Model Exploration and Analysis of Quantitative Safety Refinement in Probabilistic Systems

Ukachukwu Ndukwu¹, Thai Son Hoang² and Annabelle McIver¹

¹Department of Computing, Macquarie University Australia
²Department of Computer Science, ETH Zurich Switzerland

Abstract. Probabilistic programs permit the specification of abstract quantitative properties via the encoding of expectations — random variables defined over program state — which prescribe critical model information. Refinement steps which form the basis for elaborating the specification with implementation details must then be checked to ensure that the expectations threshold are never violated. But capturing, interpreting and coping with the failure of expectations (in case they occur) can prove challenging. As for standard systems, counterexamples are an important feature of program construction which can be used to investigate qualitative properties of proof-based models as in the method of Event-B [LB03]. In this paper, we extend our previous work [NM11] on the use of counterexamples to investigate quantitative inductive invariant properties of probabilistic systems refinement in the probabilistic B language [Hoa05]. In particular, we demonstrate how Hoang’s fundamental probabilistic theorem can be equipped with a bounded-style model checking interpretation so that each refinement step of a probabilistic system can be explored algorithmically to compute useful diagnostics where necessary. The diagnostics we obtain are precise and can be used to make accurate quantitative judgements about the evolving refinement relations for pre-defined expectations. We illustrate the technique with pB implementations of Karger’s mincut algorithm and a software engineering problem involving the refinements of an abstract embedded control system design whose components have known probabilities of failure.

Keywords: Probabilistic B; quantitative safety; refinement; model checking; counterexamples; diagnostics

1. Introduction

Quantitative assessment of probabilistic systems is not an easy task. The reason being, apart from the system designers having to cope with the growing complexity of the systems themselves, they must also identify an appropriate formalism (language and semantics) for reasoning about the systems behaviours. There is
also the need for the designers to decide on a suitable temporal logic rich enough as to express the systems associated quantitative properties. Analysis can be difficult when different sources of uncertainty (probability and nondeterminism) are simultaneously present.

Traditional development methods suggest the use of safety properties in program specifications to capture the notion of “nothing bad should happen”. The idea is to 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. This makes it possible to check program assertions — predicates which identify conditions under which the program behaves correctly — for strictly qualitative properties. An example of the use of assertions in program development is: “every output sequence must be preceded by an input sequence.” In a case where the assertion fails to hold, then the system designer’s expertise is leveraged in identifying the cause of the violation.

As for standard (non-probabilistic) systems, it is possible to establish qualitative assessment via a model-based approach with mathematical proofs that can even apply to systems with an infinite state space. The technique relies on a prover (human-assisted or automatic) to discharge proof obligations capturing properties of the systems. For example, the RODIN platform [Rod04] provides a practical framework which explores this idea in the formal language supported by the Event-B modelling method [Abr09]. Development in Event-B is based on the paradigm of abstraction and refinement — whereby a large, complex or critical project starts with a simple abstract specification, after which implementation details can be incrementally introduced. The relationship between a specification and that of its implementation is given by system invariants which guarantee consistency between the respective models.

However, when applied to the development of safety-critical probabilistic systems, a major challenge posed by the refinement methodology is on how to cope with impending failures arising from weakly formulated system invariants. In standard software engineering practices, failure is accommodated in high-level systems design by the duplication of system components to make for redundancy. But this method is not without its own problems. For example, if the probability of failure of the individual components are not known well in advance, then the analysis results become inaccurate and possibly unsuitable to make any quantitative judgements.

The refinement approach for probabilistic systems allows components to be developed separately and their abstractions to be used in the analysis of “top level” designs. In order to account for their quantitative properties, we shall rely on the fact that probabilistic systems can be used to generalise traditional safety properties and hence their refinements, thus giving a scope for the specification of random variables whose expected values must always remain above some given threshold. To do this, we shall explore the practical framework that mimics the RODIN-style method while providing quantitative assessment of probabilistic systems. Hoang provided the necessary foundational work by incorporating probabilistic choice updates into Abrial’s standard B-style method [Abr05], thus giving rise to the probabilistic B or pB method [Hoa05]. This practical inclusion of probability in proof-based probabilistic systems development (as implemented in the pB toolkit) came with many modelling advantages. First, it permits the reasoning of long-term behaviours of the system models while also allowing the verification of quantitative temporal assertions over them. Second, it supports an “expectation transformer” reasoning style based on the logic of McIver and Morgan [MM04, Mor98], thus extending Dijkstra’s weakest preconditions reasoning [Dij76] to probabilistic programs. More importantly, the development framework supports the investigation of interesting performance properties of probabilistic systems refinement.

Recently, there has been remarkable success derived by taking advantage of the complementary benefits of model checking and proof-based style verification approaches for qualitative systems assessment. One such benefit is offered by the ProB [LB03] model checker for the RODIN platform. ProB provides a model exploration facility which uses model checking techniques to provide a precise assessment of event-B models with respect to stated system invariants, so as to help a developer better understand the models under construction. A key feature of model checking incorporated in ProB is the ability to discover counterexamples [BNR03, dAHM00, GC03] which can provide critical and simplified debugging information for the model developer. In our precursor paper [NM11] we provided automation to check quantitative safety invariant properties of probabilistic systems refinement in the pB language via their translation as PRISM reward structures [KNP02]. Our technique allows probabilistic B (pB) modellers to explore the refinement of quantitative safety properties encoded within their models to obtain important diagnostic feedback in the form of counterexamples. The motivation is to adapt model checking techniques to account for the precise
quantitative properties of probabilistic systems in the $pB$ language in as much as ProB does for standard B models.

In this paper, we shall further expand that idea to accommodate practical uses of counterexamples in probabilistic system refinement in the $pB$ language. Counterexamples are a powerful feature of probabilistic model checking. They can be used to summarise the failure of a (formal) probabilistic model to meet a stated quantitative temporal assertion. Here we shall refer to counterexamples as sets of execution traces each with some non-zero probability of occurring and jointly implying that the specified threshold of safety specification is not maintained. Moreover $pB$'s consistency checking enforces inductive invariance of the quantitative safety property thus enabling the counterexample traces to also demonstrate specific points in the model's execution where the inductive property fails. We shall show how to use them to generalise bounded model checking-style analysis for probabilistic system refinement involving two case studies: one based on Karger's mincut algorithm, and another based on an embedded control system; so that an iteration can be verified by exhaustive search provided that quantitative invariants are inductive for all reachable states of the models.

We shall initiate the development method by first adapting our technique to a much simpler case study of a $pB$ implementation of Karger's min-cut algorithm [Kar93]. We shall show how the “contraction” and “amplification” steps of the $pB$ implementation can be analysed using probabilistic model checking techniques. In particular, we show how our notion of counterexamples coined from McIver and Morgan’s transformer logic can be used to provide precise quantitative analysis of the refinement. Afterward we extend the approach to an embedded controller problem with randomly failing components and show how the analysis of quantitative safety specification can be used to determine “failure modes” and “critical sets” necessary to further improve our understanding of component-level hardware systems design. Importantly, we demonstrate how the diagnostics we obtain can be used to make informed quantitative judgements about the system components in order to help a designer cope with failures as best as possible. Overall, this paper is structured as follows: we summarise the notation we shall use in §2; we introduce probabilistic programs in §3; overview of development in $pB$ is in §4; we discuss a model checking interpretation of $pB$ quantitative safety refinement in §5; a strategy to automate that procedure is in §6; we discuss Karger’s $pB$ mincut refinements in §7; we show how to adapt the reasoning to dependability analysis in §8; we demonstrate the approach with a practical case study in §9 and then we conclude.

2. Summary of Notation

In this section we summarise the notation we will need. Function application is represented by a dot, as in $f.x$ (rather than $f(x)$). We use a finite abstract state space $S$. Given predicate $\text{pred}$ we write $[\text{pred}]$ for the characteristic function mapping states satisfying $\text{pred}$ to 1 and to 0 otherwise, punning 1 and 0 with ‘True’ and ‘False’ respectively. We write $\mathcal{E}S$ for the set of real-valued functions from $S$; that is, the set of expectations; and whenever $e, e' \in \mathcal{E}S$ we write $e \Rightarrow e'$ to mean that $(\forall s \in S. e.s \leq e'.s)$ implying that $e$ is everywhere no more than $e'$. We let $\mathbb{D}S$ be the set of all discrete probability distributions over $S$ namely functions from $S$ to the interval $[0, 1]$ which sum to no more than 1; and write $\text{Exp}. \delta.e = \sum_{s \in S} (\delta.s) \times e.s$ for the expected value of $e$ over $S$ where $\delta \in \mathbb{D}S$ and $e \in \mathcal{E}S$. In the program semantics we use least and greatest fixed points to characterise weak and strong iterations respectively, so that if $f$ is a monotone function such that $f : \mathcal{E}S \rightarrow \mathcal{E}S$; that is, a function from expectation to expectation, then $\mu f$ is the least fixed point of $f$ and $\nu f$ is the greatest fixed point of $f$ in the order on expectations. Here monotone means that if $e$ and $e'$ are expectations such that $e \Rightarrow e'$, then $f.e \Rightarrow f.e'$. We also remark that $\mathcal{E}S$ is restricted to the interval $[0, 1]$.

3. Reasoning about Probabilistic Programs

Dijkstra’s “predicate transformers” logic [Dij76] provided a general way of reasoning about standard program behaviours and hence their proof of correctness. The resultant weakest preconditions or $Wp$-style semantics made it possible to visualise programs as functions from state predicates to state predicates. In the logic, a
program \( Prg \) will transform a predicate \( \text{post} \) representing its \textit{postcondition} into the greatest predicate \( \text{pre} \) representing its \textit{precondition}; that is, \( \text{pre} \) characterises the states under which \( Prg \) is guaranteed to establish \( \text{post} \). Intuitively, what this means is that the execution of \( Prg \) in a state satisfying \( \text{pre} \) establishes \( \text{post} \) on termination. To express this logically we write:

\[
\text{pre} \Rightarrow \text{wp} \cdot \text{Prg} \cdot \text{post} .
\] (1)

McIver and Morgan showed how to generalise this kind of reasoning to probabilistic systems. Their quantitative program logic, the probabilistic Guarded Command Language or \( pGCL \) [MM04, Mor98] is a probabilistic extension of Dijkstra’s Guarded Command Language (GCL) [Dij75] to include a probabilistic choice (\( \oplus \)), nondeterministic choice (\( \sqcap \)), strong iteration (\( \text{do } G \rightarrow \text{Prg} \text{ od} \)), weak iteration (\( \text{it } \text{Prg} \text{ ti} \)), and conditional choice (\( \triangleright \)). It supports logic for the specification of both \textit{probabilistic} and \textit{nondeterministic} behaviours. The underlying program logic deals with commands of the form:

\[
Prg \oplus Prg',
\] (2)

where \( Prg \) and \( Prg' \) are program statements and \( p \) is a zero-one bounded expression. The abstract statement in (2) means that the left program branch \( Prg \) will be chosen with a probability of \( p \); whereas the right program branch \( Prg' \) will be chosen with a probability of \( 1 - p \) (also written \( \overline{p} \)).

In general, 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 program \( Prg \) operating over \( S \) we write \([Prg] : S \rightarrow (S \rightarrow [0, 1])\) for the semantic function taking initial states to distributions over final states. For example, let us consider the program \( \text{FairToss} \) below which assigns the result of the toss of a ‘fair’ coin. Here we assign to the variable \( \text{coin} \) the value of \( H \) or \( T \) depending on whether a ‘head’ or ‘tail’ respectively turns up after the toss. That is:

\[
\text{FairToss} \triangleq \text{coin} := H, \text{coin} := T
\] (3)

sets the state variable \( \text{coin} \) to a \( H \) and \( T \) respectively both with probability 0.5. The semantics \([\text{fairToss}]\) for each initial state \( \text{coin} \) is a probability distribution returning 0.5 or \( (1 - 0.5) \) for final states \( \text{coin}' = H \) or \( \text{coin}' = T \) respectively.

Rather than working with this semantics directly, we shall focus on McIver and Morgan’s generalisation of Dijkstra’s weakest precondition or \( \text{wp} \) semantics defined on the program syntax of the pGCL [MM04]. The semantics of the language is set out in Table 1. This generalisation takes account the probabilistic judgements that can be made about probabilistic programs; in particular it can express when predicates can

<table>
<thead>
<tr>
<th>Name</th>
<th>Command</th>
<th>( \text{wp}) Command Expt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity</td>
<td>\text{skip}</td>
<td>( \text{Expt} )</td>
</tr>
<tr>
<td>Assignment</td>
<td>( x := f )</td>
<td>( \text{Expt}(x \mapsto f) )</td>
</tr>
<tr>
<td>Sequential composition</td>
<td>( \text{Prg;Prg'} )</td>
<td>( \text{wp} \cdot \text{Prg} \cdot \text{wp} \cdot \text{Prg'} \cdot \text{Expt} )</td>
</tr>
<tr>
<td>Conditional choice</td>
<td>( \text{Prg} \triangleright G \triangleright \text{Prg'} )</td>
<td>( \text{wp} \cdot \text{Prg} \cdot \text{Expt} \triangleleft G \triangleright \text{wp} \cdot \text{Prg'} \cdot \text{Expt} )</td>
</tr>
<tr>
<td>Probabilistic choice</td>
<td>( \text{Prg} \oplus \text{Prg'} )</td>
<td>( \text{wp} \cdot \text{Prg} \cdot \text{Expt} + (1 - p) \times \text{wp} \cdot \text{Prg'} \cdot \text{Expt} )</td>
</tr>
<tr>
<td>Nondeterministic choice</td>
<td>( \text{Prg} \sqcap \text{Prg'} )</td>
<td>( \text{wp} \cdot \text{Prg} \cdot \text{Expt} \text{ min } \text{wp} \cdot \text{Prg'} \cdot \text{Expt} )</td>
</tr>
<tr>
<td>Strong iteration</td>
<td>( \text{do } G \rightarrow \text{Prg od} )</td>
<td>( \mu X \bullet (\text{wp} \cdot X \triangleleft G \triangleright \text{Expt}) )</td>
</tr>
<tr>
<td>Weak iteration</td>
<td>( \text{it } \text{Prg ti} )</td>
<td>( \nu X \bullet (\text{wp} \cdot X \text{ min } \text{Expt}) )</td>
</tr>
</tbody>
</table>

(See section 2 for the summary of notations) The commands are with respect to expectation \( \text{Expt} \). It is of type \( E S \rightarrow E S \). Programs are composed of identity statements (\( \text{skip} \)), assignments (\( = \)), sequential composition (\( \cdot \)), Boolean statements (\( \langle \text{if } G \text{ then } \text{... else } \text{... fi} \rangle \)), conditional choice (\( \triangleright \)), strong iteration (\( \text{do } G \rightarrow \text{... od} \)), and weak iteration (\( \text{it } \text{... ti} \)). In addition, note also that commands are ordered via refinement so that more refined programs improve probabilistic results. Here \( x \) is a program variable and \( f \) is an expression evaluating to the type of \( x \) where \( (x \mapsto f) \) means that all instances of \( x \) are replaced by \( f \).
be established only \textit{with some probability}. However, as we shall see, it is even more general than that, and capable of expressing critical expected properties of random variables over the program state. Here we use Real-valued expressions over the program state interpreted as expectations. A program command such as

$$\text{pre} \Rightarrow wp \cdot \text{Prog} \cdot \text{post}$$

(4)

is said to be valid exactly when the expected value over the post-program command (RHS of (4)) is at least the value given by the pre-program command (LHS of (4)). In summary, we say that (4) is a valid program command exactly when

$$\text{Exp}\{\text{Prog}\} \cdot \text{post}.s \geq \text{pre}.s$$

(5)

for all states $s \in S$, where \text{post} is interpreted as a random variable over final states and \text{pre} as a real-valued function (as opposed to (1) where \text{pre} and \text{post} are both predicates).

