour.

[2] Here we assume that the state space is abstract until explicitly defined.

[3] Note that we also write $Exp.([\![Prog]\!].s).Expt$ to mean $Wp.Prog.Expt.s$.

Name	*Prog*	$Wp.Prog.Expt$
identity	skip	$Expt$
assignment	$x := f$	$Expt[x := f]$
composition	$Prog; Prog'$	$Wp.Prog.(Wp.Prog'.Expt)$
choice	$Prog \lhd G \rhd Prog'$	$Wp.Prog.Expt \lhd G \rhd Wp.Prog'.Expt$
probability	$Prog\ {}_p\!\oplus\ Prog'$	$Wp.Prog.Expt\ {}_p\!\oplus\ Wp.Prog'.Expt$
nondeterminism	$Prog \sqcap Prog'$	$Wp.Prog.Expt$ min $Wp.Prog'.Expt$
weak iteration	it $Prog$ ti	$\nu X \bullet (Wp.Prog.X$ min $Expt)$

Expectation $Expt$ is of type $\mathcal{ES}$, and $Wp.Prog$ is of type $\mathcal{ES} \to \mathcal{ES}$. Programs are composed of identity statements (skip), assignments (:=), sequential composition (;), Boolean statements (... $\lhd G \rhd$ ... which is a short hand for if G then ... else ... fi where G is a guard), probabilistic choice (${}_p\!\oplus$), nondeterministic choice ($\sqcap$), and weak iteration (it ... ti).

Fig. 1. Structural definition of the expectation transformer-style semantics.

MACHINE	Faulty
SEES	Int_TYPE, Real_TYPE
CONSTANTS	p
PROPERTIES	$p \in REAL \wedge p \geq real(0) \wedge p \leq real(1)$
VARIABLES	cc
INVARIANT	$cc \in \mathbb{N}$
INITIALISATION	$cc := 0$
OPERATIONS	
	OpX $\triangleq$ **BEGIN**
	PCHOICE p **OF** $cc := cc + 1$
	OR $cc := cc - 1$ **END;**
	OpY $\triangleq$ $cc := 0$
EXPECTATIONS	$real(0) \Rrightarrow cc$
END	

Bold texts on the left column capture the fields (or clauses) used to describe the machine. The **PCHOICE** keyword introduces a probabilistic binary operator; the **EXPECTATIONS** clause expresses the notion of probabilistic quantitative safety.

Fig. 2. A simple faulty *pB* machine.

The machine set out in Fig. 2 illustrates some key features of the language. There are two operations — *OpX* and *OpY* — which can update a variable *cc*. *OpX* can either increment *cc* by 1 or decrement it by the same value with probability p or $(1 - p)$, respectively, while *OpY* just resets the current value of *cc* to 0. In general, operations can execute only if their preconditions hold. But in the absence of preconditions as in this case, the choice of which operation to execute is made nondeterministically.

The remaining fields ascribe more information to the variables, constants and behaviour of the operations. Declarations are made in the CONSTANTS and VARIABLES fields, and PROPERTIES and SEES state assumed properties and context of the constants and variables. The INVARIANT field sets out invariant properties. The expression in the INITIALISATION clause establishes the invariant and the operations *OpX* and *OpY* must maintain it afterwards.

We shall concentrate on the EXPECTATIONS clause (boxed in Fig. 2), which was introduced by Hoang [19] to express quantitative invariant or safety properties. The form of an EXPECTATIONS clause is given by

$$E \Rrightarrow Expt, \tag{5}$$

where both E and *Expt* are expectations. It specifies that the expected value of *Expt* should always be *at least* E, where the expected value is determined by the distribution over the state space after any (valid) execution of the machine's operations, following initialisation. An example of the use of this construct is in Fig. 2 where the EXPECTATIONS clause is interpreted as "the expected value of *cc* is always at least 0". Hoang showed that this idea is guaranteed by the following valid annotations:

$$\{E\}\ \ init\ \ \{Expt\} \quad \texttt{and} \quad \{[pred] \times Expt\}\ \ Op\ \ \{Expt\}, \tag{6}$$

where *Op* is any operation with precondition *pred* and *init* is the machine's initialisation. In what follows we shall refer to (6) as the *proof obligations* for the associated expectations clause (5).

Checking the validity of program annotations, and in particular inductive invariants for loop-free program fragments can be done mechanically based on the semantics set out in Fig. 1 [19]. In some cases however, the proof obligation can fail to be discharged. There are two possible reasons for this. The first possibility is that *Expt* is too weak to be an inductive invariant for the machine's operations, and hence needs to be strengthened by finding another inductive $Expt'$ which again satisfies (6) so that $Expt' \Rrightarrow Expt$ and the original safety property can be validated. The second possibility is that a sequence of operations of the machine can be found which explain the probabilistic safety property violation. In this case, providing a counterexample can guide the prover with sufficient insight to correct the error.

In classical B model checking is used to locate a computation trace in terms of operation executions to find a state in which the safety property is clearly violated [25]. Our aim in the next section is to extend this idea to the case of probabilistic machines; we will find that here too a counterexample is a unique trace from the initialisation to a state where the inductive invariant fails to hold, so that similar model checking techniques can be used to locate it. In order to do this we must first interpret the EXPECTATIONS clause above in terms of a model checking problem. We turn to that problem in the next section.

3. *EXPECTATIONS* as probabilistic safety in MDPs

In abstract terms, a *pB* machine can be modelled as a Markov Decision Process (*MDP*) [34] by considering the nondeterministic choices over all its operations. We demonstrate how to use this interpretation to define and present counterexamples to probabilistic safety using model checking techniques for an MDP representation of an abstract *pB* machine. This then allows us to carry over behaviours of MDPs to *pB* machines explicitly, especially in terms of operation sequences of the latter.

We recall that an MDP combines the ideas of nondeterministic and probabilistic executions; we shall then use a formulation in which a probabilistic program P takes an initial state s_0 and outputs a set of probability distributions over final states determining a single step of P's execution [27]. When P executes arbitrarily we shall identify a *computation trace* as a finite sequence of states $\langle s_0, s_1, s_2, \ldots, s_n\rangle \in S^*$ where each (s_i, s_{i+1}) is a probabilistic transition of P, i.e. s_{i+1} can occur with nonzero probability by executing P from s_i, that is to say there is a distribution $\delta \in \mathbb{D}S$ associated with P such that $\delta.s_{i+1} > 0$. This idea is based on a *computation tree* from which the individual traces can be extracted.

As formalised by Hoang [19], we express probabilistic safety specified as an EXPECTATIONS clause (as at (5)) of a *pB* machine. A weak iteration of P given by $\mathsf{it}\ P\ \mathsf{ti}$, (as defined in Fig. 1) allows an arbitrary execution of any operation in P, and as such expresses the possible computation traces so that the EXPECTATIONS clause given by (5) is satisfied exactly when

$$\{E\} \quad (s := s_0;\ \mathsf{it}\ (P \sqcap \mathsf{skip})\ \mathsf{ti}) \quad \{Expt\} \tag{7}$$

is valid, where s is a state variable initialised to s_0 (see [19]). Our next task is to reformulate (7) in terms of finite state exploration, giving access to an investigation of this safety property using probabilistic model checking.

3.1. Distributions over execution traces

Intuitively a distribution over execution traces is a computation tree recording the set of possible execution traces together with their associated probabilities. The situation becomes more complicated in the presence of nondeterminism, where a set of computation trees can be associated with a single program, capturing the effect of the demonic choices.

Definition 1. Given a program P, an *execution schedule* is a map $\aleph : S^* \to \mathbb{D}S$ so that $\aleph.\alpha \in [\![P]\!].s$ picks a particular resolution of the nondeterminism in P to execute after the trace α, where s is the last item of α. (A more uniform formalisation would give the distribution of initial states as $\aleph.\langle\rangle$; but we prefer to give initial states explicitly.) Note that where necessary, we shall simply refer to $\aleph$ as a scheduler, to conform to standard terminology.

We can now formalise *probabilistic computation trees* using 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 [13,26].

Definition 2. Given a program P, initial state s_0 and scheduler $\aleph$, we define the corresponding trace distribution $\langle\!|P_\aleph|\!\rangle.s_0$ of type $S^* \to [0, 1]$ to be

$$\langle\!|P_\aleph|\!\rangle.s_0.(s') \triangleq 1 \text{ if } s' = s_0 \text{ else } 0$$
$$\text{and } \langle\!|P_\aleph|\!\rangle.s_0.(\alpha s s') \triangleq \langle\!|P_\aleph|\!\rangle.s_0.(\alpha s) \times \aleph.(\alpha s).s'$$

We can recover *Wp*-properties from computation trees by focussing on distributions over endpoints. If we take k steps from some s_0 according to the scheduler $\aleph$, then the probability of ending in state s' is given by

$$[\![P_\aleph^k]\!].s_0.s' \triangleq \sum_{|\alpha|=k} \langle\!|P_\aleph|\!\rangle.s_0.(\alpha s').$$

We write $(P \sqcap \mathsf{skip})^k$ for the program $P \sqcap \mathsf{skip}$ iterated k times; its output from an initial state s_0 is a set of distributions $[\![P_\aleph^k]\!].s_0$ over S for all possible schedulers $\aleph$. It is a standard property of finitary MDPs that there is an *extremal scheduler* $\aleph$ such that $Exp.([\![P_\aleph^k.s_0]\!]).Expt = Wp.(P \sqcap \mathsf{skip})^k.Expt$.

The importance of these definitions is that they give the explicit link between trees and state-based properties, ensuring consistency between them — indeed if a safety property fails to hold then it must be the case that there is a computation tree which will witness that fact. The next theorem gives alternative ways to express a safety property.

Theorem 1. *Let p be a real value, s_0 an initial state, P a probabilistic program, and Expt an expectation. The following statements are equivalent.*

(a) *The triple* $\{p\}\ \ (s := s_0;\ \text{it}\ (P \sqcap \text{skip})\ \text{ti})\ \ \{Expt\}$ *holds;*
(b) *The inequality* $Exp.([\![P^k_{\aleph}]\!].s_0).Expt \geq p$, *for all* $k \geq 0$ *and schedulers* $\aleph$*;*
(c) *There exists a strongest expectation Expt′ such that* $Expt' \Rrightarrow Expt$ *and Expt′ is an inductive invariant satisfying* $Expt'.s_0 \geq p$.

Proof. After program initialisation we see that (a) is equivalent to (b) by definition of $Wp.(\text{it}\ (P \sqcap \text{skip})\ \text{ti}).Expt$ as a limit of $Wp.(P \sqcap \text{skip})^k.Expt$ [29]; (a) is equivalent to (c) since $Expt' \triangleq Wp.(\text{it}\ (P \sqcap \text{skip})\ \text{ti}).Expt$ is the greatest inductive invariant stronger than *Expt*. □

Theorem 1 (b) expresses how safety relates to finite computation trees and (c) how it relates to state-based properties of the kind used in *pB*. In particular case (b) shows us how to refute the safety property: we must exhibit some finite *k* and a scheduler $\aleph$ such that its corresponding expected value of *Expt* over endpoints defined by $[\![P^k_{\aleph}]\!]$ falls below the specified threshold.

Definition 3. Let *p* be a real value, s_0 an initial state, *P* a probabilistic program, and *Expt* an expectation. A *counterexample* to the safety property

$$\{p\}\ \ (s := s_0;\ \text{it}\ (P \sqcap \text{skip})\ \text{ti})\ \ \{Expt\}$$

is an integer $k > 0$ and a scheduler $\aleph$ such that

$$Exp.([\![P^n_{\aleph}]\!].s_0).Expt < p.$$

We call the associated distribution over paths, i.e. $\langle\!|P^k_{\aleph}|\!\rangle.s_0$ a *failure tree* at *k*.

One of the problems of this kind of evidence for failure is that distributions are very complicated to digest, and if taken as a whole, are not very informative as to the cause of the failure of the safety property. It is a challenge to present the information in such a way that reasons for the failure can be more succinctly identified.

Classical model checking techniques (involving no probability at all) are able to identify single traces leading to an explicit failure of the property. Katoen and coworkers [15] has shown that for some probabilistic properties it is possible to use a subset of the traces as evidence of refutation. Our aim is to find a similar simple presentation both for generalised failure and for failure of *Expt* to be an inductive invariant. In fact the latter is a much simpler kind of failure than general failure of safety as, like violation of standard failures, it can be evidenced by a single trace.

Definition 4. Given a program *P*, initial state s_0 and scheduler $\aleph$ and expectation *Expt*, we say that trace αs is a *failure trace* for *Expt* and $\aleph$ if

$$\langle\!|P_{\aleph}|\!\rangle.s_0.(\alpha s) > 0 \quad \textit{and} \quad Wp.P.Expt.s < Expt.s.$$

Note that this definition corresponds to the standard definition of a counterexample trace illustrating a path to a failure of a safety property: when *Expt* is standard (i.e. a characteristic function $[G]$ representing the "good" states *G*) and *P* contains no probabilistic transitions, if $Wp.P.[G].s < [G].s$ then it must be the case that *s* satisfies *G* is true but $Wp.P.[G].s$ is false. In other words that there exists a transition of *P* from *s* to the set of "bad" states.

It turns out that failure traces, taken together, play a significant role in explaining overall failure of a safety property. Our first result says that if the general threshold for safety fails to be met, then failure traces must exist.

Corollary 1. *Let p,* s_0*, Expt and P be as defined in Theorem 1 above. If*

$$Wp.(\text{it}\ P \sqcap \text{skip}\ \text{ti}).Expt.s_0 < p,$$

then there exists a trace α *and a scheduler* $\aleph$ *such that* α *is a failure trace for* $\aleph$ *and Expt.*

Proof. Suppose for contradiction that there is no such trace. Then we may conclude that the expectation *Expt* satisfies the inductive property relative to *P* for all states "reachable" with some positive probability. Moreover it must satisfy $Expt.s_0 \geq p$, by applying this assumption to the empty trace. Now define $Expt' \Rrightarrow Expt$ as the restriction of *Expt* to all reachable states; *Expt′* is inductive by construction and so we apply Theorem 1 (c) to conclude that the safety property must be satisfied. □

Unlike classical model checking, a single example of a failure trace is not sufficient to guarantee a general failure to meet the threshold (unless of course that threshold is 1). But it does indicate that *Expt* is not inductive, and needs to be

strengthened at precisely those reachable states which do not satisfy the inductive property for *Expt*. Indeed if it cannot be strengthened then there must be a failure tree.

In the special case that the expectation *Expt* is standard it has been shown by Katoen and coworkers [15] that rather than producing the whole failure tree, it is possible to have a restriction to only those paths which have a non-zero probability of leading to unsafe states — i.e. those states violating the safety condition such that their total probability lies above the specified threshold. A generalisation of this idea in terms of expectations is to say that paths which lead to "unsafe states" are precisely those which lead to a failure of inductivity. Unfortunately there does not seem to be a sufficiently simple generalisation of that complete criterion for our general safety properties. There is however a sound rule which is similar in flavour to Katoen's result.

We say that a state *s fails inductivity* with respect to *Expt* and *P* if

$$Wp.(P \sqcap \mathsf{skip}).Expt.s < Expt.s.$$

Note from Definition 4 that a failure path ends in such a state. We shall refer to states such as *s* as *witnesses* to failure.

With this definition, the next lemma shows that if there is a reduced collection of failure paths whose expected value over the difference of *Expt* and *Wp*.*P*.*Expt* is sufficiently large then this is enough to evidence general failure of safety.

Lemma 1. *Let P be a probabilistic program, s_0 an initial state, and Expt an expectation and F be the set of reachable states which fail inductivity. If there is some k and set of states $G \subseteq F$ such that*

$$Expt.s_0 - Wp.(P \sqcap \mathsf{skip})^k.([G]\times(Expt.s_0 - Wp.P.Expt.s_0)) < p.$$

then

$$Wp.(P \sqcap \mathsf{skip})^{k+1}.Expt.s_0 < p.$$

Proof. Note that for states $s \in G$, $Wp.P.Expt.s < Expt.s$ and now we reason:

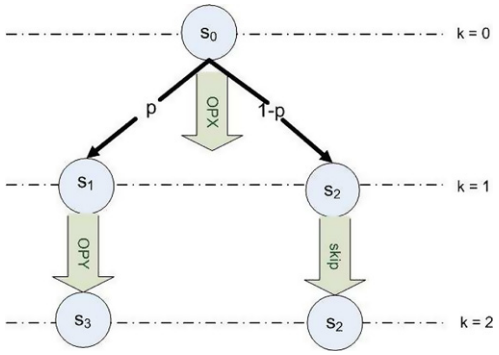
$$\begin{array}{lll}
& p & \\
> & Expt.s_0 - Wp.(P \sqcap \mathsf{skip})^k.([G]\times(Expt.s_0 - Wp.P.Expt.s_0)) & \text{Assumption} \\
\geq & & \text{“}Wp\text{-calculus”} \\
& Wp.(P \sqcap \mathsf{skip})^k.Expt.s_0 - Wp.(P \sqcap \mathsf{skip})^k.([G]\times(Expt.s_0 - Wp.P.Expt.s_0)) & \\
\geq & & \text{“sublinearity”} \\
& Wp.(P \sqcap \mathsf{skip})^k.(Expt.s_0 - ([G]\times(Expt.s_0 - Wp.P.Expt.s_0))) & \\
\geq & & \text{“arithmetic”} \\
& Wp.(P \sqcap \mathsf{skip})^k.((1-[G])\times Wp.(P \sqcap \mathsf{skip}).Expt.s_0 + [G]\times Wp.(P \sqcap \mathsf{skip}).Expt.s_0) & \\
\geq & & \text{“arithmetic”} \\
& Wp.(P \sqcap \mathsf{skip})^k.(Wp.(P \sqcap \mathsf{skip}).Expt.s_0) & \\
= & Wp.(P \sqcap \mathsf{skip})^{k+1}.Expt.s_0. &
\end{array}$$

Given expectations X and Y, sublinearity means that $Wp.P.(X - Y) \Rrightarrow Wp.P.X - Wp.P.Y$ *provided* $X - Y \Lleftarrow 0$. □

Note that Lem.1 is sufficient and thus is a restriction to failure paths. Next we illustrate the relationship between failure traces and general failure in the example that follows.

Example 1. Fig. 3 shows a failure tree representing a chosen scheduler-specific instance $\mathcal{D}$ of the pB machine construction in Fig.2. Its subtrees are captured by truncating the tree at finite *k* steps from its root; the branching probability *p* is 0.5. Table 3.1 gives the expectation transformer analysis of every state in the endpoint of a given finite sub-distribution. The initialisation *INIT* preserves the lower bound of the random-variable inductive invariant cc. Similarly, at step $k = 0$ (after the initialisation), the lower bound of cc is also preserved since after the operation OPX the expected value of cc is not decreased. However, the same cannot be said at the step $k = 1$. The operation OPY (afterwards) strictly decreases the invariant, and that corresponds to an overall failure of the safety condition $Wp.(OPX \sqcap OPY \sqcap \mathsf{skip})^2.cc$. Clearly, it has a distribution over its endpoint that similarly decreases the expected value of the invariant. This is an indication that there is a problem at at step $k = 1$ for a worst-case failure tree depicted in Fig. 3. Consequently, the state s_1 is a witness to the failure, and the first faulty execution trace of the machine is rightly given by the operations-state pair sequence: {INIT} (0), {OPX} (1), {OPY} (−1); and this can occur with probability 0.5.

Clearly, a failure of the proof obligation for safety is given by the witness s_1 evidencing the fact that $Wp.OPY.cc.s_1$ $(= 0) < cc.s_1$ $(= 1)$, corresponds to a violation of the threshold specified for the assertion in the EXPECTATIONS clause.



The parameter k keeps track of the depth of the computation in terms of the *pB* machine operations. For the interpretation of safety we also include `skip` as a valid default operation. The heavy arrows denote *pB* operations corresponding to the transitions; where the transitions are probabilistic, the probabilities appear as labels on the bold arrows. Finally we name the individual states s_0, s_1, ..., s_n while the value of the *pB* variable *cc* appears as an annotation.

Fig. 3. A failure tree describing a scheduler of the *pB* machine of Fig. 2.

Table 1
Analysis of the failure tree in Fig. 3.

Step (k)	*cc.s*	*Wp.OP.cc*	$Exp.[\![P_{\mathcal{D}}^{k+1}]\!].s_0.cc$
INIT	0	$Wp.INIT.cc = 0$	0
k = 0	$0\ (= cc.s_0)$	$Wp.OPX.cc.s_0 = 0$	0
k = 1	$1\ (= cc.s_1)$	$Wp.OPY.cc.s_1 = 0$	−0.5

The parameter k is the depth of the unfolding of the iteration. Note that the overall evaluation of safety is given by the last column, i.e. the expected value of *cc* must be at least 0, as specified by the EXPECTATIONS clause. At $k = 1$ however, the expected value falls to -0.5. Translated to proof obligations, safety becomes a check that the value of the third column must be at least the value computed in the second column. We see that failure of overall safety (at $k = 2$) corresponds exactly to a failure of the proof obligation at step $k = 1$ where the expected value is strictly decreased, i.e. $Wp.OPY.cc.s_1\ (= 0) < cc.s_1\ (= 1)$.

In this section, we have shown the role played by failure traces in the general failure of probabilistic safety. Our next task is to show how to present failure trees for *pB* machines. We use a ranking mechanism, arguing that failure traces provide the most useful diagnostic results. To do this, we use the PRISM model checker [17,33] to encode the *pB* machines and the corresponding EXPECTATIONS clauses, and then interpret PRISM's output as a failure tree from which traces could be extracted and ranked.

In summary, we extract and present counterexamples for an abstract *pB* machine as follows:

1. Translate the *pB* machine as its equivalent PRISM model, together with an expression of the EXPECTATIONS clause as a "reward structure".
2. Determine the least k for which there is a failure using PRISM's experiment facilities.
3. Extract an extremal scheduler from PRISM corresponding to a failure tree upon which we perform further algorithmic analysis.
4. Compute the set of failure traces from the resultant failure tree.
5. Rank the failure traces by their probabilities or other appropriate metrics of interest.

Automation of the first two steps has been described elsewhere [32] thus we only summarise their constructions.

We also note that it might be that such a k, whilst it exists, will not be found by this method if the computation resources prove to be insufficient for the task. In the next section we summarise the first two automation tasks which together use PRISM's engine to indicate the presence of failure trees. Thereafter we explain the last three tasks while presenting algorithmic techniques to aid their automation.

4. Extraction and presentation of counterexamples

The essential idea is to locate failure trees by successively exploring $Wp.(P \sqcap \mathsf{skip})^k.Expt.s_0$ for increasing values of k. This is possible by Theorem 1. But to do this, we first interpret an abstract *pB* machine as a PRISM model, encoding the former's EXPECTATIONS clause as the latter's reward structure [23]. In general, rewards are accumulated along paths; but we enforce no accumulation until a special transition fires in a state s, where we evaluate $Expt.s$ to correspond to the reward in that state. This finitary unwinding will be tracked by a "counter" module whose role is to record the transition steps due to execution of the machine's operations.

4.1. Generating PRISM models of abstract pB machines

The PRISM language is based on the guarded commands formalism extended with a probabilistic choice update. This allows the interaction of probability and nondeterminism in an abstract *pB* machine to be captured as an MDP in the tool. The definition below sets out that idea.

Definition 5. A PRISM model description of an abstract *pB* machine $\mathcal{A}$, is a tuple given by $P = \langle var(P), \texttt{sys}, \{MM, CM\}, Expt(val(P))\rangle$ consisting of a finite set of (Boolean or Integer) variables $var(\mathbf{P})$, a system definition `sys` over valuations $val(\mathbf{P})$ of $var(\mathbf{P})$; A finite set of modules $\{MM, CM\}$, where *MM* is the main module, and *CM* is the counter module. The system definition `sys` is a process-algebraic expression containing *MM* and *CM* exactly once. Finally, $Expt(val(\mathbf{P}))$ is the expectation of a random variable constructed from $val(\mathbf{P})$ using **P**'s reward structure. The main module *MM* encapsulates the operations defined within $\mathcal{A}$. It consists of:

- A finite set of local variables, i.e. $var(MM) \subseteq var(P)$ such that:
  - $var(MM)$ are disjoint from the local variables of *CM*
  - each variable $v \in var(MM)$ has initial value $init(\mathrm{v})$
  - $init(MM)$ denotes the initial values of the variables in $var(MM)$
- A finite set of commands $com(MM)$ describing the statements of the individual operations of $\mathcal{A}$, where each $cmd \in com(MM)$ includes:
  - a guard $\texttt{gd}(cmd)$ which is a Boolean function over $val(var(\mathbf{P}))$
  - an action $\texttt{act}(cmd)$ label which is the name of each operation of $\mathcal{A}$
  - a finite set of update statements $\texttt{updates}(cmd) \triangleq \{\langle\lambda_1, u_1\rangle, \ldots, \langle\lambda_n, u_n\rangle\}$ such that for each $\lambda_i \in (0, 1]$, u_i is a function from valuations over $var(\mathbf{P})$ to the valuations over $var(MM)$ and moreover $\sum_{i=1}^{n}\lambda_i = 1$;