A key benefit of the pGCL lies in its ability to specify nondeterministic behaviours in abstract program commands. To illustrate how this works, suppose we introduce some ‘bias’ so that the coin in the \textit{fairToss} command is no longer fair and now wish to model a coin that is within 5% of being fair. This corresponds to specifying an unknown choice in the coin toss result and that can be described by the \textit{UnfairToss} program command:

$$\text{UnfairToss} \triangleq (\text{coin} := \textit{H}.45 \oplus \text{coin} := \textit{T}) \sqcap (\text{coin} := \textit{H}.55 \oplus \text{coin} := \textit{T}).$$

(6)

An expression such as (6) captures the mix of both probability (\(\oplus\)) and nondeterminism (\(\sqcap\)) in a specification statement and thus yields an interpretation which is similar to a Markov Decision Process [Put94] (as we shall see later). But then \textit{refinement} allows nondeterminism to be decreased so that the program at (3) is a refinement of (6) because the nondeterministic choice has effectively been implemented by a probabilistic choice. This kind of reasoning is central to helping a system designer make the necessary quantitative judgments in the choice of an implementation where refinement is the key to making a transition between program designs with the intention of improving probabilistic results. Next we shall discuss how the transformer semantics can be used reason about program commands in this way.

### 3.1. Expectation transformers

The \textit{pGCL} program logic provides a lifting of Boolean values to the Real domain. Henceforth, we shall now reason in terms of the expectations arising from the embedding of \text{pre} and \text{post} program predicates. The resulting expectation transformer semantics (summarised in Table 1) caters for a larger category of quantitative properties otherwise impossible to capture using pure Boolean reasoning. It is based on the ‘weakest pre-expectation’, which guarantees a lower bound for a ‘post-expectation’ after executing a program command from some initial state thus ensuring its validity.

**Example 1.** To illustrate the expectation transformer semantics, suppose that we wish to know the probability of establishing a head in a coin toss experiment. That is, we wish to calculate the expected value of the characteristic function $[\text{coin} = \textit{H}]$ relative to the program in (6) which will yield the probability that the variable coin will be set to \textit{H} finally. This corresponds to evaluating:

$$wp \cdot \text{UnfairToss} \cdot [\text{coin} = \textit{H}]$$

$$= wp \cdot (\text{coin} := \textit{H}.45 \oplus \text{coin} := \textit{T}) \cdot [\text{coin} = \textit{H}]$$

\hspace{1cm} “nondeterministic choice”

$$\text{min}$$

$$wp \cdot (\text{coin} := \textit{H}.55 \oplus \text{coin} := \textit{T}) \cdot [\text{coin} = \textit{H}]$$

$$= (0.45 \times [H = \textit{H}] + 0.55 \times [T = \textit{H}]) \text{ min } (0.55 \times [H = \textit{H}] + 0.45 \times [T = \textit{H}])$$

\hspace{1cm} “probabilistic choice; assignment”

$$= (0.45 \times 1 + 0.55 \times 0) \text{ min } (0.55 \times 1 + 0.45 \times 0)$$

$$= 0.45 \text{ min } 0.55 = 0.45.$$
Again, this confirms our intuition that a 5% fair coin will only turn up heads with a probability as low as 0.45 because the nondeterministic choice might resolve to the left program branch in (6).

3.2. Probabilistic program refinement

We write $\text{Prog} \sqsubseteq \text{Prog}'$ to express a refinement relation between probabilistic program commands $\text{Prog}$ and $\text{Prog}'$. This relationship implies that for any given post expectation $\text{Expt}$, it must be true that:

$$wp \cdot \text{Prog} \cdot \text{Expt} \Rightarrow wp \cdot \text{Prog}' \cdot \text{Expt},$$

where the implication-like relation $\Rightarrow$ is used to express the fact that a refinement $\text{Prog}'$ of $\text{Prog}$ is as likely to establish $\text{Expt}$ as $\text{Prog}$. In simple mathematical terms, we say that $wp \cdot \text{Prog}' \cdot \text{Expt}$ is everywhere greater than or equal to $wp \cdot \text{Prog} \cdot \text{Expt}$.

**Example 2.** To see how this works, we can use the transformer semantics to establish a refinement relation between the programs given by $\text{FairToss}$ and $\text{UnfairToss}$. To see how this works, we reason:

$$wp \cdot \text{FairToss} \cdot \text{Expt} = \text{"probabilistic choice"}$$

$$= wp \cdot \text{\{(coin := H) \cdot Expt/2 + (coin := T) \cdot Expt/2\} \text{\"we use abbreviations Expt\H and Expt\T to represent Expt(coin \rightarrow H) and Expt(coin \rightarrow T) respectively"}}$$

$$= \text{\{(0.5 \times (0.45 \times \text{Expt\H} + 0.55 \times \text{Expt\T}) + (0.5 \times (0.55 \times \text{Expt\H} + 0.45 \times \text{Expt\T})\} \text{\"arithmetic"}}$$

$$\equiv \text{\{(0.5 \times \text{Expt\H} + 0.55 \times \text{Expt\T}) \text{min (0.55 \times \text{Expt\H} + 0.45 \times \text{Expt\T})} \text{\"arithmetic"}}$$

$$wp \cdot \text{UnfairToss} \cdot \text{Expt}.$$  

The refinement relation is thus established between the programs in equations (3) and (6). The use of probability 0.5 in the third step can be understood by noting that nondeterminism can be refined by probabilistic choice. As mentioned above, this is a key result in the expectation transformer logic [MM04]. Thus refinement provides ordering of program commands with respect to their resultant pre-expectations; and to establish that order for the above example, we write:

$$\text{UnfairToss} \sqsubseteq \text{FairToss}.$$  

The above two examples illustrate how the expectations transformer semantics can be used to reason about quantitative program behaviours. But to exploit this kind of reasoning in a more practical sense requires a modelling framework that can cope with both probabilistic and nondeterministic behaviours while providing facilities to account for quantitative system properties.

In this paper we shall concentrate on certifying probabilistic safety expressions for the refinement of probabilistic program commands. 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 usually characterised by *inductive invariants* — a random variable expression whose expected value is never decreased. Inductive invariants will be a significant component of the refinement of quantitative safety specifications in our $pB$ machines, to which we now turn.

4. Overview of Systems Development in probabilistic B

Abrial’s B-method [Abr05] gave rise to a specification language for describing large-scale abstract system behaviours. More importantly, the development method supports abstraction and refinement — beginning with an abstract specification of a system to which implementation details are gradually introduced via the process of refinement. The relationship between a specification and its implementation is given by some gluing invariants which guarantee consistency between the respective models. Each stage of the refinement
4.1. The abstract machine notation (AMN)

In B, abstract systems are defined by a collection of *machines* which consist of operations describing possible program executions, together with variable declarations and invariants characterising correct behaviour. State information and machine behaviours are encapsulated by the use of *abstract machine clauses*.

The **MACHINE** clause introduces the name of the specification or machine; noting that a specification can sometimes have parameters. The **CONSTANTS** clause is used to specify the abstract machine’s constants; whereas the **VARIABLES** clause introduces the state of the machine, usually a collection of variables. The **PROPERTIES** and **SEES** clauses state assumed properties and other contextual information of the constants and variables. The **INVARIANT** clause sets out constraints on the variables which must be preserved by every operations of the machine (and established by its initialisation). The **INITIALISATION** clause is the initial setup for the machine (i.e. it sets out the initial values for all variables); and the **OPERATIONS** clause contains a list of defined operations which are used to change the state of the machine. In general, operations can execute only if their preconditions hold.

In order to facilitate step-wise refinement, other forms of abstract machines, i.e. **REFINEMENT** and **IMPLEMENTATION**, are used at different stages of the development process, which is summarised in Fig. 1. A development starts with an abstract machine, subsequently step-wise refined via various refinements, finally, refined into a concrete implementation. Whereas refinements are similar to machine, implementation is a special construct that does not have any variables. Instead, the state of the implementation consists of the state of various *imported* machines. Non-deterministic and parallel substitutions cannot be used in the implementation, but sequential substitutions are allowed. In order to update the state of an imported machine, the implementation invokes some operation from that machine accordingly.

A special construct of the implementation is the **WHILE**-loop, which takes the following form
loop $\equiv$ WHILE pred DO
    body
    INVARIANT I
    VARIANT V
END

with the usual meaning that body is executed while the condition pred holds. Termination of the loop is guaranteed by the reasoning about the natural variant V (i.e. strictly decreased for every iteration of the loop). The loop is equipped with some invariant I, which can be used to estimate wp.loop.post, for some arbitrary post-condition post. Given I satisfies conditions such as (1) is maintained by every iteration of the loop and (2) on termination establishes post, i.e. $\neg\text{pred} \land I \implies \text{post}$, then $I \implies wp.loop.post$.

4.2. The PCHOICE operator

An important part of $pAMN$ is the PCHOICE operator which provides a specification construct corresponding to the probabilistic choice as in (2). The operator permits the specification of probabilistic behaviours in a typical $pB$ machine, thus enabling quantitative reasoning about random variables constructed from the program space. The form of a typical PCHOICE statement is given by the following.

PCHOICE $p$ OF S OR T. (8)

Here $p$ is a zero-one bounded expression over an abstract machine’s state space, but usually it will just be a constant. $S$ and $T$ are program statements as in (2). In modelling terms, (8) expresses that the left program branch $S$ will be chosen with a probability of $p$ while the right program branch $T$ will be chosen with a probability of $1 - p$. This important encoding enables quantitative reasoning about $pB$ machines.

4.3. The EXPECTATIONS clause

Probabilistic invariant properties defined as random variables over an abstract machine’s state space can be embedded within the EXPECTATIONS clause. An invariant property defined in this way can be thought of as an “expected-value invariant”. The clause enables a developer to keep track of average properties which must be maintained throughout the lifetime of a machine construction. Its general form is given by

$E \Rightarrow Expt,$ (9)

where both $E$ and $Expt$ are expectations which we recall are interpreted as real-valued functions of the program state. The clause 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. Therefore, this describes a threshold below which the expected value of $Expt$ is not allowed to fall.

4.4. Refinement of $pB$ quantitative safety

The probabilistic-choice PCHOICE can be considered as concrete, that is, it almost corresponds to “code”. In order to accommodate refinement, $pB$ contains a notion of probabilistic specification statements — which can be used to abstractly capture behaviour of probabilistic systems, and a fundamental theorem for proving refinement of such statements.

A probabilistic specification statement is of the following form

$A \mid v : B,$ (10)

where $A$ is an expectation defined over the program variables, $B$ is an expectation refer to $v$. The variables $v$ (which we also call the frame of the substitution) are a sub-vector of program variables $x$, which the substitution “constrains” to change only those variables\(^1\). Thus the probabilistic specification substitution

---

\(^1\) Additionally, $B$ can refer to $x_0$ as the before value of the variables $x$. For simplicity, we omit this detail here.
generalises the traditional standard specification substitution into the probabilistic program domain. As an example, the following statement

\[ \text{FlipHead} \triangleq 0.45 \mid c : [c = H], \]  

(11)

achieves \( c = H \) (post-expectation \( [c = H] \)) with probability at least 0.45 (pre-expectation). Statement (11) captures a part of \( \text{UnfairToss} \), that is, the lower bound on the probability that the coin turning up heads. We can similarly specify the probability of turning up tails for \( \text{UnfairToss} \) as follows.

\[ \text{FlipTail} \triangleq 0.45 \mid c : [c = T]. \]  

(12)

In \( pAMN \), the probabilistic statement is encoded using unbounded choice substitution as follows:

\[
\text{PRE } A \text{ THEN ANY } v' \text{ WHERE } B(v \mapsto v') \text{ THEN } v := v' \text{ END END}
\]

**Theorem 1 (The Fundamental Theorem).** Let \( A \mid v : B \) be a probabilistic specification statement and \( T \) be any substitution. Assume \( Q \) satisfies the assumption: \( \exists v \cdot B \neq 0 \). Then

\[ A \mid v : B \subseteq T \text{ iff } A \Rightarrow \wp.T.B \]

**Example 3.** Applying the fundamental theorem, we can prove that \( \text{UnfairToss} \) refines \( \text{FlipHead} \), i.e.

\begin{verbatim}
MACHINE FlipHead
SEES Real_TYPE

OPERATIONS
    c ← Flip    \triangleq
PRE 0.45 THEN
    ANY c' WHERE
    [c' = H] THEN
    c := c'
END
END

IMPLEMENTATION UnfairToss
REFINES FlipHead
SEES Real_TYPE

OPERATIONS
    c ← Flip    \triangleq
    CHOICE
    PCHOICE 0.45 OF
        c := H
    OR
        c := T
    END
    OR
    PCHOICE 0.55 OF
        c := H
    OR
        c := T
    END
END

\end{verbatim}

The reasoning is similar as the derivation in Sec. 1. Similarly, we can prove that

\[ \text{FlipTail} \not\subseteq \text{UnfairToss} \]
\[ \text{FlipHead} \not\subseteq \text{FairToss} \]
\[ \text{FlipTail} \not\subseteq \text{FairToss} \]

The **WHILE**-loop construct is extended with an **EXPECTATIONS** clause which will act as expected-value invariant of the loop.

\[ \text{loop} \triangleq \text{WHILE } \text{pred} \text{ DO body INARIANT } I \text{ EXPECTATIONS } E \text{ VARIANT } V \text{ END} \]
The loop expectation can be used to estimate $W_{p\text{.loop}.B}$, for some arbitrary post-expectation $B$. Given $E$ satisfies conditions such as (1) is maintained by every iteration of the loop, i.e. $E \Rightarrow W_{p\text{.body}.E}$ and (2) on termination establishes $B$, i.e. $\neg \text{pred} \times E \Rightarrow B$, then $E \Rightarrow W_{p\text{.loop}.B}$.

In the subsequent sections, we look at how $pB$ machines can be corresponding transformed to some Markov Decision Processes and model checked to find counter-examples (if any). Furthermore, the technique can be extended for checking consistency of probabilistic loops and ultimately refinement relationships in $pB$.

5. Probabilistic Safety in Markov Decision Processes

In abstract terms pGCL programs and pB machines may be modelled as a Markov Decision Process (MDP). Recall that an MDP combines the notion of probabilistic updates together with some arbitrary choice between those updates [Put94]: that combination of probabilistic choices together with nondeterministic choices is present in pGCL and captures both features.

In this section we summarise pB models and their quantitative safety specifications in terms of MDPs, and show how to apply model checking’s search techniques for counterexamples to prove quantitative safety as a first step towards generalising standard bounded model checking verification for a controller model. This will then make it possible to represent embedded controller behaviours as pB machines, using the EXPECTATIONS clause to capture relevant safety properties prior to our analysis. The safety properties of interest will be encoded as inductive invariants. This is crucial to the application of exhaustive state exploration for the intended goal.

Here we consider an MDP expressed as a nondeterministic selection $P \triangleq P_0 \sqcap \ldots \sqcap P_n$ of deterministic pGCL programs, where the nondeterminism corresponds to the arbitrary choice, and each $P_i$ corresponds to the probabilistic update for a choice $i$. When $P$ is iterated for some arbitrarily-many steps, we identify a computation path as a finite sequence of states $\langle s_0, s_1, s_2, \ldots, s_n \rangle$ where each $(s_i, s_{i+1})$ is a probabilistic transition of $P$, i.e. $s_{i+1}$ can occur with non-zero probability by executing $P$ from $s_i$. Note that the choice (between $0 \ldots n$) can depend on the previous computation path since for example guards for the individual operations $P_i$ must hold for their selection to be enabled.

Standard safety properties identify a set of “safe” states — the safety property then holds provided that all states reachable from the initial state under specified state transitions are amongst the selected safe states. A generalisation of this for probabilistic systems specifies thresholds on the probability for which the reachable states are always amongst the safe states. The quantitative safety properties encapsulated by the EXPECTATIONS clause are even more general than that, allowing the possibility to specify thresholds on arbitrary expected properties. And since MDPs contain both nondeterministic and probabilistic choice, taking expected values only makes sense over well-defined probability distributions — we need to resolve the nondeterministic choice in all possible ways to yield a set of probability distributions. The next definition sets out a mechanism for doing just that.

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

Once a particular schedule has been selected, the resulting behaviour generates a probability distribution over computation path. We call such a distribution a probabilistic computation tree; such distributions are well-defined with respect to Borel algebras [Mar75] based on the traces.

Definition 2. Given a program $P$, initial state $s_0$ and execution schedule $\mathcal{S}$, we define the corresponding trace distribution $\langle P_k \rangle$. $s_0$ of type $S^* \rightarrow [0, 1]$ to be

$$\langle P_k \rangle . s_0 (s') \triangleq 1 \text{ if } s' = s_0 \text{ else } 0$$

and

$$\langle P_k \rangle . s_0 (\alpha s') \triangleq \langle P_k \rangle . s_0 (\alpha \mathcal{S}) \times \mathcal{S}(\alpha). s'$$

Computation trees of finite depth generate a distribution over endpoints as follows. If we take $K$ steps from
some initial $s_0$ according to the schedule $\aleph$, then the probability of ending in state $s'$ is given by

$$\mathbb{P}[P^K_{\aleph}.s_0.s' \triangleq \sum_{|\alpha|=K} P^{|\alpha|}.s_0.(\alpha s').$$

General quantitative safety properties are intuitively specified via a numeric threshold $e$ and a random variable $\text{Expt}$ over the state space $S$: the expected value of $\text{Expt}$ with respect to any distribution over endpoints should never fall below the threshold $e$.

**Definition 3.** Given threshold $e$ and an expectation $\text{Expt}$ the general quantitative safety property is satisfied by the program $P$ if for all schedules $\aleph$ and $K \geq 0$, we have that $\text{Exp}[P^K_{\aleph}].\text{Expt}.s_0 \geq e$.

The probabilistic Computation Tree Logic or $pCTL$ [HJ94] safety property, which places a threshold on the probability that the reachable states always satisfy the identified “safe” states is expressible using Def. 3 via characteristic expectation $[safe]$. However many more general properties are also expressible, including the expected time complexity of a reachability type temporal specification to satisfy a goal [KNP02, HKNP06]. Here we are interested in identifying situations where the inequality in Def. 3 does not hold. Evidence for the failure is a finite computation tree whose distribution over endpoints illustrates the failure to meet the threshold.

**Definition 4.** Given a probabilistic safety property, a failure tree is defined by a scheduler $\aleph$ and an integer $K \geq 0$ such that $\text{Exp}[P^K_{\aleph}].\text{Expt}.s_0 < e$.

Elsewhere [Ndu11] we showed that if $\text{Expt}$ is an inductive invariant, then the safety property based on $\text{Expt}$ is implied, provided that $e \leq \text{Expt}.s_0$. In fact, given a failure tree, there must be some finite trace $\alpha$ such that $\mathbb{P}[P^{|\alpha|}.s_0.(\alpha s)] > 0$ and $W_p.(P \sqcap \text{skip}).\text{Expt}.s < \text{Expt}.s$ [Ndu11]. Thus, as for standard model checking, we are able to locate specific traces which lead to the failure of the invariant property. We define a counterexample to inductive invariance as follows.

**Definition 5.** Given a scheduler $\aleph$, an expectation $\text{Expt}$ and a program $P$, a counterexample to inductive invariance safety property is a trace $(\alpha s)$ which can occur with non-zero probability, and such that $W_p.P.\text{Expt}.s < \text{Expt}.s$. A state such as $s$ is a witness to failure.

But note that in practice there will be a number of counterexamples. Our technique is able to identify them all given any depth $K$ of computation. Next we discuss how the strategy can be extended to probabilistic loops reasoning.

### 5.1. Extending the approach to probabilistic loops

Here our investigation shall be based on probabilistic loops of the form

$$\text{loop} \triangleq \text{while } G \text{ do } \text{body } \text{od}$$

where $G$ is a predicate over the program state representing the loop guard; $\text{body}$ is a probabilistic program consisting of a finite nondeterministic choice over probabilistic updates. Our aim in this section is to generalise the technique of bounded model checking to prove the safety assertion of the form

$$e \Rightarrow W_p.\text{loop.inv}$$

where $e$ is as previously defined and $\text{inv}$ is the loop invariant.

In the case that (14) does not hold there must be a failure tree (Def. 4) to witness that fact. This will then make it possible to extract failure traces to explain the violation of $\text{inv}$.

But here we shall be interested in the complementary problem, that is, the case that the property does hold. For standard programs this can be established by exhaustively searching the reachable states; any revisiting of a state terminates the search at that point, so that the method is complete for finite state programs: either a counterexample is discovered or all reachable states are visited, and each one checked for satisfaction of the (qualitative) safety property.

The situation is not quite so straightforward for probabilistic programs, and that is because the technique of exhaustive search does not generalise immediately to quantitative safety properties. However via inductive
invariants it does. Consider the program which repeatedly sets a variable $x$ uniformly in the set $\{0, 1, 2\}$ after the initialisation $x := 1$, and terminates whenever $x$ is set to $2$. In this case we might like to verify the safety property that $x \in \{1, 2\}$ with probability at least $1/2$. Expressed as an assertion, it becomes

$$ wp \cdot (x := 1; \textbf{while} (x = 1) \textbf{do} \; x := 0_1/3 \oplus (x := 1 \_1/2 \oplus x := 2) \; \textbf{od}) \cdot [\textit{post}] \geq 1/2, \quad (15) $$

where $[\textit{post}]$ is the lifting to the Reals of the predicate $\textit{post}$ defined such that $\textit{post} \triangleq x \in \{1, 2\}$. A quantitative inductive invariant establishing that fact is given by $x/2$, expressing the probability that the safety property is always satisfied at that state. (When $x$ is 2, probability is 1, when $x$ is 1, it is 1/2 and when $x$ is 0 it is 0.) In fact the property (15) is equivalently formulated by setting $\textit{post} \triangleq x/2$, which can be seen as a strengthening of $x \in \{1, 2\}$.

Since the assertion (15) does indeed hold, no failure trees exist; more generally, in standard model checking and for finite state spaces such a failure to establish the presence of a failure tree can be converted to a proof that the property holds (provided all reachable states are examined). For probabilistic systems however, it is not clear when to terminate a state exploration, since $\text{Exp}[\textit{body}_{\mathcal{K}}] x/2$ steadily approaches $1/2$ from above (where here $\textit{body}$ is taken to be the guarded loop body of (15)). However we can recover the termination property even for probabilistic systems by looking at inductive invariants, as the next lemma shows.

**Lemma 1.** Let $P$ be a probabilistic program operating over a finite state space $S$; let $s_0$ be the initial state. If for all states $s$, reachable from $s_0$ under executions via $P$, the inductive invariant property $Wp.P.inv.s \geq inv.s$ holds, then $\text{Exp}[P^K],inv \geq inv.s_0$ for all $K$ and schedules $\mathcal{K}$.

**Proof 1.** (Sketch) We use proof by induction on $K$. When $K = 1$ we note that $\text{Exp}[P[1],inv \geq inv.s_0$ is a consequence of the assumption since $\text{Exp}[P[1],inv \geq Wp.P.inv.s_0$.

For the general step, we observe similarly that $\text{Exp}[P[K+1],inv \geq \text{Exp}[P^K][Wp.P.inv]$. The result follows through monotonicity of the expectation operator.

Lem. 1 implies that we can use exhaustive search to verify quantitative safety properties using inductive invariants and exhaustive state exploration. The search terminates once all reachable states have been verified as satisfying the inductive property. In the case of (15), using $x/2$ for the invariant, each of the three states satisfies the inductive property. Next we discuss how this has been implemented in a prototype tool framework which analyses pB machine refinements as a probabilistic model checking problem with a particular focus on the quantitative safety specification.

6. Exploring Probabilistic Systems Refinement $A \sqsubseteq_{\textit{Expt}} B$

YAGA [NM10b] is a prototype suite of programs for inspecting safety specifications of abstract pB machines and their refinements via a K-shortest path procedure [Epp98, JM99]. Importantly, it allows a pB machine designer to explore experimentally the details of system construction in order to ascertain the cause(s) of failure of a pB refinement with respect to a stated quantitative property.

YAGA inputs a pB machine or its refinement violating a specific safety property expressed in its EXPECTATIONS clause, and generates its equivalent MDP representation in the PRISM language [KNP02, HKNP06]. PRISM is a probabilistic model checker (based on the reactive modules formalism [AH99]) that permits pB models as MDPs in the tool framework and thus can investigate critical expected values of random variables as “reward structures” — a part of PRISM’s specification language. PRISM can then be used to explore the computation of $\text{Exp}[P^K],\text{Expt.s_0}$ for values of $K \geq 0$, and thus (modulo computing resources) can determine values of $K$ for which the expectations clause fails. If such a $K$ is discovered, YAGA is able to extract the resultant failure tree as an “extremal scheduler” that fails the inductivity test. The extremal scheduler is a transition probability matrix which gives a description of the best (or worst-case) deterministic scheduler of the PRISM representation of an abstract ‘faulty’ pB machine — that is, one whose probability (or reward) of reaching a state where our intended safety specification is violated is maximal (or minimal) [dA97].

Finally, YAGA analyses the resultant extremal scheduler using algorithmic techniques set out in [Ndu11] and generates “the most useful” diagnostic information composed of finite execution traces as sequences of
The implementation of $Op$ typically contains some local variables $impVar$ and consists of sequential composition of several statements. Beside control structures such as \texttt{IF...THEN...ELSE...END} or \texttt{WHILE}-loops, the implementation can also invoke operation $OP_A$ from included machine $A$.

\textbf{Fig. 2.} A simple pB implementation containing loop.

The procedure sets out how to construct PRISM’s unit of communication, that is modules from a pB refinement relation. Details of how to construct the individual modules are set out in Def. 6 while the algorithmic transformation is set out elsewhere [Ndu11].