  The syntax of a typical PRISM update statement is:

$$[act]\, gd \rightarrow \lambda_1 u_1 + \cdots + \lambda_n u_n$$

- The counter module *CM* is similarly defined in the same way as the main module *MM* but we restrict its set of local variables such that $var(CM) = \{\texttt{terminate}, \texttt{count}, \texttt{action}\}$ where
  - `terminate` is of type Boolean and *init* (`terminate`) = **false**;
  - `count` is of type Integer and `count`: [0, `MAXCOUNT`] and *init* (`count`) = 0, where `MAXCOUNT` is a model Integer constant;
  - `action` is of type Integer and `action`: [0, `TNA` + 1] and *init* (`action`) = 0 where `TNA` is the total number of actions of $\mathcal{A}$.
- In addition, for each update statement of *MM*, there is an update of *CM* that synchronises with it. Those updates of *CM* can only update the `action` variable and increment the `count` variable by 1. Also, *CM* must contain two extra update statements: a similar unsynchronised update statement[4] that can only update the `action` variable and increment the `count` variable by 1; and a T-labelled update statement that can only reset the `terminate` variable to **true** whenever the guard (`count = MAXCOUNT`) holds. We shall discuss the usefulness of the T-labelled transition later on. Note that successively incrementing the integer value `MAXCOUNT` provides a means of unwinding computation trees by increasing the depth of computation.

Next we shall describe how to set up the main module of an abstract *pB* machine using the transformation rules below.

4.2. Translating pB syntax

The *pB* translation to PRISM is straightforward and essentially involves constructing the guards to each guarded command and the corresponding updates of each PRISM language statement. The idea is to encapsulate the essential behaviours of a *pB* machine in the PRISM main module. Given the structure of the main module in Definition 5, we can achieve this transformation for an abstract *pB* machine $\mathcal{A}$, as follows:

- PRISM constants list: will be constructed from $\mathcal{A}$'s parameter list (if any) and its CONSTANTS clause. The type of a constant is implicitly checked from the PROPERTIES clause.
- PRISM formula list: will be generated as *atomic* predicates from $\mathcal{A}$'s PROPERTIES and INVARIANTS clauses.
- PRISM module name: will be $\mathcal{A}$'s name.
- PRISM variables declaration and initial values list: will be constructed from each variable in the VARIABLES clause, its type in the INVARIANT clause, and its initial values from the INITIALISATION clause. The lower and upper limits of the variables are, respectively, the default lowest values of their types, and a bound specified from in PRISM constants list (above).

[4] This update statement is analogous to the skip programme.

```
rewards
  [T] (count = MAXCOUNT) : Expt
endrewards
```

Fig. 4. Encoding *Expt* as a PRISM reward allows us model check *pB* machines using PCTL reward specifications of the form: $R_{min \geq E.s_0}[F\ \texttt{terminate}]$ where $E.s_0$ is the safety threshold. Note that this reward is computable only after the T-labelled transition fires thus setting the future predicate `terminate` to **true** whenever `(count = MAXCOUNT)` holds.

- PRISM statements: each update statement is labelled with the distinct operation names from $\mathcal{A}$'s OPERATION clause. In addition,
  (a) its guard is inherited from the guard of the operations in $\mathcal{A}$'s OPERATIONS clause and strengthened by the formulas in the PRISM formula list, such that
  (b) the choice of formula selection is dependent on the expressions in $\mathcal{A}$'s update statement. For each update, we check that the formula-dependent expressions are included in the PRISM guard.

4.3. Encoding EXPECTATIONS clause as PRISM reward

The PRISM model checker permits models to be augmented with information about rewards. A reward structure essentially assigns a non-negative real value worth to a state of a DTMC — an MDP whose nondeterminism has been resolved by some scheduler. The tool can then analyse properties which relate to the expected value of the rewards if specified in the temporal logic PCTL [16]. To further help us explore the usefulness of the T-labelled transition of the counter module we set out the definitions below:

Definition 6. A transition reward is assigned to transitions of a DTMC by defining the reward function $\underline{l} : S \times S \to \mathbb{R}_{\geq 0}$. For example, the transition reward $\underline{l}(s, s')$ is acquired each time a transition is enabled from state s to state s' for all $s, s' \in S$.

Definition 7. The reward specification $R_{min \sim E.s}[F\ \Phi]$ is true if from a state s the minimum expected reward accumulated before reaching a state satisfying the *future* predicate Φ meets the bound $\sim E.s$. We note here that R is a computed reward value, $\sim \in \{<, \leq, \geq, >\}$, $E.s \in \mathbb{R}_{\geq 0}$, and Φ is a PCTL state formula.

For an MDP, the reward specification in Definition 7 [5] enables us to inspect the expected values of the rewards accumulated in some future time over computation trees. To do this, we encode a T-labelled transition reward in our model using the PRISM **rewards** . . . **endrewards** keywords (as in Fig. 4). Thus we can compute $Wp.(P \sqcap \text{skip})^{\texttt{MAXCOUNT}}.Expt.s_0$ by defining a reward such that the instantaneous reward is always 0, whilst there is no accumulation part for any other transitions except on the last step when the T-labelled transition fires. This fully defines $Expt(\text{val}(\mathbf{P}))$ for our model as contained in Definition 5.

Finally, Fig. 5 illustrates this transformation — revealing the main module, its counter module and the reward structure for the *pB* machine in Fig. 2. Recall that the unlabelled action corresponds to the "skip" programme (see Definition 5). The complete algorithmic description to enable this transformation is set out in Fig. 13 of the Appendix.

In the next section, we shall explain more practical details on how we can obtain a scheduler violating reward specifications of the type we have discussed so far, i.e. with respect to PRISM's reward structures.

4.4. Extracting an extremal scheduler

An extremal scheduler is a best (or worst-case) deterministic scheduler of the PRISM representation of an abstract faulty *pB* machine — i.e. one whose probability (or reward) of reaching a state where our intended reward specification is violated is maximal (or minimal). On model checking the machine using PRISM's sparse engine, [6] PRISM outputs this adversary in a file named 'adv.tra' [24]. The file represents the resolution of the nondeterminism of a PRISM MDP which achieves the extremal result.

But to do this, we first use the experiment facilities of PRISM to establish that indeed there is some `MAXCOUNT` (the least one would do) for which the reward specification $R_{min \geq E.s_0}[F\ \texttt{terminate}]$ is not satisfied, that is, the safety threshold $E.s_0$ has been violated. This can simply be done by using PRISM to explore the model — i.e. increasing `MAXCOUNT` while verifying the specification until the safety threshold becomes violated. Thereafter, for that value of `MAXCOUNT`, PRISM outputs the adversary file. However, to select the extremal scheduler, we analyse probabilistic transition matrices constructed from the file using an additional *state-value.txt* file from the model's state space. The *state-value.txt* file marks every state of the matrix with (i) the valuation of the program's variables occurring in the reward structure, and (ii) a corresponding action that is

[5] Note that PRISM can only generate adversaries violating properties of this type.
[6] As at the time of exploring this technique, only the sparse engine of the PRISM tool has been implemented to output the adv.tra file.

```
const                 MIN;
const                 MAX;
const                 MAXCOUNT;

module                Faulty
cc : [MIN..MAX]       init 0;

[OPX]                 (MIN < cc < MAX) → 0.5 : (cc′ = cc + 1) + 0.5 : (cc′ = cc − 1);
[OPY]                 true → (cc′ = 0);
endmodule

module                Counter
count:[0..MAXCOUNT]   init 0;
terminate : bool      init false;
action:[0..3]         init 0;

[OPX]                 (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 1);
[OPY]                 (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 2);
[]                    (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 3);
[T]                   (count = MAXCOUNT) → (terminate′ = true);
endmodule

rewards
[T]                   (count = MAXCOUNT) : cc + MAXCOUNT;
endrewards
```

Fig. 5. This encoding allows us to model check the reward specification $R_{min\geq 0}$ [F `terminate`]. However, if this specification fails to hold, we can then proceed with obtaining an extremal scheduler that demonstrates the failure. Note that the padding with *MAXCOUNT* is to ensure that the PRISM engine is consistent with computing positive rewards. Finally, we can achieve consistency by subtracting this parameter from the PRISM computed reward value.

enabled therein. By so doing, we establish an extremal scheduler as one that minimally (with respect to rewards) and maximally (with respect to probability) violates the threshold for safety for the given machine expectation *Expt*. As an example, we note that the deterministic schedule in Fig. 3 corresponds to the extremal scheduler of the faulty *pB* machine in Fig. 2.