\textbf{Fig. 3.} A pB refinement transformation procedure.

operations and their state valuations leading from the initial state of the pB machine to a state where the property is violated. Details of the underlying theory of YAGA, its algorithms and implementation can be found elsewhere [NM10b, Ndu11]. In the next section we discuss practical details on how to use exhaustive search of pB machines to verify compliance of inductivity for finite probabilistic models.

Next we set out a formal definition that enables PRISM to interpret a pB machine refinement in such a way that the semantic equivalence of the machine refinement is preserved as in the action systems formalism [BS91].

\section{Understanding pB refinement as a PRISM model}

In this section we shall provide a model checking interpretation of a pB machine refinement in the PRISM language. Our focus is on those pB refinements incorporating probabilistic loops. 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 a $pB$ model to be captured as an MDP in the tool. The idea here is to interpret the pB operations as well as their loop behaviours as PRISM modules — PRISM’s
basic unit of communication — and use a special ‘counter module’ which sets flags that control entry and exit conditions into the loops. We shall then discuss a set of operational rules akin to process-algebraic commands which are used to determine for example, when a loop has finished its execution. But the key idea is that we are able to check the ‘reward specification’ constructed from the refinements EXPECTATIONS clause to inform our judgement as to when a refinement does not hold with respect to the expectations property of interest.

We write \( A \sqsubseteq_{\text{Expt}} B \) to mean that we want to inspect a \( \text{pB} \) refinement relation between two probabilistic programs \( A \) and \( B \) incorporating probabilistic loops with respect to an expectation \( \text{Expt} \) as in Fig. 2. It is possible to derive an equivalent model checking interpretation for the refinement relation \( A \sqsubseteq_{\text{Expt}} B \) in the PRISM language and then use algorithmic techniques to confirm or refute the statement. We note that as in Fig. 2, the abstract machine \( A \) and its refinement \( B \) are assumed to have a finite number of operations.

The procedure described in Fig. 3 shows how this can be done. The essential idea is to capture distinct operations defined within \( A \) and \( B \) as separate PRISM modules with probabilistic loops defined within them well synchronised by the operational rules set out in Fig. 4. The EXPECTATIONS clause of \( B \) is then interpreted as a PRISM reward. Next we discuss the resultant PRISM model following from the procedure.

**Definition 6.** A PRISM MDP model description of the refinement procedure in Fig. 3 is a tuple given by \( P = \langle \text{var}(P), \text{sys}, \{MMA_1, ..., MMA_{n1}, MMB_1, ..., MMB_{n2}, LMA_1, ..., LMA_{n}, LMB_1, ..., LMB_{n2}, CM\}, \text{Expt}(P) \rangle \) consisting of a finite set of (Boolean or integer) variables \( \text{var}(P) \), a system definition \( \text{sys} \) over valuations \( \text{val}(P) \) of \( \text{var}(P) \); a finite set consisting of main modules \( MMA_1, ..., MMA_{n1} \) of \( A \), main modules \( MMB_1, ..., MMB_{n2} \) of \( B \), loop modules \( LMA_1, ..., LMA_{n} \) of \( A \) and loop modules \( LMB_1, ..., LMB_{n} \) of \( B \), and finally a counter module \( CM \). The system definition \( \text{sys} \) is a process-algebraic expression containing each module \( n \in \{MMA_1, ..., MMA_{n1}, MMB_1, ..., MMB_{n2}, LMA_1, ..., LMA_{n}, LMB_1, ..., LMB_{n2}, CM\} \) exactly once. Finally, \( \text{Expt}(\text{val}(P)) \) is the expectation of a random variable constructed from \( \text{val}(P) \) using \( P \)’s reward structure. Each main module \( mm \in \{MMA_1, ..., MMA_{n1}, MMB_1, ..., MMB_{n2}\} \) defines the operations of \( A \) and \( B \). It consists of:

- A finite set of local variables i.e. \( \text{var}(mm) \subseteq \text{var}(P) \) such that:
  - \( \text{var}(mm) \) are disjoint from other local variables of all other modules
  - each variable \( v \in \text{var}(mm) \) has initial value \( \text{init}(v) \)
  - \( \text{init}(mm) \) denotes the initial values of the variables in \( \text{var}(mm) \)

- A finite set of commands \( \text{com}(mm) \) which describe the statements of the individual operations of \( mm \), where each command \( cmd \in \text{com}(mm) \) includes:
  - a guard \( \text{gd}(cmd) \) which is a Boolean function over \( \text{val}(\text{var}(P)) \)
  - an action label \( \text{act}(cmd) \) which is the name of each unique operation of \( A \) or \( B \) (see Fig. 3)
  - a finite set of update statements \( \text{updates}(cmd) \triangleq \{\langle \lambda_i, u_1, ..., \lambda_n, u_n \rangle \mid \text{for each } \lambda_i \in (0, 1) \} \) is a function from valuations over \( \text{var}(P) \) to the valuations over \( \text{var}(mm) \) and moreover \( \sum_{i=1}^{n} \lambda_i = 1 \).

The syntax of a typical PRISM update statement is given by:

\[
\text{[act]} \quad \text{gd} \rightarrow \lambda_1 u_1 + ... + \lambda_n u_n
\]

Each loop module \( lm \in \{LMA_1, ..., LMA_{n}, LMB_1, ..., LMB_{n2}\} \) describes the necessary conditions to execute its associated main module. It is defined such that:

- \( \text{var}(lm) = \{\text{flag, guard}\} \) where
  - \( \text{flag} \) is of type Integer and \( \text{init}(\text{flag}) = 0 \); flag is then bounded by the maximum number of operation calls before a loop exit condition is satisfied.
  - \( \text{guard} \) is of type Boolean and \( \text{init}(\text{guard}) = \text{true} \).

---

Note that \( \text{impVar} \) is the list of implementation variables used to implement the operations within the loop body itself; the statement \( \text{init}(\text{impVar}) \) only initialises those variables.
• A finite set of commands $\text{com}(lm)$ that control the loop entry and exit conditions. The commands of $\text{com}(lm)$ do this such that:
  - the entry command corresponds to a valuation of $\text{var}(P)$ where a loop guard $G$ holds
  - the exit command corresponds to a valuation of $\text{var}(P)$ where the negation of a loop guard $\neg G$ holds

Each $\text{com}(lm)$ follows a similar rule as the commands of $\text{com}(mm)$ except that $\lambda_i$ for all updates is always 1.

• The counter module $\text{CM}$ is similarly defined in the same manner as the main modules and the loop modules but we restrict its set of local variables such that $\text{var}(\text{CM}) = \{\text{terminate}, \text{count}, \text{action}\}$ where
  - $\text{terminate}$ is of type Boolean and init ($\text{terminate}$) = false;
  - $\text{count}$ is of type Integer and $\text{count}:[0, \text{MAXCOUNT}]$ and init ($\text{count}$) = 0, where $\text{MAXCOUNT}$ is a model Integer constant;
  - $\text{action}$ is of type Integer and $\text{action}:[0, \text{TNA} + 1]$ and init ($\text{action}$) = 0 where $\text{TNA}$ is the total number of actions captured in the refinement.

• In addition, for each update statement of the main modules that commits changes to the variables $\text{impVar}$, there is an update of $\text{CM}$ that synchronises with it. Those updates of $\text{CM}$ can only update the $\text{action}$ variable and increment the $\text{count}$ variable by 1. The same also applies to the update statements of the loop modules that are used to control the entry and exit condition into the outermost loop. Also, $\text{CM}$ must contain two extra update statements: a similar unsynchronised update statement$^3$ that can only update the $\text{action}$ variable and increment the $\text{count}$ variable by 1. This can happen whenever a loop entry or exit condition is yet to be satisfied so that time just advances; and a T-labeled update statement that can only reset the $\text{terminate}$ variable to true whenever the guard ($\text{count} = \text{MAXCOUNT}$) holds. This T-labeled transition ensures that we can investigate the EXPECTATIONS clause in terms of PRISM rewards by successively unwinding the program computation tree until the guard ($\text{count} = \text{MAXCOUNT}$) holds. Details of how to construct PRISM modules from pB programs have been set out elsewhere [Ndu11].

6.2. Operational rules for loop modules communication

For simplicity sake, we shall assume that after executing the procedure of Fig. 3, the refinement relation $A \sqsubseteq B$ which holds according to Theorem 1 produces two loop modules, one each for the probabilistic programs $A$ and $B$. In addition, let $LMA$ and $LMB$ denote the loop modules of the program commands $A$ and $B$ respectively so that $\text{flag}_{LMA}, \text{guard}_{LMA}$ (respectively $\text{flag}_{LMB}, \text{guard}_{LMB}$) denote the flags and guards of $LMA$ and $LMB$. Also, note that $v_A$ and $v_B$ are the valuations of the variables of $LMA$ and $LMB$ respectively. Let $P$ be a PRISM MDP model that captures the refinement relation (see Def. 6) where $\text{INIT}(P)$, $\text{RUN}(P)$ and $\text{STOP}(P)$ are basic process algebraic commands offered by $sys$ over a particular scheduler of $P$ to initialise, execute and stop a process. The operational rules in Fig. 4 provide the mechanism for shared variables communication of the set of variables $v$ of the loop modules of a PRISM MDP model summarising a pB machine refinement. Next we describe how $\text{Exp}(P)$ can be computed from a refinement model of $P$. Next we describe how the EXPECTATIONS clause can be interpreted as a PRISM reward prior to algorithmic analyses.

6.3. 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

$^3$ This update statement is analogous to the $\text{skip}$ programme.
The expected value of the rewards if specified in the temporal logic $\text{PCTL}$ [HJ94]. To further help us explore the usefulness of the T-labeled transition of the counter module we set out the definitions below:

**Definition 7.** A transition reward is assigned to transitions of a DTMC by defining the reward function $\bar{l}: S \times S \rightarrow \mathbb{R}_{\geq 0}$. For example, the transition reward $\bar{l}(s, s')$ is acquired each time a transition is enabled from state $s$ to state $s'$ for all $s, s' \in S$.

**Definition 8.** 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 $\Phi$ 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 $\Phi$ is a PCTL state formula.

For an MDP, the reward specification\(^4\) 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-labeled transition reward in our model using the PRISM rewards ... endrewards keywords (as in Fig. 5). The remainder of the investigation then becomes algorithmic as it reduces to finding a scheduler of the MDP interpretation of the refinement which can be used to establish a strict decrease of the expectation of interest.

Given two probabilistic programs $A$ and $B$ such that $A \sqsubseteq_{\text{Exp}} B$ is a refinement problem for investigation with respect to an expectation $\text{Expt}$. We shall write $P \models A \sqsubseteq_{\text{Exp}} B$ if and only if there is a PRISM MDP program $P$ that models the problem. The next PRISM refinement theorem follows:

**Theorem 2 (PRISM Refinement Theorem).** Given that $P \models A \sqsubseteq_{\text{Exp}} B$ holds if and only if $\text{Exp}[P_s] \geq \text{Expt}$ for all schedulers $\mathcal{S}$ of $P$ where $\text{Exp}[P_s] \geq \text{Expt}$ is given by the reward formula $R_{\min \sim [\text{F terminate}]}$.

**Proof 2.** A practical illustration of the proof of the theorem can be established by encoding $\text{Expt}$ as in Fig. 5 and exploring $P$ via bounded model checking by computing $W_{p,s_0}[\text{MaxCount}.\text{Expt}, s_0]$ for

---

\(^4\) Note that PRISM can only generate adversaries violating properties of this type.
Formal Aspects of Computing: LaTeX \textcopyright Submissions

---

**Fig. 5.** Encoding \textit{Expt} as a PRISM reward allows us model check pB machines using PCTL reward specifications of the form:

\[ R_{min \geq E \cdot s_0}[F \text{ terminate}] \] where \( E \cdot s_0 \) is the safety threshold. Note that this reward is computable only after the T-labeled transition fires thus setting the future predicate \text{terminate} to \text{true} whenever \((\text{count} = \text{MAXCOUNT})\) holds.

---

**Fig. 6.** A pB refinement of the contraction specification of the Mincut algorithm.

---

reachable states of the model 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-labeled transition fires.

**Corollary 1.** If \( P \models A \sqsubseteq_B \text{Expt} \) and \( A \sqsubseteq_B \text{Expt} \) does not hold for a given \text{MAXCOUNT} then there must exist a state \( s \) at that depth and a trace \((\alpha_s)\) which can occur with non-zero probability, and such that \( W_{pB}(P, \sqsubseteq_B \text{Expt}) < \text{Expt.s} \).

Practical details of how to explore this investigation have been provided elsewhere [Ndu11]. Next we shall discuss how to use the method to analyse a two-step pB refinement of Karger’s mincut specification [Kar93].

---

7. Case Study One: Min-cut

We discuss one of Hoang’s pB models [Hoa05]: a randomised solution to finding the “minimum cut” in an undirected graph. The probabilistic algorithm is originally due to Karger [Kar93]. We also report experimental results after running our refinement checking diagnostic tool.

Let an undirected graph be given by \((N, E)\) where \( N \) is a set of nodes and \( E \) is a set of edges. The graph is said to be disconnected if \( N \) is a disjoint union of two nonempty sets \( N_0, N_1 \) such that any edge in \( E \) connects nodes in \( N_0 \) or \( N_1 \); a graph is connected if it is not disconnected. A cut in a connected graph is a subset \( E' \subseteq E \) such that \((N, E \setminus E')\) is disconnected: a cut is minimal if there is no cut with strictly smaller size. Cuts are useful in optimisation problems but are difficult to find. Karger’s algorithm uses a randomisation technique which is not guaranteed to find the minimal cut, but only with some probability. That probability can be increased to be arbitrarily close to 1 using the well-known technique of “probabilistic amplification” [MR97]. The idea of the algorithm is to use a “contraction” step, where first an edge \( e \) connecting two nodes \((n_1, n_2)\) is selected at random and then a new graph created from the old by “merging” \( n_1 \) and \( n_2 \) into a single node \( n_{12} \); edges in the merged graph are the same as in the original graph except for edges that connected either \( n_1 \) or \( n_2 \). In that case if \((n_1, a)\), say was an edge in the original graph then \((n_{12}, a)\) is an edge in the merged graph. We keep merging while the number of nodes is greater than 2. The specification
Table 2. Performance result of inductive invariance checking for the contraction step.