Since an extremal scheduler now corresponds to a failure tree, next we present a combination of algorithmic techniques that will enable the presentation of precise diagnostic information sufficient to explain a violation of the inductive safety specifications embedded within a *pB* machine EXPECTATIONS clause.

4.5. Computing failure traces

Following the construction of an extremal scheduler, we discuss a K-constrained k-shortest path algorithm, a variant of Jiménez and Marzal's REA [22] for identifying the faulty execution traces in the resultant failure tree. A key feature of the algorithm is the use of the expectation transformer semantics for locating the faulty traces themselves. We recall from Corollary 1 that any trace ending in the successor of a witness is an ideal candidate for a counterexample (i.e. if traced back to the root of the tree from that successor). Therefore, summarising this semantic interpretation in a C-labelled procedure (see Fig. 6), we have the following problem:

> Traverse the depth of the failure tree and locate all the witnesses to failure — i.e. find states like s such that $Wp.P.Expt.s < Expt.s$.

In the traversal the name of the original *pB* operation causing the transition is noted as well as the probability with which the transition occurs. This allows a failure trace as a sequence of *pB* operations, together with the probability of the sequence to be returned to the *pB* developer (as in Example 1).

In detail, the procedure in Fig. 6 sets out a recursive strategy for computing the failure traces violating the property of interest for a given extremal scheduler corresponding to a failure tree. It employs a breadth-first recursive search of all the states at a given depth of a failure tree. On visiting any state s at a given depth k, we evaluate $Wp.(\alpha.s \sqcap skip).Expt.s$ for a machine operation α that is enabled at s and compare the computed value to its previous value $Expt.s$. Therefore, any state s at that depth which strictly decreases its previous value as a result of the operation execution is then considered a witness to the failure.

On the other hand, Fig. 11 in the Appendix shows an algorithmic representation of the procedure to compute the most useful diagnostic information for any given failure tree after the parameters k and K are initialised prior to the search. The idea is that if we apply the algorithmic procedure over an extremal scheduler interpreted as a transition probability matrix, then we can always compute all the failure traces at a minimal depth k of the tree.

4.6. Ranking faulty execution traces

The notion of ranking the faulty execution traces constituting the counterexamples will further make more sense to our idea of the most useful diagnostic information required to debug a *pB* machine. The need to do this cannot be overemphasized especially since the number of failure traces themselves may grow in the size of the state space of the faulty *pB* machines.

C.0 Fix the value of $K > 0$ before making an entry into the algorithm;
C.1 Set $k \leftarrow 0$ on the assumption that the initial state must preserve E;
C.2 **if** $k < K$
loop
if (s is unmarked) **then**
compute $Wp.(\alpha.s \sqcap$ **skip**$).Expt.s$, where s is the last state of w_k and $w_k \in paths^{\mathfrak{N}^*}_{\mathcal{D}}(0)$;
if (s is a witness) **then** go to C.3;
else next s;
endloop
set $k \leftarrow k + 1$ and repeat C.2;
else go to C.4;
C.3 return prefix containment-free failure trace from the immediate successor(s) of s and mark s;
continue;
C.4 Enter $K' > K$ and go to C.0;

We assume that $\mathfrak{N}^*$ is an extremal scheduler to the original problem; $paths^{\mathfrak{N}^*}_{\mathcal{D}}(0)$ is the set of paths of the failure tree. Every state s of the probabilistic failure tree is labelled with an action $\alpha.s$ enabled at some depth k of the tree, where $Expt$, is the expectation for the given expression $E \Rrightarrow Expt$ embedded within the pB machine's EXPECTATIONS clause. Note here that k and K are analogous to the `count` variable and `MAXCOUNT` respectively.

Fig. 6. A K-contrained k-shortest path procedure.

Fig. 12 I in the Appendix is an algorithmic interpretation for ranking the failure traces. Unlike standard model checking we are also able to rank the failure traces in order of importance defined by appropriate metrics based on probability of occurrence or residual expected values, i.e. $Wp.P.Expt.s - Expt.s$. Failure traces with high probability masses are possibly more useful for debugging compared with ones with low probability masses. Dually, failure traces whose expected values deviate minimally from their thresholds would usually constitute more useful diagnostic information than traces that deviate maximally.

5. Automation and experiments

Given the algorithmic techniques and strategies discussed in the previous section, we have produced a prototype system nicknamed YAGA which provides their implementation. We briefly describe YAGA in this section while referring readers to [31] for a complete description.

5.1. YAGA: implementing the algorithmic techniques

YAGA [31] implements the collection of algorithms in the Appendix. It is a suite of programs for the performance analysis of probabilistic systems development in the *pB* language. Most importantly, it allows a *pB* machine designer to explore faulty machine behaviours experimentally in order to ascertain the cause(s) of failure.

YAGA inputs a faulty *pB* machine violating a specific safety property expressed in its EXPECTATIONS clause, and generates its equivalent MDP representation in the PRISM language. On model checking the resultant PRISM file using the technique discussed in the previous section, it constructs a transition probability matrix from an extremal scheduler output from the PRISM tool.

Finally YAGA analyses the resultant failure tree using the technique set out in Fig. 6 to generate the most useful diagnostic information composed of finite execution traces as sequences of actions and their state valuations leading from the initial state of the *pB* machine to states where the property is violated.

5.2. Case study: faulty probabilistic library revisited

In [32] we used the translator facility of YAGA, and the PRISM tool to locate a faulty behaviour in a *pB* machine (shown in Fig. 7) which captures the basic operations underlying the accounting package of a library system [18]. In that paper, we formulated a reward property which enabled us to verify a safety feature of the machine. Moreover, our experimental investigations revealed that the inclusion of a demonic "StockTake" operation (described within Fig. 7) in the library machine was responsible for the violation of the specified safety feature. In this paper, we revisit that library example and further explain the reasons behind the failure of the machine to observe its specified inductive safety property, i.e. after the operation inclusion.

The state of the machine has four variables: *booksInLibrary*, *loansStarted*, *loansEnded* and *booksLost* which are, respectively, used to keep track of: the number of books in the library, the number of book loans initiated by the library, the number of book loans completed by the library, and the number of books possibly never returned to the library.

Initially, the machine has two operations: *StartLoan*, to initiate a loan on a book, and *EndLoan*, to terminate the loan of a book. The *StartLoan* operation has a precondition that there are books available for loan; it decrements *booksInLibrary* and increments *loansStarted*. When a book is returned, the *EndLoan* operation reverses the effect of the *StartLoan* operation by recording that either the book "really is" returned, or is actually reported lost with some probability *pp*, so that *booksLost* is incremented.

MACHINE *ProbabilisticLibrary (totalBooks, cost)*

SEES *Real_type*

CONSTANTS *pp*

PROPERTIES $pp \in REAL \ \wedge \ pp \leq real\ (1) \ \wedge \ real\ (0) \leq pp$

VARIABLES *booksInLibrary, loansStarted, loansEnded, booksLost, totalCost*

INVARIANT *booksInLibrray, loansStarted, loansEnded, booksLost, totalCost* $\in$ NATURAL $\wedge$ *loansEnded* $\leq$ *loansStarted* $\wedge$
booksInLibrary + booksLost + loansStarted - loansEnded = totalBooks

EXPECTATIONS $real\ (0) \Rrightarrow pp \times real\ (loansEnded) - real(booksLost)$

INITIALISATION *booksInLibrary, loansStarted, loansEnded, booksLost, totalCost*
:= totalBooks, 0, 0, 0, 0

OPERATIONS

StartLoan =
PRE *booksInLibrary > 0* **THEN**
booksInLibrary := booksInLibrary - 1 ||
loansStarted := loansStarted + 1
END;

EndLoan =
PRE *loansEnded < loansStarted* **THEN**
PCHOICE *pp* **OF** *booksLost := booksLost + 1*
OR *booksInLibrary := booksInLibrary + 1* **END;** ||
loansEnded := loansEnded + 1 **END**;

StockTake =
BEGIN
totalCost := cost $\times$ *booksLost* ||
booksInLibrary := booksInLibrary + booksLost ||
loansStarted := loansStarted - loansEnded ||
loansEnded := 0 || *booksLost := 0*
END;

END

Fig. 7. A *pB* machine describing the basic operations of a library system.

The inclusion of the StockTake operation is to enable a library accountant do a periodic bookkeeping of library transactions. An appropriate specification of the library machine is the nondeterministic choice

$$Library \ \triangleq \ \text{it}\ (StartLoan \sqcap Endloan \sqcap StockTake)\ \text{ti}\ .$$

In the sections that follow, we shall investigate this machine design with respect to its EXPECTATIONS clause (boxed in Fig. 7). That is, we shall assume that the machine provides an accurate representation of the library functionality and only wish to investigate the inductiveness of the expression in its EXPECTATIONS clause with respect to the design.

5.2.1. Machine's invariant inductive?

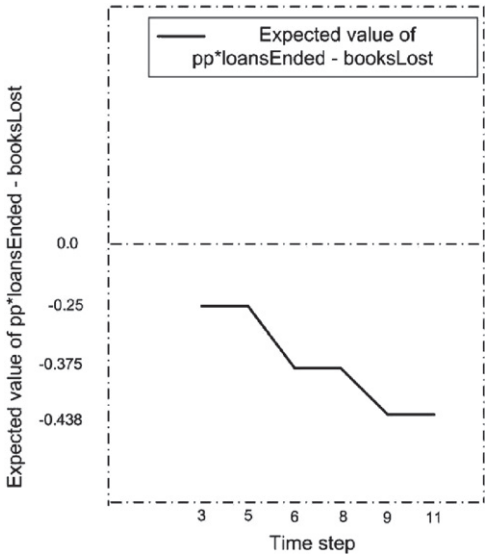
The machine's EXPECTATIONS clause uses the random variable $pp \times loansEnded - booksLost$ to define an inductive invariant expression, i.e. $Expt = pp \times loansEnded - booksLost$. The safety specification is such that: "the expected value of $pp \times loansEnded - booksLost$ can never be decreased below 0" by the interleaved executions of the machine's operations. Interpreting this in our formalism, the machine must then guarantee that

$$Exp.[\![Library^{k}_{\aleph}]\!].s_0.Expt \ \geq \ 0 \tag{8}$$

for all execution step k and library scheduler $\aleph$. However, in [32] we saw that (8) was violated at the third execution step, i.e. where $k = 3$, as depicted in Fig. 8 hence suggesting that *Expt* meets the second possibility discussed in Section 2 — it is not inductive. But to find out why the inductive definition fails, we explore the capabilities of YAGA in the next section to capture the machine's exact execution sequence corresponding to an extremal scheduler which demonstrates the failure.

5.3. Experimental results

In our precursor paper [32] we explored the instantaneous variant of the PRISM reward structure to establish the faulty behaviour of the library machine with respect to its safety specification. But here, we can explain the exact reasons to back up that failure using the T-labelled transition reward encoding of the PRISM reward structures.



This graph was generated in the PRISM pre-processing stage and displays the expected value of the expression $pp \times loansEnded - booksLost$ treated as a random variable under operations of the pB machine. It shows for example that the quantitative safety property is first violated on the third step, and that the longer the machine can execute the more negative becomes that expected value.

Fig. 8. Experiment over the EXPECTATIONS clause of Fig. 7.

```
************ Starting Error Reporting  ... ***************
     Faulty path located on 3 step(s)
Sequence of operations and state valuations ::>>>
[{INIT} (0,0), {StartLoan} (0,0), {EndLoan} (0,1), {StockTake} (0,0)]
    Path Probability mass Information::>> 0.5
************ Finished Error Reporting  ... ***************
```

Note that every action of the trace information is marked with a pair which denotes the state valuations of the program variables occurring in the EXPECTATIONS clause, in this case (*booksLost*, *loansEnded*).

Fig. 9. A YAGA diagnostic trace information report.

Again, Fig. 8 summarises our initial experimental investigation of the PRISM model of the faulty library machine described in the previous section, after setting the model parameters as $pp = 0.5$, *totalBooks* = 1, *cost* = 1 and MAXCOUNT = 3. As we shall see later, the counterexample identifies exactly which trace corresponds to the failure of the machine to observe its inductive safety definition. The complete MDP PRISM model representation of the faulty library machine is in Fig. 14 of the Appendix.

5.3.1. The most useful diagnostic trace located

After model checking the transformed safety property using the T-labelled transition step, YAGA constructs a failure tree for our analysis. Moreover, given the technique of Section 4, YAGA reports a faulty execution trace responsible for violating the machine's safety property. Fig. 9 shows a sample report output summarising the effect of YAGA on the resultant failure tree. It gives a snapshot of the sequence of actions, after the machine initialisation `INIT` that constitute a faulty trace. This suggests to the verifier that after executing the sequence of machine actions `INIT`, `StartLoan`, `EndLoan` a witness s is reached where the action `StockTake` is enabled and whose valuation captured by the pair $(\texttt{booksLost}, \texttt{loansEnded}) = (0, 0)$ is such that $Wp.\texttt{StockTake}.(pp \times \texttt{loansEnded} - \texttt{booksLost}).s\ (= 0)$ is strictly less than $(pp \times \texttt{loansEnded} - \texttt{booksLost}).s\ (= pp = 0.5)$. YAGA then returns a complete diagnostic trace capturing this failure.

5.3.2. Diagnostic trace "expectations" information

Finally, YAGA ranks the faulty execution traces with respect to a verifier's metric of interest. Note that since we only have a single diagnostic trace whose probability information is given above, ranking it is does not portray much elegance. Nevertheless, to complete a demonstration of the entire technique, we present a sample ranking report in Fig. 10.

```
******* Starting Ranking Information By "expectations" ... *********

    *********** Ranking in Ascending Order ***************
              Path plus expectations Information::>>
[{INIT} (0,0), {StartLoan} (0,0), {EndLoan} (0,1), {StockTake} (0,0)]
              Path residual expectations Information::>> - 0.5
    ************ Finished Ranking Information  ... *********
```

Fig. 10. A YAGA diagnostic trace "expectations" information report.

6. Related work

McIver et al. [28,29] previously explored bounded model checking techniques to explain the failure of inductive invariants by casting the problem of computing a "refutation-of-safety" certificate for probabilistic systems. But what we have done here is to state more succinctly that the existence of a certificate corresponds to an inductive invariant failure in our own context of safety. More importantly, we have also presented these certificates as useful information to a *pB* developer even though it turns out that we have explored them in the form of probabilistic counterexamples.

Counterexample generation for probabilistic system models is quite a recent and burgeoning research discipline. Very notable contributors in this field are Katoen and coworkers [14,15], Aljazzar et al. [4,5], and D'Argenio and coworkers [6,7]. While our work largely borrows from the advances made by these researchers and many more, it does specifically compare to theirs as follows.

In [15] as in [32], we start off with the assumption that a certain unknown state violates a given PCTL temporal property. In [15], Katoen and coworkers showed how to generate counterexamples for a reachability kind of property by computing the set of *prefix-containment free* paths such that the sum of the probabilities along all paths exceed a given threshold over the property.

But this approach in itself is very general. A major problem that arises then is: how do we present system diagnostics to a debugger in a manner that is not vague and difficult to digest? According to our formalism, it turns out that failure traces and hence witnesses will suffice (see Definition 4). The usefulness of that definition is that we take away the burden of trying to comprehend imprecise path information from the debugger and supply only debugging information that are essential to addressing the main cause(s) of system failure.

However, a generalisation of [15] in our approach is the encoding of expected values of random variables as reward structures. This then allows us to use path probabilities to investigate properties of the form $\mathbb{P}_{\sim p}[\Box^{\leq k}\Phi]$ where the usual box notation $\Box$ will ensure Φ is a safety property. But key to doing this is that we can recast counterexamples for violating a safety property to that for violating an equivalent reachability specification given the transformation rule: $\mathbb{P}_{\sim p}[\Box^{\leq k}\Phi] \equiv \neg\mathbb{P}_{\sim 1-p}[\Diamond^{\leq k}\neg\Phi]$ where $\Diamond$ stands for eventuality. This homomorphic transformation allows us to reuse already existing ideas to locate counterexamples by formulating a shortest path problem for deterministic schedulers of MDP representations of *pB* machines.

In comparison to [4,7], we are able to use the PRISM tool to extract a scheduler of the MDP whose probability of reaching our goal state (a state where Φ fails to hold) is maximal with respect the now transformed safety specification. This is because we have encoded expectations as PRISM rewards. But while Aljazzar et al. [4] try to do this by computing the maximum probability for a formulated scheduler compatibility problem over all possible schedulers of an MDP, D'Argenio et al. [7] state that to do so follows an obvious result already established by de Alfaro [11] — that a maximising (deterministic and memoryless) scheduler for an MDP can be extracted by solving a set of linear minimisation problem with respect to a reachability property.