<table>
<thead>
<tr>
<th>NN</th>
<th>States, transitions</th>
<th>Probability to find a mincut</th>
<th>Number of steps</th>
<th>Duration (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>72517, 128078</td>
<td>2.2222 E-2</td>
<td>23</td>
<td>18.046</td>
</tr>
<tr>
<td>50</td>
<td>412797, 732718</td>
<td>8.1633 E-4</td>
<td>143</td>
<td>131.363</td>
</tr>
<tr>
<td>100</td>
<td>797647, 1416518</td>
<td>2.0202 E-4</td>
<td>293</td>
<td>277.605</td>
</tr>
</tbody>
</table>

PRISM model checking results for the contraction step MAXCOUNT = 1000

of the merge function for an initial number of nodes NN is such that

\[ ans \leftarrow \text{merge}(nn, aa) \triangleq nn \in NN \land aa \in BOOL \mid ans := (\text{false} \leq 2/nn \oplus aa). \]

It expresses that with a probability of at most \(2/nn\), the minimum cut will be destroyed by the contraction step. Otherwise the minimum cut is guaranteed to be found. Contraction satisfies an interesting combinatorial property which is that if the edge is chosen uniformly at random from the set of edges then the merged graph has the same minimum cut as does the unmerged graph with probability at least \(2/(NN(NN–1))\). Although this probability can be small, it can be amplified by repeating the algorithm to give a probability of assurance to within any specified threshold. To inspect the probabilities of finding a true mincut, we first generate a PRISM model of the refinement according to Def. 6 and then use PRISM’s reward structure to computing an equivalent reward corresponding to the probabilities of finding a mincut according to Theorem 2. However, if this specification fails to hold, we can then proceed with the details of Corollary 1 by obtaining an extremal scheduler that demonstrates the failure. Such a scheduler will then be used to compute the necessary diagnostics trace.

7.1. Refinement step 1 - contraction

The pB implementation shown in Fig. 6 sets out part of the refinement step for the contraction step of the min-cut algorithm. The refinement describes an iteration where the merge function is called to perform the contraction described above. The result of a call to merge is that the number of nodes in the graph (given by the variable \(nn\)) is diminished by 1 and either the original minimum cut is preserved (with probability mentioned above), or it is not; the Boolean \(ans\) is used to indicate which of these possibilities has been selected.

Here we use the expectation(.) function to check that the expression \([ans \times 2/(nn(nn–1))\) simplifies to an inductive property; that is, that the probability of preserving the minimum cut should always be at least \(2/(nn(nn–1))\) while \(ans\) remains true, but is 0 if \(ans\) ever becomes false. Note that if this property holds then we are able to deduce exactly that the overall probability that the original minimum cut is preserved when the graph is merged to one of 2 nodes is the theoretically predicted \(2/(NN(NN–1)).\)

7.1.1. Experiment 1: introducing error into the EXPECTATIONS clause

Here we summarise a number of experiments using bounded model checking style techniques of our method to analyse the refinement. In our first experiment we introduce an error \(^5\) in the design of the merge function so that the invariant is no longer inductive and thus fails. The graph depicted in Fig. 7 shows a failure to preserve the expected probability threshold of the mincut algorithm. Specifically the graph shows that the probability falls below \(2/(NN(NN–1)).\) An examination of the resultant failure tree produces the counterexample depicted in Fig. 8. It clearly reveals a problem ultimately leading to a witness after executing the merge operation.

\(^5\) We set the probability of choosing the left branch in the merge specification to be “at most” 3/4 so that the new specification becomes \(ans := (\text{false} \leq 3/4 \oplus aa)\)
Fig. 7. Graph comparing the probabilities to find a min-cut for the correct and incorrect implementations of the contraction specification of the min-cut algorithm. The incorrect implementation is where we have introduced a high probability in the left branch of the merge operation thus forcing the variable ans to become false often.

******* Starting Error Reporting for Failure Traces located on step 2 **********
Sequence of operations leading to bad state ::>>>
[[INIT] (3,true), {Skip} (3,true), {Merge} (3,false)] >> 0.75
[[INIT] (3,true), {Skip} (3,true), {Merge} (3,true)] >> 0.25
Total probability mass of failure traces is:>>>> 1
******* Finished Error Reporting***************

Fig. 8. Diagnostics detailing a failure of the inductive invariance at the implementation step (for NN = 3) involving the merge operation. Note that this is a counterexample since the execution of the merge operation will result in an endpoint distribution which yields a decreased expectation (see Def.5). That is, there is a witness s (nn = 3, ans = true) such that \( W_p.\text{merge.2/(nn(nn - 1))}.s = 1/12 < 2/(nn(nn - 1)).s = 1/3 \). Note that every trace component of the counterexample is marked with a pair which denotes the state valuations of the program variables occurring in the frame (see Theorem 1) given by the EXPECTATIONS clause, in this case (nn, ans).

7.1.2. Experiment 2: proof of correctness for small models

In this experiment we fix the error in the merge function and attempt a verification of mincut for specific (small) model sizes. In particular, we use YAGA to check that the EXPECTATIONS clause satisfies the inductive property for all reachable states of the equivalent PRISM model defined by MAXCOUNT. The result is shown in Table 2. It depicts the various sizes of the PRISM model relative to the number of nodes NN of interest of the original graph. We are also able to compute the minimal number of steps required to find a mincut. The results we obtain further prove that this refinement step is correct according to Theorem 2. In the next experiment, we shall attempt another level of refinement and generate the equivalent PRISM model. We shall again, embed the expectations as PRISM rewards and then compute the resultant (increased) probability to find a min-cut.

7.2. Refinement step 2 - amplification

Similar to the contraction machine, the specification MinCut does not have any variables to represent its state. The machine has only one operation, namely minCut. As well as the input NN representing the number of nodes in the original graph; the operation has one extra input MM that represents the number of times we “amplify” the probability, that is, the number of times we will carry out a contraction. The output ans of the operation again abstractly models whether we find a true minimum cut or not after the amplification process. A more detailed pB specification of the amplification step can be found elsewhere [Hoa05]. Importantly, we want to maintain the inductive property in the specification that the probability of finding the correct minimum cut should be at least \( P(NN, mm) = 1 - (1 - p(NN))^mm \), where \( p(NN) = 2/(NN(NN-1)) \).

The implementation of the probabilistic amplification is shown in Fig. 9. In the implementation, the variable mm is assigned MM to ensure that we can repeat the contraction process MM times; and ans is assigned false since we have not yet found the right minimum cut. Within the loop body, we undergo a
**IMPLEMENTATION**

minCutImp

**REFINES**

minCut

**SEES**

Bool_Type, Int_TYPE, Real_TYPE, Bool_TYPE_Ops, contraction

**OPERATIONS**

\[
\begin{align*}
\text{ans} & \leftarrow \text{minCut} \left( NN, MM \right) \\
& \equiv \text{VAR} \ mm, \ aa \ \text{IN} \\
& \ \ mm := MM; \ ans := \text{FALSE}; \\
& \ \text{WHILE} (mm > 0) \ DO \\
& \ \ aa \leftarrow \text{contraction}(NN); \\
& \ \ ans \leftarrow \text{DISBOOL}(aa, ans); \\
& \ \ mm := mm - 1 \\
& \ \text{VARIANT} \ mm \\
& \ \text{INVARIANT} \ mm \in \mathbb{N} \land mm \leq MM \land 2 \leq nn \land ans \in \text{BOOL} \land \\
& \ \text{expectation}(\left[\text{ans}\right] + [mm \neq 0] \times [\neg\text{ans}] \times (1 - (1 - p(\text{NN}))^{mm}))
\end{align*}
\]

**END;**

**END**

---

**Fig. 9.** A pB refinement of the contraction specification of the Mincut algorithm.

**rewards**

\[
\begin{align*}
\left[ T \right] (\text{count} = \text{MAXCOUNT}) & \land (\text{ans} = \text{true}) : 1 \\
\left[ T \right] (\text{count} = \text{MAXCOUNT}) & \land (\text{ans} = \text{false}) \land (mm \neq 0) : 1 - \text{pow}(1 - (2/(NN \times (NN - 1))), mm)
\end{align*}
\]

endrewards

---

**Fig. 10.** Encoding Expt as a PRISM reward allows us model check pB machines using PCTL reward specifications of the form: \(R_{\text{min}} \geq E.s_0[F \text{ terminate}]\) where \(E.s_0\) is the safety threshold. Note that this reward is computable only after the T-labeled transition fires thus setting the future predicate terminate to true whenever \((\text{count} = \text{MAXCOUNT})\) holds.

The contraction process and the result is output to \(aa\); we then compute \(ans\) as the disjunction of the new result \(aa\) and the previous value of \(ans\) (since if we find the correct (least) cut once, we can never lose it); and finally, the counter decreases accordingly.

Key to understanding the refinement as a model checking problem in the PRISM language is the interpretation of the expected value invariant. The equivalent PRISM reward structure that captures that idea is shown in Fig. 10: The complete PRISM model of the refinement following Def. 6 is shown in Fig. 11.

Next we compare the performance results of the amplification step to the contraction in terms of the model checking time, minimal number of steps and probability to find a mincut.

**7.2.1. Experiment 3: proof of correctness of the amplification step**

This experiment shows that the refinement passes the inductivity test with respect to the prescribed invariance. YAGA was unable to compute any diagnostic trace to summarise a failure hence confirming that the refinement is correct and that the fundamental theorem (Theorem 1) holds. Next we tabulate bounded model checking results of the refinement with respect to the PRISM reward structure of Fig. 10 verified using the PCTL specification statement in Theorem 2. The results are shown in Table. 3.

**Table 3.** Performance result of inductive invariance checking for mincut. Observe that improvement in the probability to find a mincut when compared with the results in Table. 2.

<table>
<thead>
<tr>
<th>NN</th>
<th>States, transitions</th>
<th>Probability to find a mincut</th>
<th>Duration (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>63801, 87756</td>
<td>2.0127 E-1</td>
<td>27.895</td>
</tr>
<tr>
<td>50</td>
<td>336021, 476836</td>
<td>8.1333 E-3</td>
<td>141.761</td>
</tr>
<tr>
<td>100</td>
<td>660546, 940686</td>
<td>2.0184 E-3</td>
<td>307.288</td>
</tr>
</tbody>
</table>
Formal Aspects of Computing: L

endmodule → [T] (count = MAXCOUNT)
[contraction] (flag2 = 2) & (loop2 = true)
→ [renewouterloopguard] (flag1 = 0) & (loop2 = false) & f6
→ [mincut] (flag1 = 2)
→ [merge] f4 & f5
→ [merge] true
ansmerge:
module
endmodule
≤ [outerloopguard] (flag1 = 0) & (loop2 = true) & (count + 1 ≤ MAXCOUNT)
[mege] (flag2 = 1) & (loop2 = true)
loop2:

module
ContractionLoop
endmodule

[decide] (flag1 = 1) & (loop1 = true) & (loop2 = false) → (aa' = anscnt)
[decide] (flag1 = 1) & (loop1 = true) & (loop2 = true) → (flag'1 = 1);
[decide] (flag1 = 1) & (loop1 = true) & (loop2 = false) → (flag'1 = 2);
[outerloopguard] (flag1 = 0) & (loop2 = true) & f6 → (Loop'1' = true) & (flag'1 = 1);
[innerloopguard] (flag1 = 1) & (loop2 = true) & f7 → (Loop'1' = false) & (flag'1 = 0);
[decide] (flag1 = 2) → (flag'1' = 0);
[renewouterloopguard] (flag1 = 0) & (loop1 = true) & (loop2 = false) & f6 → (loop'1' = true) & (flag'1' = 1);

module
ContractionLoop
endmodule

[merge] (flag2 = 1) & (loop2 = true) → (flag'2 = 2);
[merge] (flag2 = 2) & (loop2 = true) → (flag'2 = 0);
[merge] (flag2 = 0) & (loop2 = false) & f8 → (Loop'2' = true) & (flag'2' = 1);
[innerloopguard] (flag1 = 1) & (flag2 = 0) & f3 → (Loop'2' = false) & (flag'2' = 3);
[merge] (flag2 = 0) & (loop2 = true) & f6 → (loop'2' = true) & (flag'2' = 0);

module
Counter
endmodule

[merge] (loop1 = true) & (loop2 = false) & (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 1);
[merge] (loop2 = true) & (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 2);
[merge] (loop1 = true) & (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 3);
[merge] (loop1 = false) & (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 4);
[merge] (loop1 = false) & (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 4);
[merge] (count = MAXCOUNT) → (terminate = true);

module
MincutImp
endmodule

mm:[0..MM] init MM;
ans:bool;
[decide] (flag1 = 1) & (loop1 = true) & (loop2 = true) & (count = MAXCOUNT) → (count' = count + 1) & (action' = 1);
[decide] (flag1 = 1) & (loop1 = true) & (loop2 = true) & (count = MAXCOUNT) → (count' = count + 1) & (action' = 2);
[decide] (flag1 = 1) & (loop1 = true) & (loop2 = true) & (count = MAXCOUNT) → (count' = count + 1) & (action' = 3);
[decide] (flag1 = 0) & (loop1 = false) & (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 4);
[decide] (flag1 = 0) & (loop1 = false) & (count + 1 ≤ MAXCOUNT) → (count' = count + 1) & (action' = 4);
[decide] (count = MAXCOUNT) → (terminate = true);

module
MincutLoop
endmodule

[merge] (ansmerge = false) & (aa = false) & f1 & f0
→ [mincut] (ans = false) & (aa = true) & f1 & f0
→ [mincut] (ans = true) & (aa = false) & f1 & f0
module
Counter
endmodule

[merge] (ansmerge = false) & (aa = false) & f1 & f0
→ [mincut] (ans = false) & (aa = true) & f1 & f0
→ [mincut] (ans = true) & (aa = true) & f1 & f0

Fig. 11. A YAGA-generated PRISM representation of the refinement in Fig. 9
8. Probabilistic Diagnostics of Dependability using YAGA

In this section we investigate how the use of probabilistic counterexamples can play a role in the analysis of dependability, especially in compiling quantitative diagnostics related to specific “failure modes”.