Finally, the idea of ranking counterexamples for probabilistic models is not new — D'Argenio et al. [7] have previously explored this. But the notion of ranking counterexamples information for *pB* machines cannot be overemphasized especially since returning a large set of failure traces demonstrating the failure of a machine inductive safety property will only amount to too much information for a debugger to digest. Consequently, a debugger can exercise the right to rank counterexamples information by the trace total probability mass or the residual expectation. This helps to further bring to focus the first point of machine repair.

7. Conclusion and future work

In this paper, we have presented a theoretical approach as well as its automation for the generation of counterexamples treated as failure traces for simple inductive safety properties for systems development in *pB*. Our technique relies on the assumption that embedding an inductive invariant within the EXPECTATIONS clause of an 'accurate' *pB* machine suffices to initiate the output of a failure trace which demonstrates its violation (if it does exist). Note that our approach does not distinguish between internal and external nondeterminism arising from the fact that *pB* itself is a specification language. But whenever explicitly captured in a system specification, then our method can provide an equivalent interpretation.

In Section 2, we identified two possible causes for the failure of a proof obligation generator to discharge proofs. This work has focused on the second possibility — we aim to disprove the inductiveness of an expression representing expectations provided that the underlying *pB* machine gives an accurate system representation. An interesting future extension of this work will be to explore the complementary scenario. That is, whether or not we can use a correct inductive safety definition of a *pB* machine for bug detection. This is particularly interesting since it will guarantee a full coverage of issues relating to the completeness of our approach rather than just a subset of it.

The elegance of our technique is in its strength to guarantee a failure trace as an evidence for the failure of inductiveness of a safety property. And in the case where multiple failure traces exist, we can drill-drown to finer levels of expectations or probability details using our proposed ranking methodology. In general, the approach is aimed at presenting digestible diagnostic information to a *pB* developer in a very simplified and friendly manner.

Finally, on a large scale, our technical approach is targeted at developing a performance analysis suite of programs for industrial strength safety-critical probabilistic systems construction extending the RODIN platform [2].

Acknowledgements

The author is grateful to his Ph.D. thesis advisor, A.K. McIver for her very useful comments on the early drafts of this paper. The author is also thankful to Dave Parker of the Oxford University Computing Laboratory for the very useful email exchanges that have helped the development of the paper. Finally, the author expresses his profound gratitude to the anonymous reviewers.

Appendix

Algorithm counterexamplegenerator$(T, Expt, E)$

Reserved:
$K \leftarrow ?$: initialize depth of tree prior to search
k : records number of steps from root of T under a witness is located

```
1 : while (unmarked state s at index j exists) do
2 :       prob ← 1;
3 :       exp ← Expt.s_i;
4 :       tr.empty();
5 :       tr.enqueue({INIT}, exp);
6 :       computepath(i, j, T[i, j], T);
7 :       trprobMap.put(tr.dequeue(), prob);
8 :       trexpMap.put(tr.dequeue(), exp);
9 :       if (unmarked successor of s at index j exists) then
10 :          continue;
11 :      else
12 :         mark j;
13 :      next j;
14 : end while;
15 : if (!trprobMap.isEmpty() || !trexpMap.isEmpty()) then
16 :    rankgenerator(0 | 1);
17 : else
18 :    repeat search with a higher value of K;
```

The input parameters to the algorithm are a transition probability matrix representing an instance of a failure tree T, its expectation *Expt* as well its threshold E. A condition to enter the recursive procedure is that a transition is enabled from the initial state s_0(*i.e.* $i = 0$) and next we scan the tree for the next possible successor of s_0 (we say that j gets the column index of the first unmarked non-zero probability on row i of T). In addition, for every reachable successor state of s_0, we record the action label at that state (*i.e.* $\alpha.s_j$) and also the valuation of the expectation *Expt* at that state *i.e.* $Expt.s_j$. But we note that the action label $\{INIT\}$ holds for the initial state which also has the original expectation $Expt.s_0$. So as we traverse T, we enqueue these labels in a FIFO implemented list *tr*. The idea is that when we finally reach a witness *i.e.* any state s where $Wp.(\alpha.s \sqcap skip).Expt.s < Expt.s$ (see the called algorithm *computepath*) then *tr* is always guaranteed to contain the sequence of action labels leading to that failure state. Finally, to enable a verifier rank the trace information with respect to probability *prob* and expectations *exp* metric of choice, the map objects *trprobMap* and *trexpMap* respectively meet that purpose (see also the utility algorithm called *rankgenerator*).

Fig. 11. Counterexample algorithm.

I *rankgenerator*(*rankParamIndex*)

```
Reserved:
0: rank by total probability mass
1: rank by total expectations
mapkeys: stores keys of map objects; it is initially empty
mapvalues: stores values of map objects; it is initially empty
1 : if (rankParamIndex = 0) then
2 :       mapkeys ← extract all keys from trprobMap;
3 :       mapvalues ← extract all values from trprobMap;
4 :       for (size = 0 → |trprobMap|) do
5 :             sort trprobMap on mapvalues;
6 :       end  for;
7 :       output traces and their probability information;
8 : else
9 : if (rankParamIndex = 1) then
10 :      mapkeys ← extract all keys from trexpMap;
11 :      mapvalues ← extract all values from trexpMap;
12 :      for (size = 0 → |trexpMap|) do
13 :            sort trexpMap on mapvalues;
14 :      end  for;
15 :      output traces and their expectations information;
16 : end if;
```

II *computepath*(*rrIndex*, *ccIndex*, *pr*, *T*)

```
Reserved:
tmprrIndex : temporary row index of T;
tmpccIndex : temporary column index of T;
tmprrIndex ← ccIndex;
1 : while (k < K && unmarked column index on row tmprrIndex of T exists) do
2 :       prob ← prob × pr;
3 :       exp ← Wp.(α.s_rrIndex ⊓ skip).Expt · s_rrIndex;
4 :       k + +;
5 :       tr.enqueue(α.s_ccIndex);
6 :       if (exp′ ≥ exp) then
7 :             exp ← exp′;
8 :             tmpccIndex ← 1st unmarked column index on tmprrIndex;
9 :             mark first unmarked column index on tmprrIndex;
10 :             computepath(tmprrIndex, tmpccIndex, T[tmprrIndex, tmpccIndex], T);
11 :      else
12 :            return;
13 :      end if;
14 : end while;
```

Fig. 12. Ranking and computepath algorithms. (I) **Ranking algorithm**: The input parameter to the algorithm is a ranking parameter index *rankParamIndex*. (II) **Computepath algorithm**: The input parameters *rrIndex*, *ccIndex*, *pr* and *T* are passed after a call from the main algorithm *counterexamplegenerator*.

Algorithm pAMN2PRISM (pB_model_in_pAMN)

Required: An interface for pB syntax, PRISM syntax, and regular math operators.
Reserved: MAXCOUNT (integer constant), count, action (integer variables), terminate (boolean)

1: get pAMN parameter list if any
2: create a map object with pAMN clauses as the the keys. Insert their respective values
3: construct value objects with (1) and (2)
4: set module type as MDP
5: construct PRISM constants list from the pAMN parameter list and PROPERTIES key
6: construct PRISM formula list as atomic predicates from the INVARIANT and PROPERTIES keys
7: get PRISM module name from MACHINE key (declare module name)
8: construct PRISM variables list and their initial values from the VARIABLES and INVARIANT keys and the pAMN parameter list (if any)
9: **for** the OPERATIONS key in the map object **do**
  get the list of operations
  for each operation in the list **do**
    for each guard and update statement in operation **do**
      check variables dependency on (6)
10: construct PRISM update statements with the pAMN operations names as the action labels
11: declare **endmodule**
12: construct expectations label from the EXPECTATIONS key in map object
13: declare **module** counter
14: declare count variable, initialised to zero and bounded by MAXCOUNT
15: declare terminate variable, initialised to **false**
16: declare action variable, initialised to zero and bounded by total number of machine operations plus 1 and initialized to zero
17: **for** the OPERATIONS key in the map object **do**
  get the list of operations
  for each operation in the list **do**
    construct a synchronised update statement (increment) on the count variable with (10) and mark operation with an action label
18: construct an unsynchronised update statement on the count variable and mark update with an action label
19: construct a T-labelled update statement on the term variable such that if count = MAXCOUNT set terminate variable to **true**
20: declare **endmodule**
21: declare PRISM **rewards**
22: for states where (count = MAXCOUNT)
23: set T-labelled transition reward to random variable value on EXPECTATIONS key plus MAXCOUNT
24: declare **endrewards**

Fig. 13. An algorithmic description of YAGA's *pB* machine translation

```
const                          totalBooks;
const                          cost;
const double                   pp;
const                          MAXCOUNT;

formula formula0   =           (loansEnded ≤ loansStarted);
formula formula1   =           (booksInLibrary + booksLost + loansStarted - loansEnded = totalBooks);
formula formula2   =           (pp ≤ 1);
formula formula3   =           (0 ≤ pp);

module                         ProbabilisticLibrary
booksLost:[0..totalBooks]      init 0;
totalCost:[0..totalBooks]      init 0;
loansEnded:[0..totalBooks]     init 0;
loansStarted:[0..totalBooks]   init 0;
booksInLibrary:[0..totalBooks] init totalBooks;

[StockTake]                    formula1 & formula0 → (totalCost ′ = cost * booksLost) &
                               (booksInLibrary ′ = booksInLibrary + booksLost) &
                               (loansStarted ′ = loansStarted - loansEnded)&
                               (loansEnded ′ = 0) & (booksLost ′ = 0);

[StartLoan]                    (booksInLibrary > 0) & formula1 & formula0&
                               (loansStarted + 1 ≤ totalBooks) →
                               (booksInLibrary' = booksInLibrary - 1) &
                               (loansStarted' = loansStarted + 1);

[EndLoan]                      (loansEnded < loansStarted) & formula2 & formula1
                               formula3 & formula0 →
                               pp: (booksLost' = booksLost + 1)
                               (loansEnded' = loansEnded + 1) +
                               (1 - pp): (booksInLibrary' = booksInLibrary + 1) &
                               (loansEnded' = loansEnded + 1);
endmodule

label "expectations"   =       (pp * loansEnded - booksLost ≥ 0);

module                         Counter
count:[0..MAXCOUNT]            init 0;
terminate : bool               init false;
action:[0..4]                  init 0;

[StockTake]                    (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 1);
[StartLoan]                    (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 2);
[EndLoan]                      (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 3);
[]                             (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 4);
[T]                            (count = MAXCOUNT) → (terminate ′ = true);
endmodule

rewards
[T]                            (count = MAXCOUNT) : (pp * loansEnded - booksLost) + MAXCOUNT;
endrewards
```

Fig. 14. A YAGA-Generated PRISM Representation of Fig. (7)

References

[1] J.R. Abrial, The B-Book: Assigning Programs to Meaning, Cambridge University Press, 1996.
[2] J.-R. Abrial, M.J. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in event-B, STTT 12 (6) (2010) 447–466.
[3] J.R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, in press.
[4] H. Aljazzar, S. Leue, Generation of counterexamples for model checking of markov decision processes, QEST, IEEE Computer Society, 2009, pp. 197–206.
[5] H. Aljazzar, H. Hermanns, S. Leue, Counterexamples for timed probabilistic reachability, in: P. Pettersson, W. Yi (Eds.), FORMATS, Lecture Notes in Computer Science, vol. 3829, Springer, 2005, pp. 177–195.
[6] M.E. Andrés, P. D'Argenio, Derivation of counterexamples for quantitative model checking, Master's thesis, Universidad Nacional de Córdoba, Argentina, 2006.
[7] M.E Andrés, P.R. D'Argenio, P. van Rossum, Significant diagnostic counterexamples in probabilistic model checking, in: H. Chockler, A.J. Hu (Eds.), Haifa Verification Conference, Lecture Notes in Computer Science, vol. 5394, Springer, 2008, pp. 129–148.
[8] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, Y. Zhu, Bounded model checking, Adv. Comput. 58 (2003) 118–149.
[9] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: D. Kozen (Ed.), Logic of Programs, Lecture Notes in Computer Science, vol. 131, Springer, 1981, pp. 52–71.
[10] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 1999.
[11] L. de Alfaro, Formal verification of probabilistic systems, Ph.D. thesis, Stanford University, USA, 2005.
[12] D. Eppstein, Finding the k shortest paths, SIAM J. Comput. 28 (2) (1998) 652–673.
[13] G.R. Grimmnett, D. Stirzaker, Probability and Random Processes, Oxford University Press, 1982.
[14] T. Han, J.-P. Katoen, Counterexamples in probabilistic model checking, in: O. Grumberg, M. Huth (Eds.), TACAS, Lecture Notes in Computer Science, vol. 4424, Springer, 2007, pp. 72–86.
[15] T. Han, J.-P. Katoen, B. Damman, Counterexample generation in probabilistic model checking, IEEE Trans. Software Eng. 35 (2) (2009) 241–257.
[16] H. Hansson, B. Jonsson, A logic for reasoning about time and reliability, Formal Asp. Comput. 6 (5) (1994) 512–535.
[17] A. Hinton, M.Z. Kwiatkowska, G. Norman, D. Parker, PRISM: A tool for automatic verification of probabilistic systems, in: H. Hermanns, J. Palsberg (Eds.), TACAS, Lecture Notes in Computer Science, vol. 3920, Springer, 2006, pp. 441–444.
[18] T.S. Hoang, Z. Jin, K. Robinson, A.K. McIver, C.C. Morgan, Probabilistic invariants for probabilistic machines, in: D. Bert, J.P. Bowen, S. King, M.A. Waldén (Eds.), ZB, Lecture Notes in Computer Science, vol. 2651, Springer, 2003, pp. 240–259.

[19] T.S. Hoang, Developing a robabilistic B-method and a supporting toolkit, Ph.D. thesis, University of New South Wales, Australia, 2005.
[20] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.
[21] J. Hurd, Formal verification of probabilistic algorithms, Ph.D. thesis, University of Cambridge, United Kingdom, 2002.
[22] V.M. Jiménez, A. Marzal, Computing the k shortest paths: a new algorithm and an experimental comparison, in: J.S. Vitter, C.D. Zaroliagis (Eds.), Algorithm Engineering, Lecture Notes in Computer Science, vol. 1668, Springer, 1999, pp. 15–29.
[23] M.Z. Kwiatkowska, G. Norman, D. Parker, Stochastic model checking, in: M. Bernardo, J. Hillston (Eds.), SFM, Lecture Notes in Computer Science, vol. 4486, Springer, 2007, pp. 220–270.
[24] M.Z. Kwiatkowska, G. Norman, D. Parker, PRISM: probabilistic model checking for performance and reliability analysis, SIGMETRICS Perform. Evaluation Rev. 36 (4) (2009) 40–45.
[25] M. Leuschel, M.J Butler, ProB: A model checker for B, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), FME, Lecture Notes in Computer Science, vol. 2805, Springer, 2003, pp. 855–874.
[26] D.A. Martin, Borel determinacy, Ann. Math. 102 (2) (1975) 363–371.
[27] A.K. McIver, C.C. Morgan, Abstraction, refinement and proof for probabilistic systems, Monographs in Computer Science, Springer-Verlag, 2004..
[28] A.K. McIver, C.C. Morgan, C. Gonzalia, Proofs and refutations for probabilistic refinement, vol. 5014in: J. Cuéllar, T.S.E. Maibaum, K. Sere (Eds.), FM, Lecture Notes in Computer Science, , Springer, 2008, pp. 100–115.
[29] A.K. McIver, C.C. Morgan, C. Gonzalia, Probabilistic affirmation and refutation: case studies, in: Proceedings of Automatic Program Verification, 2009.
[30] C.C. Morgan, The generalised substitution language extended to probabilistic programs, vol. 1393in: D. Bert (Ed.), Lecture Notes in Computer Science, , Springer, 1998, pp. 9–25.
[31] U. Ndukwu, A.K. McIver, YAGA: Automated analysis of quantitative safety specifications in probabilistic B, vol. 6252in: A. Bouajjani, W.-N. Chin (Eds.), ATVA, Lecture Notes in Computer Science, , Springer, 2010, pp. 378–386.
[32] U. Ndukwu, Quantitative safety: Linking proof-based verification with model checking for probabilistic systems, in: S. Andova, A. McIver, P.R. D'Argenio, P.J.L. Cuijpers, J. Markovski, C. Morgan, M. Núñez (Eds.), QFM, EPTCS, vol. 13, 2009, pp. 27–39.
[33] PRISM: Probabilistic symbolic model checker. Available from: <http://www.prismmodelchecker.org/>
[34] M.L. Puterman, Markov Decision Processes, Wiley, 1994.
[35] J.-P. Queille, J. Sifakis, Specification and verification of concurrent systems in CESAR, vol. 137in: M. Dezani-Ciancaglini, U. Montanari (Eds.), Symposium on Programming, Lecture Notes in Computer Science, , Springer, 1982, pp. 337–351.