We shall assume a probabilistic model of the embedded controller in Fig. 15, and then use the notation and conventions set up in Sec.5. In addition, we shall reserve the symbol $F$ for a special designated state corresponding to “complete failure” of the system. In the first case (evidenced by the first refinement step) we shall posit that no more actions are possible after the system completely fails (i.e. enters the $F$ state). In the design of dependable systems, one of the goals is to understand what behaviours lead to complete failure, and how the design is able to cope overall with the situation where partial failures occur. For example, the design of dependable systems, one of the goals is to understand what behaviours lead to complete failure, and those combinations are usually referred to as failure modes. In such cases, dependability analysis would seek to confirm that the relevant failure modes were very unlikely to occur and also, to produce some estimate of the time to complete failure once the failure mode arose.

We first set out definitions of failure modes and related concepts relative to an MDP model. In the definitions below we refer to $P$ as an MDP, with $F$ a designated state to indicate “complete failure”, such that the annotation $\{F\} P \{F\}$ holds. Let $\phi$ be a predicate over the state space and $\alpha$ a sequence of states indicating an execution trace of $P$. We define the the path formula $\phi \alpha$ to be $(\phi \alpha)\land \alpha = \text{true}$ if and only if there is some $n \geq 0$ such that $\alpha.n$ satisfies $\phi$, corresponding to the usual definition of “eventuality” [HJ94].

Our next definition identifies a failure mode: it is a predicate which, if ever satisfied, leads to failure with probability 1. We formalise this as the conditional probability i.e. that $F$ occurs given that the failure mode occurs. We use the standard formulation for conditional probability: if $\mu$ is a distribution over an event space, we write $\mu.A$ for the probability that event $A$ occurs and $\mu.(A \land B)$ for the probability that event $A$ occurs given that event $B$ occurs. It is defined by the quotient $\mu.(A \land B)/\mu.B$.

Standard approaches for dependability analysis largely rely on the failure mode and effects analysis or (FMEA) [Int85] for identifying a “critical set” — the minimal set of components whose simultaneous failure constitutes a failure mode. Next we shall show how probabilistic model checking can be used to generalize this procedure.

**Definition 9.** Let $P$ be an MDP and let $\aleph$ be a scheduler; we say that a predicate $\phi$ over the state space is a failure mode for $\aleph$ if the probability that $F$ occurs given that $\phi$ ever holds is 1:

$$[P^K]s_0.\langle \phi \rangle \cdot \langle \phi \rangle = 1,$$

where we write $\text{Exp.}[P^K]s_0.\langle \phi \rangle$ as the conditional probability over traces such that $F$ is reachable from the initial state $s_0$ given that $\phi$ previously occurred. We say that $\phi$ defines a critical set if $\phi$ is a weakest predicate which is also a failure mode.

Given the assumption that once the system enters the state $F$, it can never leave it, Def. 9 consequently identify states of the system which certainly lead to failure.

Once a critical set has been identified, we can use probabilistic analysis to give detailed quantitative profiles, including the probability that it occurs, and estimates of the time to complete failure once it has been entered. The probability that a critical set $\phi$ occurs for a scheduler $\aleph$ is given by $\text{Exp.}[(P^K)_\aleph.\langle \phi \rangle)$. The next definition sets out the basic definition for measuring the time to failure — it is based on the conditional probability measured at various depths of the execution tree.

**Definition 10.** Let $P$ be an MDP, $\aleph$ a scheduler and let $K$ refer to the depth of the associated execution tree. Furthermore let $\phi$ be a critical set. The probability that complete failure has occurred at depth $K$ given that $\phi$ has occurred is given by:

$$[P^K]s_0.\langle \phi \rangle \cdot \langle \phi \rangle.$$

Thus even though a failure mode has been entered, the analysis can determine the approximate depth of computation $k \leq K$ before complete failure occurs.
8.1. Instrumenting model checking with failure mode analysis

In this section we describe how the definitions above can be realised within a probabilistic model checking environment in order to identify and analyse particular combinations of actions that lead to failure.\(^6\)

8.1.1. Identification of failure modes

The first task is to interpret Def. 9 as a model checking problem; this relies on the calculation of conditional probabilities which is not usually possible using standard techniques. However, adopting the more general expectations approach — instrumented as reward structures of MDPs — we are able to compute lower bounds on conditional probabilities after all.

Lemma 2. Let \( P \) be a \( pGCL \) program and \( \mathcal{N} \) a scheduler, \( X, C \) are predicates over \( S \), and \( \lambda \) is a real value at least 0. Starting from an initial state \( s_0 \), the following relationship holds.\(^7\)

\[
\text{Exp}[P_{\mathcal{N}}].s_0.((C \land X)) - \lambda \times (\mathcal{C}) \geq 0 \iff \text{Exp}[P_{\mathcal{N}}].s_0.(X \mid C) \geq \lambda.
\]

Proof 3. Follows from linearity of the expectation operator and the definition of conditional probability as \( \text{Exp}[P_{\mathcal{N}}].s_0.((C \land X))/\text{Exp}[P_{\mathcal{N}}].s_0.(C) \) provided that \( C \) has a non-zero probability of occurring.

From Lem. 2 we can see that (putting \( \lambda = 1 \)) if \( \text{Exp}[P_{\mathcal{N}}].s_0.((C \land X)) - \mathcal{C} \geq 0 \) then the conditional probability \( \text{Exp}[P_{\mathcal{N}}].s_0.(X \mid C) = 1 \). On the other hand, we can verify the expression \( \text{Exp}[P_{\mathcal{N}}].s_0.((C \land X)) - \mathcal{C} \geq 0 \) directly using YAGA’s output. Thus the following steps summarise our proposed method for failure mode analysis.

1. Use YAGA to identify a failure tree consisting of traces which terminate in \( F \).
2. From the failure tree identify candidate combinations of events \( C \) which correspond to traces terminating in \( F \).
3. Using YAGA’s output, verify that the candidate combinations \( C \) are indeed failure modes by evaluating the constraint \( \text{Exp}[P_{\mathcal{N}}].s_0.((C \land X)) - \mathcal{C} \geq 0 \) i.e. after setting \( \lambda = 1 \).
4. Compute expected times to failure for the identified failure modes.

In the next section we shall illustrate this technique on a case study of an embedded controller design.

8.1.2. Component-level repairs

Here, our intention is to attempt to recover a system from a failed configuration state such that persistent failure is no longer possible. We note that a failed system configuration arises due to inability of the system to recover from the failure of one or more of its components. To identify the failed components, we inspect the individual failure modes — the strongest predicates corresponding to a particular critical set \( \phi \). Let \( \text{repair}_K(c) \) be a repair predicate over the state space for a given failed system component \( c \) at associated depth \( K \) of our model. In addition, we use the boolean functions \( \text{active}(c) \) and \( \text{dead}(c) \) (queries a system for whether a component is active and dead respectively) to return the state of \( c \) at a time step \( k \leq K \). We define the repair predicate\(^8\) such that:

\[
\text{repair}_K(c) \triangleq \begin{cases} 
\text{true} & \text{if } \text{active}(cK) \mid \exists k < K : \text{dead}(cK) \\
\text{false} & \text{if } \text{dead}(cK) \mid \exists k < K : \text{dead}(cK)
\end{cases}
\]

Again, we shall reserve the notation \( \text{orepair}_K(c) \) to mean that \( \text{orepair}_K(c),\alpha = \text{true} \) if and only if there is some \( k \geq 0 \) such that \( \alpha.k \) satisfies \( \text{repair}_K(c) \) for a particular component \( c \). Also, we write \( P_r(\text{orepair}_K(c)) \) for the probability that the component \( c \) will eventually be repaired at \( k \). The next lemma captures the idea that the probability of failure of a repaired system configuration is at most its value before the repair.

---

\(^6\) Note that YAGA computes probabilities over endpoints rather than over traces, thus we assume that failure modes can be identified by entering a state which persists according to Def. 9. These will be deadlock states of the MDP being analysed.

\(^7\) This expression may be generalised to allow for non-determinism: \( \text{Exp}[P].s_0.((C \land X)) - \lambda \times (\mathcal{C}) \geq 0 \iff [P_\mathcal{N}].s_0.(X \mid C) \geq \lambda \) for any scheduler \( \mathcal{N} \). Note also that if \( C \) does not hold with a non-zero probability then this definition assumes that the conditional probability is still defined and is maximal.

\(^8\) Note that this definition only implements a one-step repair. Multi-level repairs are possible by storing the history of previous repairs.
Lemma 3. Let $P$ be an MDP, $N$ a scheduler and let $K$ refer to the depth of the associated execution tree. Furthermore let component $c$ be a failed component of a failure mode defined by the critical set $\phi$. The probability that complete failure can occur after repairing $c$ at depth $K$ given that $\phi$ holds is such that

$$[P^K_N]_{s_0}.(G(F \Rightarrow \diamond \text{repair}_K(c)) \mid \diamond \phi) \leq [P^K_N]_{s_0}.(\diamond F \mid \diamond \phi).$$

Observe here that we have been a bit cavalier with the “G” temporal operator to capture the fact that a failure configuration must “always” be followed by a component repair. Later on, we shall demonstrate how to use model checking techniques to investigate this statement for a one-step system repair.

However, an ideal repair mechanism will involve the recovery of multiple failed components $c_1, \ldots, c_n$ that are captured in failure modes directly inferred by $\phi$.

Definition 11. Let $fm(\phi)$ relate to a failure mode inferred by $\phi$; let $P$ be an MDP, $N$ a scheduler and let $K$ refer to the depth of the associated execution tree. Furthermore let failed components $c_1, \ldots, c_n$ be part of $fm(\phi)$. The probability that complete failure can occur after the repair predicate over $c_1, \ldots, c_n$ is given by

$$[P^K_N]_{s_0}.(G(F \Rightarrow (\diamond \text{repair}_K(c_1) \lor \ldots \lor \diamond \text{repair}_K(c_n))) \mid \diamond \phi) = [P^K_N]_{s_0}.(\diamond F \mid \diamond \phi) \cdot \prod_{c_i \in fm(\phi)} \Pr(\diamond \text{repair}_K(c_i))$$

Next we demonstrate this idea on the pB refinement of an embedded controller system with randomly failing components. The components of the controller are set to have known probabilities of failure and repair. Once the abstract pB description of the controller and its components is complete, we shall then proceed with a two-step refinement. The first level of refinement will represent the implementation of the controller. Afterwards we shall proceed with another level of refinement, this time, attempting to recover the failing components by incorporating a repair mechanism (an abstract pB machine that could try to repair failed controller components). Most importantly, at each level of refinement, we shall attempt to diagnose the controller’s safety specification with respect to Theorem 1. If any level of the refinement fails inductivity test with respect to the stated safety requirement, we shall compute diagnostic traces to explain the failure and then use the traces to further improve our understanding of the controller model. We now turn to the case study.

9. Case Study Two: An Embedded Control System with Randomly Failing Components

The case study in Fig. 12 was motivated by an earlier investigation by G"udemann and Ortmeier [GO10]. It consists of two redundant input sensors S1 and S2 measuring some input signal I. This signal is then processed in an arithmetic unit to generate the required output signal O. Two arithmetic units exist, a primary unit A1 and its backup unit A2. The primary unit gets an input signal from both sensors, and the secondary unit only from one of the two sensors. The sensors deliver a signal in discrete time intervals (but this requirement is not a key design issue since we assume that signals will always be propagated). If A1 produces no output signal, then a monitoring unit M switches to A2 for the generation of the output signal. A2 should only produce outputs when it has been triggered by M.
StatusCtx specifies the enumerated set of statuses STATUS, consisting of idle, active and dead.

Fig. 13. Machine StatusCtx

To analyse a model of the system, we shall incorporate the key dimensions of dependability — availability — the probability that a system resource(s) can be assessed; reliability — the probability that a system meets its stated requirement; safety — expresses that nothing bad happens. The requirements of the controller can be summarised as follows:

- **Functional requirements:** It is assumed that each of the controller component has a known availability; the reliability of the system is also known.
  - **Sensors:** It is assumed that the sensors will have three states: an ‘idle’ state, an ‘active’ state and a ‘dead’ state. A sensor can either become active or dead probabilistically after leaving its idle (initial) state. An active sensor must read signals from the input unit. A dead sensor can only allow time to advance while a repair mechanism probabilistically recovers it (makes it active again) or leaves it in its dead state.
  - **Actuators:** Are clearly identified for redundancy. It is assumed that the actuators will have three states: an ‘idle’ state, an ‘active’ state and a ‘dead’ state. An actuator can either become active or dead probabilistically after leaving its idle (initial) state. An active actuator must forward signals to the output unit. A dead actuator can only allow time to advance while a repair mechanism probabilistically recovers it (makes it active again) or leaves it in its dead state.
  - **Monitor:** It is assumed that the monitor will have three states: an ‘idle’ state, an ‘active’ state and a ‘dead’ state. The monitor can either become active or dead probabilistically after leaving its idle (initial) state. The monitor can only be triggered by the primary unit. An active monitor must forward signals to the secondary actuator. A dead monitor can only allow time to advance while a repair mechanism probabilistically recovers it (makes it active again) or leaves it in its dead state.
  - **A failed system must resume from its last known configuration:** This important requirement of the system ensures that once we reach a failure tree (evidenced by an initial failure configuration), we cannot restart the system but rather we attempt to repair the dead component(s) and then proceed with a re-try to output a signal.

- **Non-functional requirements:**
  - **Safety:** The safety property of interest also encapsulates the stated system reliability. Informally, we want to maintain that the system reliability must never be decreased by any operations of the controller model irrespective of the advance of time.
  - **Liveness:** Every failed controller component will eventually be repaired.

The states of the machine are given by the set constant STATUS in the abstract machine of Fig. 13.

### 9.1. An abstract pB model of a controller component

In this section we provided a simplified pB description of the components of the controller. To do this, we use a single machine (see Fig. 14) with signature Device(pp) where pp denotes the probability that a particular component or Device will be available upon request. The machine offers three abstract operations: an Init operation used to initialise a component; an Act operation used to probabilistically mark a component activate or dead; and finally a Repair operation which is used to probabilistically re-activate a dead component. The specification of the controller itself is shown in Fig. 15. In the next section we shall proceed with the first level of refinement of the specification. However, a key development method is to use YAGA to inspect that the safety requirement is maintained by any valid execution of the operations of the controller model.
MACHINE Device(pp)
SEES Real_TYPE, StatusCtx

CONTRAINTS

\[ pp \in \text{REAL} \land pp \geq \text{real}(0) \land pp \leq \text{real}(1) \]

VARIABLES sts

INARIANT

\[ sts \in \text{STATUS} \]

INITIALISATION

\[ sts := \text{idle} \]

OPERATIONS

\[
\text{Act} \triangleq \begin{cases} 
\text{PRE} & \text{sts} = \text{idle} \text{ THEN} \\
\text{PCCHOICE} & \text{pp} \text{ THEN} \\
\text{OF} & \\
\text{sts} := \text{active} \\
\text{sts} := \text{dead} \\
\text{END} \\
\text{END} \\
\text{END}; \\
\text{Init} \triangleq \text{BEGIN} \\
\text{sts} := \text{idle} \\
\text{END}; \\
\text{Repair} \triangleq \begin{cases} 
\text{PRE} & \text{sts} = \text{dead} \text{ THEN} \\
\text{PCCHOICE} & \text{pp} \text{ THEN} \\
\text{OF} & \\
\text{sts} := \text{active} \\
\text{sts} := \text{dead} \\
\text{END} \\
\text{END} \\
\text{END} \\
\end{cases}
\]

Device models a generic component with some probability of failure \( pp \), modelled as a parameter of the machine. The contraints state that \( pp \) is some real probability between 0 and 1 (inclusive). The state of the machine is modelled by a single variable \( sts \), representing the current status of the device. There are three operations: \text{Init}, \text{Act} and \text{Repair}, corresponding to initialising the device, how the device works probabilistically, and how the device can be repaired.

Fig. 14. Machine Device

MACHINE SignalTracker(s1p, s2p, a1p, a2p, mp, MAXCOUNT)
SEES Real_TYPE

CONTRAINTS

\[
s1p \in \text{REAL} \land s1p \geq \text{real}(0) \land s1p \leq \text{real}(1) \\
s2p \in \text{REAL} \land s2p \geq \text{real}(0) \land s2p \leq \text{real}(1) \\
a1p \in \text{REAL} \land a1p \geq \text{real}(0) \land a1p \leq \text{real}(1) \\
a2p \in \text{REAL} \land a2p \geq \text{real}(0) \land a2p \leq \text{real}(1) \\
mp \in \text{REAL} \land mp \geq \text{real}(0) \land mp \leq \text{real}(1) \\
\text{MAXCOUNT} \in \mathbb{N}
\]

CONSTANTS \( rr \)

PROPERTIES

\[ rr \in \text{REAL} \land rr \geq \text{real}(0) \land rr \leq \text{real}(1) \]

OPERATIONS

\[
\text{sgout} \leftarrow \text{SendSignal} \triangleq \begin{cases} 
\text{PRE} & (rr) \text{ THEN} \\
\text{ANY} & sg \text{ WHERE} \\
\text{(} & \text{[} \text{sg} = \text{true} \text{]} \text{] } \\
\text{THEN} & \text{sgout} := sg \\
\text{END} \\
\text{END} \\
\end{cases}
\]

SignalTracker contains a constant \( rr \) representing the probability of succeed for the entire system. \text{SendSignal} operation uses a “probabilistic specification statement” specifies that the signal is successfully sent \( sgout = \text{true} \) with probability of at least \( rr \).

Fig. 15. SignalTracker specification
9.2. Refinement step 1 - the controller behaviour

The first level of refinement of the controller specification (see Fig. 16) shows how to adapt standard $B$-style modelling of timing constraints [CMR07, But09] to $pB$ models. Next we show how the refinement can be analysed using our technique in order to compute the necessary diagnostics that define a critical set. Note that in the implementation we used an EXPECTATIONS clause of the form $q \Rightarrow p \times [(s \neq F)] \uplus [\text{success}]$, which supports the idea that the probability of reaching the ‘success’ state should exceed the given threshold $q$. Here $p$ is a parameter which may vary over the state, but which should initially be at least the value of $q$. Observe that $F$ denotes a persistent failure state (i.e. one where signal is lost).

To analyse the refinement we proceed as follows: (a) we translate the refinement into a PRISM MDP; (b) we then assign concrete availability to components of the controller as specified in the CONSTANTS clause of their abstract machine descriptions (see Fig. 14) such that for every component availability $\rho$ (i.e. $p\rho$), specified as a Device input parameter, we define an abstract template that will form a pool for selecting their concrete values. We let Low($L$), Normal($N$) and High($H$) describe the possible abstract values to be assigned to a component availability. We refer to the particular selection of these range of values for a single run of the machine as its availability index (see Fig. 17). Our first experiment shows how to use these values to address an incompleteness problem in the specification.

9.2.1. Experiment 1: a specification incompleteness problem

In the specification of the controller above, no clear mention was made whether or not the sensors are used to make provision for redundancy. This therefore leaves a system designer to guess whether or not the primary actuator should read signals from only one of the sensors or both of them before correctly outputting a signal. We shall refer to the first and second assumptions as the OR and AND design options respectively. We observe that this choice constitutes an incompleteness problem in the specification. But with YAGA, we can figure out which of the design options is most appropriate after interpreting the refinement model as a PRISM MDP and then using the tool’s experiment facilities to confirm the correct design option. Afterwards, YAGA can then be used to compute necessary diagnostics to confirm the design selection decision.

Fig. 18 shows how the effective reliability of the refined system compares with varying availability indexes for an initial reliability ($rr$) value of 99%. The experimental result provides a way to inspect whether or not the refinement conforms to the desired expectation: that is, whether or not it preserves the expression within the EXPECTATIONS clause — which says that the reliability value must never be decreased. However, as shown in the graphs of Fig. 18, this is not the case as the reliability of the system ($rr$) for both designs is diminished after the 6th execution time step. This therefore suggests a failure of both designs to satisfy the fundamental theorem of refinement Theorem 1.

To further understand the reason for the failure, we drill down to the component level by selecting the pool of values from availability index 3. That is, we set each component availability to 95% ($s1p = s2p = a1p = a2p = mp = 0.95$). The result we obtain by observing the last operation and state tuple from the diagnostic traces is shown in Fig. 19. It is interesting as it reveals that the failure traces of the OR design are clearly subsumed in those of the AND design and thus provide a lower probability of eventual signal loss with respect to the expression in the EXPECTATIONS clause. We are not only able to make this claim, in addition, we are also able to provide specific trace information and the overall probability that a failure would exactly correspond to the trace at a particular discrete time stamp. We also note that the maximum probabilities before failure are given by the PRISM PCTL formula

$$P_{\max} = \exists s g \leq MAXCOUNT (sg = 3). \quad (16)$$

Since we have now been able to identify a more fault tolerant design (the OR design) for the system, we shall adopt with this design henceforth and use it to complete the rest of our analysis. Our aim is to use the strength of YAGA’s automated refinement checking to further improve systems design with the intention of achieving an overall improved system reliability. The next experiment investigates a more interesting problem on how to identify the critical sets that are necessary to define complete system failure for the controller.
The main part of the implementation is a **WHILE**-loop until the time reaches the limit \( \text{MAXCOUNT} \). In each iteration, the system makes a step \( (\text{Action}) \), the time advances \( (\text{Tick}) \) and output \( \text{sgout} \) is updated to record if the signal has been sent successfully or not \( (\text{IsSucceed}) \). The called operations are from the imported machines \( \text{SignalProcess} \) and \( \text{Clock} \).
$Low(L): \quad L_1 \leq \rho \leq L_2$
$Normal(N): \quad N_1 \leq \rho \leq N_2$
$High(H): \quad H_1 \leq \rho \leq H_2$

$$st : L_2 < N_1 \land N_2 < H_1$$

### Availability Index for experimental investigation

<table>
<thead>
<tr>
<th>Index</th>
<th>S1 (s1p)</th>
<th>S2 (s2p)</th>
<th>A1 (a1p)</th>
<th>A2 (a2p)</th>
<th>M (mp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50% (Low)</td>
<td>50% (Low)</td>
<td>50% (Low)</td>
<td>50% (Low)</td>
<td>50% (Low)</td>
</tr>
<tr>
<td>2</td>
<td>75% (Normal)</td>
<td>75% (Normal)</td>
<td>75% (Normal)</td>
<td>75% (Normal)</td>
<td>75% (Normal)</td>
</tr>
<tr>
<td>3</td>
<td>95% (High)</td>
<td>95% (High)</td>
<td>95% (High)</td>
<td>95% (High)</td>
<td>95% (High)</td>
</tr>
</tbody>
</table>

**Fig. 17.** The template defines how to select concrete values for the components availability. Each selection per state is called an availability index. The table shows three availability indexes we shall use to investigate the system throughout our analysis.

**Fig. 18.** The graphs on the LHS and RHS compare the effective system reliability of the AND and OR system designs respectively. Observe the sharp drop of the values for the AND design (compare to OR design) from the original value of 0.99.

### 9.2.2. Experiment 2: identification of critical sets

Here we identify the critical sets for the chosen OR design necessary to define a complete system failure given the same availability index 3. To do this, we shall proceed as follows:

**Step 1:**
We set the parameters $q, p := 1$ in the expression $q \Rightarrow p \times [(s \neq F)] \sqcup \text{[success]}$ to identify all failure traces for chosen values of the components availability. Fig. 20 lists three of the failure traces relevant to our discussion, resulting in a maximum probability of failure of 0.0025 after the 6th execution time stamp (i.e. $\text{MAXCOUNT} = 6$).

### AND DESIGN - Effective Reliability = 98.30%    OR DESIGN - Effective Reliability = 98.75%

<table>
<thead>
<tr>
<th>Trace ID#</th>
<th>Last action plus state tuple</th>
<th>Probability</th>
<th>Trace ID#</th>
<th>Last action-state tuple</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND-01</td>
<td>SignalLoss (3,2,2,0,2)</td>
<td>1.300E-4</td>
<td>OR-01</td>
<td>SignalLoss (3,2,1,2,0,2)</td>
<td>1.200E-4</td>
</tr>
<tr>
<td>AND-02</td>
<td>SignalLoss (3,1,1,2,0,2)</td>
<td>2.260E-3</td>
<td>OR-02</td>
<td>SignalLoss (3,1,1,2,0,2)</td>
<td>2.260E-3</td>
</tr>
<tr>
<td>AND-03</td>
<td>SignalLoss (3,1,2,0,2)</td>
<td>2.380E-3</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**Fig. 19.** This information was extracted from the diagnostic trace summarising the loss of signal for both the AND and OR design options. Here we use only the last action plus state tuple computed from endpoint probability distributions corresponding to the failure of the controller to deliver an output signal (after setting parameter $\text{MAXCOUNT} = 6$). Observe that the state tuple in this case is given by $(s_g, s_1, s_2, a_1, a_2, m)$ where values of 0,1,2 and 3 denote idle, active, dead and signal loss respectively.
Sequence of operations leading to bad state ::>>>
\{
{INIT} (1,0,0,0,0),
{Sensor2Action} (1,0,1,0,0,0),
{PrimaryAction} (1,0,1,2,0,0),
{MonitorAction} (1,0,1,2,0,2),
{Skip} (1,0,1,2,0,2),
{Sensor1Action} (1,2,1,2,0,2),
{SignalLoss} (3,2,1,2,0,2)
\}
Probability mass of failure trace is:>>>> 0.00012

Sequence of operations leading to bad state ::>>>
\{
{INIT} (1,0,0,0,0),
{Sensor2Action} (1,0,2,0,0,0),
{Sensor1Action} (1,1,2,0,0,0),
{PrimaryAction} (1,1,2,2,0,0),
{MonitorAction} (1,1,2,2,0,2),
{Skip} (1,1,2,2,0,2),
{SignalLoss} (3,1,2,2,0,2)
\}
Probability mass of failure trace is:>>>> 0.00012

Sequence of operations leading to bad state ::>>>
\{
{INIT} (1,0,0,0,0),
{Sensor2Action} (1,0,1,0,0,0),
{PrimaryAction} (1,0,1,2,0,0),
{MonitorAction} (1,0,1,2,0,2),
{Skip} (1,0,1,2,0,2),
{Sensor1Action} (1,1,1,2,0,2),
{SignalLoss} (3,1,1,2,0,2)
\}
Probability mass of failure trace is:>>>> 0.00226

************ Finished Error Reporting ... ************** *

Fig. 20. Diagnostic feedback revealing single traces at endpoint probability distributions (after setting parameter MAXCOUNT = 6).

**Step 2:** From inspection of the above traces we notice that the failure of $A_1$ and $M$ enables us to identify them as potential candidates for the construction of our critical set.

**Step 3:** We verify that their failure will indeed result in overall failure by examining the value of the expectation $[F \land A_1 \land M] - [(A_1 \land M)]$.

For candidates such as $A_1$ and $M$, we use the diagnostic traces to calculate the conditional probabilities as in Def. 9. To do this we extract all the traces which result in $F$ and then examine the variations of the component failures in the traces to identify those which corresponded to a failure configuration.

The results were unsurprising and included for example, identifying that a simultaneous failure of the primary unit $A_1$ and the backup monitor $M$. On the other hand, once the $pB$ modelling was completed, the generation of the failure traces was automatic improving the confidence of full coverage.

9.2.3. Experiment 3: investigating time to failure

This experiment investigates the time to first occurrence of failure given a particular critical set. In fact, the results show that members of the set of interest are indeed critical after verifying their overall conditional probabilities of failure. Table 4 summarises this fact given that MAXCOUNT = 6 thus corresponding to endpoints distributions whose traces result in complete failure as in Fig. 20.

9.3. Refinement step 2 - recovering faulty components

Motivated by the assumption that a probabilistic model of the embedded controller in Fig. 15 is not allowed to remain in a persistent state of failure, we shall further refine the implementation in Fig. 16 by incorporating a repair mechanism that uses the $Repair$ function of the abstract $Device(pp)$ machine. As earlier stated, our intention is to improve probabilistic results — increasing the likelihood that signal will eventually be output.

As a first step, we need to identify what constitutes the failure modes for our original implementation and then attempt to factor them into our new refinement. Once a system has failed, we would then check the failure mode corresponding to that failure configuration. We do this in the MDP model of the refinement by encoding a failure recovery automaton to run side-by-side the system. The input parameter of the automaton at any one time is the set of failure modes that have occurred. The automaton then tries to probabilistically
### Identifying critical components time to first failure

<table>
<thead>
<tr>
<th>Critical Components</th>
<th>Time step to first failure</th>
<th>Maximum probability of failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1, S2</td>
<td>2 steps</td>
<td>2.5000 E-3</td>
</tr>
<tr>
<td>A1, M</td>
<td>3 steps</td>
<td>2.4938 E-3</td>
</tr>
<tr>
<td>A1, A2</td>
<td>4 steps</td>
<td>2.3691 E-3</td>
</tr>
<tr>
<td>A1, S2</td>
<td>3 steps</td>
<td>2.3750 E-3</td>
</tr>
</tbody>
</table>

Table 4. Maximum probabilities of failure are computed with respect to endpoint distributions of failure traces (Fig. 20) and conditional probabilities are given by Def. 9 — observe that the sum of the probabilities of the failure traces in Fig. 20 above = maximum probability of failure = 0.0025. This therefore makes A1 ∧ M critical. The same investigation follows for the other critical components S1 ∧ S2, A1 ∧ A2, etc...

### Failure modes & their maximum probability of occurrence, MAXCOUNT = 10

<table>
<thead>
<tr>
<th>Failure mode1</th>
<th>S1 active</th>
<th>S2 active</th>
<th>A1 idle</th>
<th>A2 dead</th>
<th>Time to failure</th>
<th>Maximum probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure mode2</td>
<td>S1 dead</td>
<td>S2 active</td>
<td>A1 ideal</td>
<td>A2 idle</td>
<td>4 steps</td>
<td>3.1250 E-3</td>
</tr>
<tr>
<td>Failure mode3</td>
<td>S1 dead</td>
<td>S2 active</td>
<td>A1 idle</td>
<td>A2 dead</td>
<td>4 steps</td>
<td>3.1250 E-3</td>
</tr>
<tr>
<td>Failure mode4</td>
<td>S1 dead</td>
<td>S2 active</td>
<td>A1 ideal</td>
<td>A2 dead</td>
<td>NULL</td>
<td>0.0000</td>
</tr>
<tr>
<td>Failure mode5</td>
<td>S1 active</td>
<td>S2 active</td>
<td>A1 dead</td>
<td>A2 dead</td>
<td>5 steps</td>
<td>1.4844 E-2</td>
</tr>
<tr>
<td>Failure mode6</td>
<td>S1 active</td>
<td>S2 active</td>
<td>A1 dead</td>
<td>A2 dead</td>
<td>5 steps</td>
<td>1.4844 E-2</td>
</tr>
<tr>
<td>Failure mode7</td>
<td>S1 active</td>
<td>S2 active</td>
<td>A1 dead</td>
<td>A2 active</td>
<td>5 steps</td>
<td>1.4844 E-2</td>
</tr>
<tr>
<td>Failure mode8</td>
<td>S1 dead</td>
<td>S2 dead</td>
<td>A1 active</td>
<td>A2 dead</td>
<td>NULL</td>
<td>0.0000</td>
</tr>
<tr>
<td>Failure mode9</td>
<td>S1 active</td>
<td>S2 dead</td>
<td>A1 active</td>
<td>A2 dead</td>
<td>0 steps</td>
<td>4.4443 E-2</td>
</tr>
</tbody>
</table>

Fig. 21. Description of the failure modes. Each failure mode was identified following the PRISM tool’s suggestion that are potential deadlock states. We observe that our critical sets can be derived from the failure modes via a combinatorial approach. Their probability to occur in the first level of refinement was computed by setting the availability of the components as follows: \( s_1 p = s_2 p = 0.5; \ a_1 p = a_2 p = 0.75 \) and \( mp = 0.95 \).

recover as many components as possible. Note that it is also possible that no component is recovered in which case time just advances. The \( pB \) model of the refinement is in Fig. 22.

The failure modes shown in Fig. 21 were identified by the PRISM tool after YAGA’s translation of the first level of refinement (Fig. 16) to a Markov Decision Process \(^9\). Each failure mode is a potential deadlock state of the model (\( i.e. \) after accounting for the precise system behaviour). Next we report a number of experimental results with respect to the new refinement. To do this we increase the likelihood of failure in the system refinement by assigning the availability of the components as follows: \( s_1 p = s_2 p = 0.5; \ a_1 p = a_2 p = 0.75 \) and \( mp = 0.95 \); we then set the parameters \( q, p := 0.99, 1 \) in the expression \( q \leftrightarrow p \times [(s \neq F)] \sqcup \text{success} \).

#### 9.3.1. Experiment 4: repaired Vs unrepaired system

Again, as in the case of the maximum probabilities of the failure traces to occur (Fig. 19), using the PCTL formula (16), we query the PRISM MDP model of the refinement for the maximum probability that complete failure will occur within MAXCOUNT. In particular, we observe that (see Fig. 23) the maximum probabilities of failure at the 6th and 7th time steps for the unrepaired system are 9.3748 E-3 and 9.8479 E-2 respectively. This is in contrast to those of the repaired system at the 7th and 8th time steps which are 1.1719 E-4 and 1.0693 E-2 respectively. Note that probabilities are constant once persistent failure has occurred. This result explains the general idea captured in Lem. 3 and the case where no persistent failure is possible can be achieved with a multi-step controller components repair.

#### 9.3.2. Experiment 5: inspecting a failure trace after repair

Here we observe (see Fig. 24) the effect of the repair mechanism on the refined system for one of the failure traces (out of a total of fourteen (14) traces) output by YAGA after the 7th time step (\( i.e. \) MAXCOUNT

\(^9\) We note that there can be many more failure modes leading to the definition of the critical sets. The choice of definition of what constitutes a failure mode for the system is purely based on design decisions.
Finally, we modify the implementation SignalTrackerI to allow devices to be repaired when the system has failed. The operation SendSignal now contains possibly three sequential steps:

- In the first step, the system is executed as long as the time allows and the system is not yet failed.
- In the second step, if the system is failed then all dead devices are repaired (probabilistically).
- In the last step, after repairing, the system is then restarted and executes until timeout.

Note that the last two steps are carried out only if in the first step, timeout is not yet reached, and as a result the system must be in a failure state. In other words, if in the first step the system has been successful then the system will stay as successful until timeout occurs.

**Fig. 22.** The implementation SignalTrackerI with repairing

= 7). We note that after the 5th time step, S1 and M were both active even though S2, A1 and A2 were all marked as dead (thus corresponding to critical set \( \phi = S2 \land A1 \)). On a closer inspection of the trace, we see that this failure configuration corresponds to failure mode 6 (see Fig. 21). But the execution of a recovery action (RepairMode6) at the 6th step only manages to ‘probabilistically’ recover A2. Unfortunately, this repaired configuration eventually led to signal loss as shown by the last action-state tuple of the trace, and according to Def. 11, that can occur with a probability of 0.00139 (= 1.4844 E-2 \( \times \) \( s2p \times a1p \times a2p \)) since S2 was never recovered.

### 9.3.3. Experiment 6: taking advantage of repair diagnostics

In our analysis of the fourteen diagnostic traces output by YAGA, we observed that sensor S1 gets repaired 50% of the time as opposed to sensor S2 which gets repaired only 33% of the time, that is within 7 time steps (MAXCOUNT = 7). Another usefulness of our technique is on how to use the trace diagnostic results to further improve software engineering confidence especially in the presence of tradeoffs. For example, since
Fig. 23. This graph compares the maximum probability of failure of the original implementation (unrepaired system) with that of its refinement (repaired system). We observe the marked improvement in the probability of eventual successful signal output. The result however establishes the goal of the second refinement step thus increasing the confidence of probabilistic refinement. We also note that the repaired system results in an overall effective reliability increase of 8.68% when compared to that of the unrepaired system at the same time step.

***** Starting Error Reporting for Failure Traces located on step 7 *****

Sequence of operations leading to bad state ::>>>

\[
\text{[\{INIT\} (1,0,0,0,0,0), \{Sensor2Action\} (1,0,2,0,0,0), \{Sensor1Action\} (1,1,2,0,0,0),} \\
\text{\{PrimaryAction\} (1,1,2,2,0,0), \{MonitorAction\} (1,1,2,2,0,1),} \\
\text{\{BackupAction\} (1,1,2,2,2,1), \{RepairMode6\} (1,1,2,2,1,1), \{SignalLoss\} (3,1,2,2,1,1)]}
\]

Probability mass of failure trace is:>>>> 0.00139

************ Finished Error Reporting ...

Fig. 24. Diagnostic feedback revealing single traces at endpoint probability distributions after component level repair).

we know that failure of both sensors S1 and S2 are critical to the system (see Fig. 4), one important question would be: which of the sensors is actually worth considering for maximum controller signal throughput given the constraint that a pB designer can only increase the availability of only one of the sensors and not both of them (assuming that he can also change the availability of the other components at will)? The answer to the question is in Table. 25. It lies in the effective reliability of the system for various configurations of component availability. We observe that when all other components have fixed availability as in case 2 and case 4, increasing the availability of sensor S2 will result in an overall increased reliability on the refined system provided that the availability of the other components is also maintained at a reasonably high level.

<table>
<thead>
<tr>
<th>Case</th>
<th>Effective System Reliability</th>
<th>Component Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>Case 1</td>
<td>0.9799</td>
<td>50%(L)</td>
</tr>
<tr>
<td></td>
<td>0.9855</td>
<td>50%(L)</td>
</tr>
<tr>
<td>Case 2</td>
<td>0.9899</td>
<td>50%(L)</td>
</tr>
<tr>
<td></td>
<td>0.9899</td>
<td>50%(L)</td>
</tr>
<tr>
<td>Case 3</td>
<td>0.9718</td>
<td>75%(N)</td>
</tr>
<tr>
<td></td>
<td>0.9682</td>
<td>75%(N)</td>
</tr>
<tr>
<td>Case 4</td>
<td>0.9896</td>
<td>50%(L)</td>
</tr>
<tr>
<td></td>
<td>0.9854</td>
<td>50%(L)</td>
</tr>
</tbody>
</table>

Fig. 25. Investigating the sensors.
10. Conclusion

This paper has explored how model checking techniques can be used to improve our understanding of probabilistic systems refinement in the probabilistic B language framework. Often times when we refine systems, our goal is to improve performance results with respect to the properties of interest. More so, with probabilistic systems refinement, we also want to guarantee improved probabilistic results — establish with certainty that a program expectation can never be decreased. For simple probabilistic specifications of the form \( A \mid v : B \), Hoang’s refinement rule comes down to checking that the probabilities that the outcome holds is all that is required.

We have also shown how this kind of reasoning can be adapted to investigating safety related issues as part of a controller modelling problem to enable a system designer cope with failure as best as possible. For standard (non probabilistic) control problems, there has been a huge success in exploring the causal relationships between the system subcomponents leading to a persistent state of failure or property violation. Very notable industrial techniques for this kind of analysis include the Failure Mode and Effect Analysis (FMEA) [Int85], Deductive Cause Consequence Analysis (DCCA) [ORS06], Hazard Operability Studies (HAZOP) [HAZOP], and the Fault Tree Analysis (FTA). Even though these methods have succeeded in analysing a number of industry-sized problems, they do not address the realistic viewpoint that it is possible to have control systems with randomly failing components. In these cases the quality of the computer system is no longer determined by qualitative correctness, and indeed, in some cases, complete correctness can never be guaranteed. Rather, the quality is determined by a measurement of performance or the probability that it is correct.

The methodology of abstraction and refinement can be used to elaborate quantitative safety specifications in embedded control systems design via stepwise development where the random behaviours of the controller components are encoded as purely probabilistic properties. Development then reduces to inspecting the safety specification via operational proofs over the pB models to establish consistency between the refinement steps. Unfortunately, for quantitative safety specifications (a focus of this paper), a human verifier has no way of inspecting that this requirement is met even though the pB automated prover readily establishes consistency between the refinements by discharging associated proof obligations. One way to resolve this uncertainty is to explore algorithmic approaches similar to probabilistic model checking techniques which can provide exact diagnostics summarising the failure (if indeed it exists) of the refinement goal.

The use of probabilistic model-based analysis to explore dependability features in systems construction has recently become a topical issue [KNP07, GCW07, GO10, AFGKLL09]. One way to achieve this is to use probabilistic counterexamples [HKD09, AL09, ADvR08] which can guarantee profiles refuting the desired property, that is, after visiting the reachable states of the supposedly ‘finite’ probabilistic system models. What we have done here is to show how a similar investigation can be achieved for the refinement of proof-based models by taking advantage of the state exploration facility offered by probabilistic model checking. Our method is very precise since it provides a way to inspect the goal of refinement — improving probabilistic results. However, if this does not hold then we are able to output exact diagnostics summarising the failure provided that computation resources are not scarce.

References


